The YASS Programming Language

A practical introduction to Yet Another Simple Syntax

Jamie Balfour

First working edition

Preface

This book was prepared with the assistance of artificial intelligence under the direction of Jamie Balfour. Its descriptions and examples are based on the official ZPE documentation and are intended to be valid for ZPE 1.14.11.

YASS is designed to make programming approachable without making useful programs unnecessarily small. This book is intended to teach the language through carefully chosen examples, practical exercises, and a steady progression from simple scripts to larger programs.

The examples in this first working edition target the ZPE/YASS environment. They are written to be read, modified, and run.

Contents

1 Introducing YASS **1**

1.1 A language for making things . . . 1
1.2 Three ways to write YASS . . . 1
1.3 ZPE's philosophy . . . 2
1.4 Running YASS through ZPE . . . 2
1.5 Choosing an editor . . . 3
1.5.1 Try it: choosing tools . . . 4
1.6 Your first program . . . 4
1.7 Comments . . . 4
1.7.1 Try it: first programs . . . 4
1.8 Chapter-wide review exercises . . . 5

2 Data Types **6**

2.1 The main data types . . . 6
2.2 Tuples . . . 6
2.3 Optional type declarations . . . 7
2.4 Types and later chapters . . . 7
2.5 Names and assignment . . . 8
2.6 Expressions . . . 8
2.7 Declarations and names . . . 8
2.8 The number type . . . 9
2.8.1 Try it: numbers and strings . . . 9
2.9 The string type . . . 9
2.10 Working with strings . . . 10
2.10.1 Inline documents and heredocs . . . 10

2.10.2 String functions . . . 11
2.11 The boolean type . . . 11
2.11.1 Try it: boolean values . . . 12
2.12 Lists of values . . . 12
2.13 Maps of named values . . . 13
2.14 Alternative values . . . 13
2.14.1 Try it: alternative values . . . 14
2.14.2 Try it: collections . . . 14
2.15 Changing a variable . . . 14
2.16 Constants and stable values . . . 14
2.17 Expressions and precedence . . . 15
2.18 Conversions and mismatched types . . . 15
2.19 Scope and lifetime . . . 15
2.20 Choosing useful names . . . 16
2.21 A complete example . . . 16
2.22 Chapter-wide review exercises . . . 16

3 Decisions and Repetition 18

3.1 Conditions and truth tables . . . 18
3.1.1 Exclusive or (XOR) . . . 19
3.2 Making a decision . . . 20
3.2.1 Try it: simple decisions . . . 20
3.3 Ternary expressions . . . 20
3.4 Several alternatives: elseif . . . 20
3.5 Match statements . . . 21
3.6 Matcher expressions . . . 22
3.7 Repeating a task . . . 22
3.8 The while loop . . . 23
3.8.1 Do–while loops . . . 23
3.8.2 Loop-until form . . . 24
3.9 The for-to loop . . . 24
3.9.1 Try it: numeric ranges . . . 25
3.10 The for-each loop . . . 25

3.10.1 Try it: visiting values . . . 25
3.11 Inline iteration . . . 25
3.12 The repeat loop . . . 26
3.12.1 Try it: decisions . . . 26
3.13 Adding a running total . . . 27
3.14 Calculating an average . . . 27
3.15 Finding a value . . . 27
3.16 Filtering a collection . . . 28
3.17 Processing a list of tasks . . . 28
3.18 Repeating a check . . . 29
3.19 Chapter-wide review exercises . . . 29

4 Collections **30**
4.1 Checking whether a collection is empty . . . 30
4.1.1 Lists . . . 30
4.1.2 List subscripting . . . 31
4.1.3 Counting values . . . 31
4.1.4 Maps . . . 32
4.1.5 Typed collections . . . 32
4.1.6 Map subscripting . . . 32
4.1.7 Testing whether an index or key exists . . . 33
4.1.8 Iterating through a map . . . 33
4.1.9 Arrays . . . 34
4.1.10 Ordered maps . . . 34
4.1.11 Choosing a collection . . . 35
4.1.12 Try it: loops and collections . . . 35
4.2 Choosing a loop . . . 35
4.3 Collection review exercises . . . 36

5 Functions **37**
5.1 Returning a value . . . 37
5.2 Defining a function . . . 37
5.3 Parameters and return values . . . 38

5.4 Union return types . . . 38
5.5 Named and default parameters . . . 39
5.5.1 Try it: parameter choices . . . 39
5.6 Functions that produce text . . . 39
5.7 The test function . . . 40
5.8 Functions and collections . . . 40
5.9 Functions as values . . . 40
5.10 Lambda functions . . . 41
5.11 Arrow functions . . . 41
5.12 Choosing a function design . . . 41
5.13 Chapter-wide review exercises . . . 41

6 Records 43

6.1 Defining a record . . . 43
6.2 Creating a record . . . 43
6.3 Reading and changing fields . . . 44
6.4 Records and maps . . . 44
6.5 Records and structures . . . 44
6.6 Exercises . . . 44

7 Structures and Objects 46

7.1 Defining a structure . . . 46
7.2 Creating objects . . . 46
7.3 Creating simple objects directly . . . 47
7.4 Built-in objects . . . 47
7.5 Constructors . . . 48
7.6 Methods and object behaviour . . . 48
7.7 Encapsulation . . . 49
7.8 Inheritance . . . 49
7.9 Scope and access control . . . 50
7.10 Structures and collections . . . 51
7.11 Structures as types . . . 51
7.12 Chapter-wide review exercises . . . 51

8 Interfaces **53**

8.1 Declaring an interface . . . 53

8.2 Multiple interfaces and checking . . . 53

8.3 Interface exercises . . . 54

9 Exceptions **55**

9.1 Trying an operation . . . 55

9.2 The Exception object . . . 55

9.3 Creating useful failures . . . 56

9.4 Choosing what to catch . . . 56

9.4.1 Try it: the library project . . . 57

9.5 Chapter-wide review exercises . . . 57

10 Enumerations **58**

10.1 Declaring an enum . . . 58

10.2 Nested enums . . . 58

10.3 Enumeration exercises . . . 59

11 Components and Templating **60**

11.1 Component declaration and use . . . 60

11.2 Backtick templates . . . 61

11.3 Exercises . . . 61

12 Scope **62**

12.1 Global and local variables . . . 62

12.2 Function scope . . . 63

12.3 Object scope and the `this` keyword . . . 63

12.4 Scope blocks . . . 64

12.5 Public, protected, and private scope . . . 64

12.6 Choosing a scope . . . 64

12.6.1 Scope review exercises . . . 64

13 GoTo and Labels **66**

13.1 A labelled loop . . . 66

13.2 Why structured loops are usually better . . . 66

13.3 Labels and decisions . . . 67
13.3.1 Try it: the library project . . . 67
13.4 Chapter-wide review exercises . . . 68

14 Modules **69**
14.1 Why use a module? . . . 69
14.2 Built-in modules . . . 70
14.3 Namespaces . . . 70

15 Directives and Annotations **72**
15.1 Documenting a function . . . 72
15.2 Compiler directives . . . 72
15.3 Permission requirements . . . 73
15.3.1 Try it: documenting the library project . . . 73

16 Importing Libraries and Including Code **74**
16.1 Importing a library . . . 74
16.2 Including source code . . . 74
16.3 Choosing between import and include . . . 75
16.3.1 Try it: split the library project . . . 75
16.4 Chapter-wide review exercises . . . 75

17 ZEQL **76**
17.1 A first query . . . 76
17.2 Filtering with where . . . 76
17.3 Predicate lambdas . . . 77
17.4 Queries and ordinary loops . . . 77
17.4.1 Try it: query a collection . . . 77
17.5 Exercises . . . 77

18 Unit Testing **79**
18.1 Assertions . . . 79
18.2 Comparing expected and actual values . . . 79
18.2.1 Try it: test a small function . . . 80
18.3 A small test suite . . . 80

18.4 Exercises 80

19 YASS and the Web **82**

19.1 A first YWP page 82

19.2 The REQUEST object 82

19.3 Forms and POST data 83

19.4 The RESPONSE object 83

19.5 Reusable pages 84

19.6 Routers 84

19.7 Running a local web site 84

19.8 Web chapter exercises 85

A Running ZPE **86**

A.1 Running a program 86

A.2 Useful modes 86

A.3 Version and documentation information 87

B The ZPE formal specification **88**

Chapter 1

Introducing YASS

1.1 A language for making things

YASS is a general-purpose language with a readable syntax and a broad standard library. In this chapter we will install the tools, run a first program, and learn the basic shape of a YASS program.

1.2 Three ways to write YASS

YASS supports three closely related ways of writing blocks of code. The first is a compact, C-inspired style that uses braces:

```
boolean ready = true
if (ready) {
    print("Begin")
}
```

The second uses YASS's keyword endings. The same block can be written with `end if`:

```
boolean ready = true
if (ready)
    print("Begin")
end if
```

The third is indentation-based. A colon introduces the block, and indentation shows where the block ends:

```
boolean ready = true
if ready:
    print("Begin")
```

These styles express the same language constructs. This book uses the keyword-ending style for instructional control-flow examples because the opening and closing structure is visible on the page, but readers will encounter all three styles in YASS programs. The styles may also be mixed where the language permits it; consistency within a program is usually the clearest choice.

1.3 ZPE's philosophy

ZPE is built around a simple idea: programming should be accessible, quick to run, and consistent across the tasks that a programmer commonly performs. YASS is therefore not only a small teaching language. It is the language at the centre of an environment that also provides an interactive interpreter, a command-line runtime, an editor, a debugger, web support, networking, databases, modules, and a broad standard library.

This philosophy affects the way YASS is designed. A beginner should be able to write a useful script without first assembling a large collection of dependencies, while an experienced programmer should be able to keep building when the script becomes a real application. Readability, direct feedback, and practical facilities are treated as complementary goals rather than competing ones.

The result is a language that supports procedural, functional, and object-oriented styles. It is deliberately flexible about syntax, but it still encourages clear names, explicit data types where they help, small functions, and programs whose structure can be understood by the next person who reads them. The examples in this book follow that principle: they introduce one idea at a time, then connect it to a useful program.

1.4 Running YASS through ZPE

YASS programs are run by the ZPE Programming Environment. ZPE provides the compiler, runtime, interactive interpreter, and the command-line tools used throughout this book.

On macOS or Linux, the web installer can download and install ZPE with:

```
curl -fsSL https://www.jamiebalfour.scot/projects/zpe/install.sh \
  | /bin/bash -s -- zpe
```

On Windows PowerShell, use:

```
irm "https://www.jamiebalfour.scot/projects/zpe/install.ps1?product=zpe" `
  | iex
```

If a web installer is unavailable or does not work on the system, use the JAR installer as a fallback. Download ZPE manually from the ZPE downloads page, place the JAR in a convenient directory, and run the following command from that directory:

```
java -jar zpe.jar --install
```

This starts the installer and displays the installation instructions. If the `zpe` command is already available, the same installation mode can be started with:

```
zpe --install
```

Once installation is complete, check that ZPE is available:

```
zpe --version
```

The version command should print the installed ZPE version. If the command cannot be found, restart the terminal or follow the path instructions printed by the installer.

ZPE also provides the exceptionally useful `zpe -man` command. It gives focused reference information without requiring a programmer to search through the entire documentation. For example, the first command describes the predefined `BluetoothDevice` object, while the second describes its `scan` method:

```
zpe --man BluetoothDevice
zpe --man BluetoothDevice::scan
```

The same form can be used to look up predefined functions and modules, making it a useful companion while learning or exploring a particular ZPE installation.

ZPE also protects sensitive operations with permission levels. Ordinary language features can be used at the lowest level, while operations involving resources such as files, networks, devices, or processes may require a higher level. A function can state its minimum requirement with an annotation such as `@requires permission 4`; this makes the access requirement visible to both the reader and the runtime. Permission levels are covered in more detail in the chapter on directives and annotations.

There are two useful ways to run YASS. To experiment at the command line, use interactive mode:

```
zpe -i
```

YASS source files use the official `.yas` extension. To run a program saved in a file, pass the file name to ZPE:

```
zpe hello.yas
```

The interactive interpreter is useful while learning because it gives immediate feedback. Files are more suitable once a program has a name, a purpose, and more than a few lines of code.

1.5 Choosing an editor

A terminal is enough to run YASS, but a suitable integrated development environment (IDE) can make learning much easier. Syntax highlighting separates keywords, data types, strings, and comments. Diagnostics can point out mistakes close to where they occur, while an integrated terminal makes it easy to run the current `.yas` file. A debugger can also pause a program, inspect variables, step through instructions, and continue execution.

Visual Studio Code can be used with a YASS plugin. It is a good choice if you already use VS Code or want a lightweight, familiar editor with YASS language support.

For a first-class ZPE and YASS workflow, ZIDE is recommended. It has close ties to both ZPE and YASS, so the editor and the runtime are designed to work together. In particular, ZIDE can manage ZPE debugging sessions with facilities such as breakpoints, stepping, variable inspection, and profiling. This reduces the gap between what the editor displays and what ZPE actually runs. More information about this relationship is given in the ZPE developer documentation.

The command line remains useful and should not be abandoned. It is transparent, easy to automate, and helps learners understand what an IDE is doing behind the scenes. A productive workflow is to write and inspect code in an IDE, then occasionally run the same `.yas` file with `zpe` from a terminal.

1.5.1 Try it: choosing tools

1. Open a YASS file in an IDE and identify its syntax highlighting.
2. Run the same file from the IDE and from a terminal, then compare the output.

1.6 Your first program

```
print("Hello, YASS")
```

This program asks YASS to display a string. It is small, but it already contains the essential pattern of programming: give the computer a precise instruction and observe the result.

1.7 Comments

Comments are notes for people reading the program. ZPE ignores them when it runs the code, so they do not change the program's result. A short comment can explain the purpose of a section or record an important decision, but comments should not simply repeat an obvious instruction.

Use `//` for a comment that continues to the end of the line:

```
// Display the first message.
print("Hello, YASS")
```

For a longer explanation, use a block comment enclosed by `/*` and `*/`:

```
/*
  This example keeps the message separate from the output
  so that it can be changed without changing the print call.
*/
string message = "Welcome to YASS"
print(message)
```

Comments are especially useful when explaining why a program has chosen a particular approach. Keep them accurate: an outdated comment can be more misleading than no comment at all.

1.7.1 Try it: first programs

1. Change the message so that it introduces you.
2. Print a three-line welcome message for a new user.

1.8 Chapter-wide review exercises

Build one small welcome-and-profile program, extending the same program after each exercise.

1. Print a welcome message and a project title.
2. Add a name and favourite colour using separate print statements.
3. Add a simple menu with three choices.
4. Add comments explaining the program's purpose.
5. Save the program as a `.yas` file and run it from the terminal.
6. Open the same program in an IDE and improve its layout and messages.

Chapter 2

Data Types

Programs work with values. YASS provides numbers, strings, lists, maps, and other useful kinds of data. Names allow us to keep hold of values so that we can use them again.

Every YASS value has a data type. A type describes what a value represents and which operations make sense for it. Understanding types early makes later decisions and loops easier to read: a condition produces a boolean, a counter is usually a number, and a collection gives a loop a sequence of values to process.

2.1 The main data types

The basic types used throughout this book are `number`, `string`, and `boolean`. Numbers represent quantities, strings represent text, and booleans represent either `true` or `false`. YASS also provides collection types such as `list`, `map`, and `array`. Lists store values by position, maps associate keys with values, and arrays provide a distinct fixed-size indexed collection. An ordered map extends the map idea with a defined iteration order, so entries can be processed in the order in which the map specifies them.

YASS and ZPE also support richer data types. Records describe data with a declared set of fields; structures are templates from which objects can be created; and expression queries represent lazy ZEQL results. These types are introduced here so that you recognise their names, but each has its own later chapter: records, structures and objects, and ZEQL should be studied there rather than treated as ordinary lists or maps.

2.2 Tuples

A tuple is a fixed-size, ordered group of values in which each position can have its own type. A tuple is useful when a function needs to return or pass several closely related values whose positions have a known meaning. Unlike a list, a tuple is not intended to grow and its element types are declared position by position.

Tuples are defined first and instantiated second:

```
person = (number, string)
person = (35, "Jamie")
print(person[1])
```

The first assignment declares that the tuple contains a number followed by a string. The second supplies values matching that shape. Tuples use subscripting like other indexable values, so position 1 contains the name in this example. They are also useful for multiple-return functions, which are discussed in the Functions chapter.

2.3 Optional type declarations

YASS can be written without explicit type declarations, but this book normally uses TYPO's optional strong typing. Writing the type beside a name makes the intended use visible and allows ZPE to identify incompatible assignments earlier.

```
string name = "Ada"
number temperature = 21
boolean outside = true
list scores = [72, 81, 95]
map colours = ["warm" => "orange", "cool" => "blue"]
```

The declaration does not change the value itself; it documents and checks the kind of value the name is intended to hold. When learning or experimenting, the declarations may be omitted, but explicit types are especially useful in larger programs and in the worked examples in this book.

An ordered map is still accessed by key like an ordinary map, but its ordering behaviour matters when producing reports, menus, serialised data, or other output where a stable sequence is important. Ordered maps receive fuller treatment in the Collections chapter.

2.4 Types and later chapters

The simple types are used immediately in expressions, conditions, and loops. Collections receive a dedicated chapter because their operations include subscripting, counting, iteration, and map traversal. Records, expression queries, structures, and objects receive their own chapters because each introduces concepts that go beyond storing a value.

YASS normally allows the type of a value to be inferred from the value assigned to it. This unbound style is concise:

```
x = 10
name = "Ada"
```

Explicit data types are also available when they make a program clearer or when a programmer wants additional checking. For example:

```
string name = "Ada"
number temperature = 21
```

Both styles are part of YASS. The unbound form is convenient for short programs and

experimentation, but this book uses explicit types throughout its examples so that each variable's intended kind of value is visible. We will return to types and the optional TYPO system later in the book.

2.5 Names and assignment

```
string name = "Ada"
number year = 1843
print(name)
print(year)
```

A name gives a value a role in the program. Good names make a program easier to read because they explain what its values mean.

2.6 Expressions

An expression produces a value. Expressions can contain literal values, names, operators, and function calls.

```
number width = 8
number height = 5
number area = width * height
print(area)
```

2.7 Declarations and names

A declaration introduces a name and tells YASS what kind of value the name is intended to hold. In this book we will normally declare variables with an explicit type. This makes examples slightly more verbose, but it also makes them easier to read: the reader can see the role of a value without having to work it out from a later assignment.

```
string title = "A short story"
number pages = 12
boolean published = false
```

The name `title` refers to a string, `pages` refers to a number, and `published` refers to a boolean value. The equals sign assigns the value on its right to the name on its left. It is not a mathematical claim that the two sides are permanently equal; it is an instruction to store a value under a name.

Names should describe the value they represent. A name such as `pages` gives useful information immediately, while a name such as `x` gives very little information outside a small calculation. Short names are appropriate for small, local counters, but meaningful names are usually better for data that survives beyond a few lines.

2.8 The number type

The `number` type is used for numeric values. It can represent values used for counting, measuring, and calculating.

```
number books = 3
number price = 8.50
number total = books * price
print(total)
```

The usual arithmetic operators are addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`). Parentheses make the intended order of evaluation clear.

```
number items = 4
number unit_price = 2.25
number delivery = 3
number cost = (items * unit_price) + delivery
print(cost)
```

Even when the precedence rules would produce the same answer, parentheses can be worthwhile. They help the reader confirm the structure of a calculation and make later changes safer.

Numbers can also be updated. The following program keeps a running total:

```
number total = 0
number first = 7
number second = 11
total = total + first
total = total + second
print(total)
```

The name `total` first refers to zero. The later assignments calculate a new value and make the name refer to that value. Assignment changes the value associated with a name; it does not change the meaning of the number type.

2.8.1 Try it: numbers and strings

1. Calculate the cost of two products and a delivery charge using typed variables.
2. Join a first name and surname into a complete greeting.

2.9 The string type

The `string` type represents text. Strings are written between quotation marks. A string may contain spaces and punctuation, and an empty string is still a valid string value.

```
string first_name = "Ada"
string last_name = "Lovelace"
string greeting = "Hello, "
print(greeting)
print(first_name)
print(last_name)
```

Strings are values, so they can be stored, passed to functions, placed in lists, and returned from calculations. A program should keep text as text until it genuinely needs to treat it as another kind of value.

String concatenation joins pieces of text. In YASS, the `&` operator performs string concatenation. The exact functions available for more advanced formatting are covered later, but a short example already shows the idea:

```
string subject = "YASS"
string sentence = "I am learning " & subject
print(sentence)
```

Be careful to distinguish a number from a string containing digits. `42` is a number, while `"42"` is text. They may look similar when printed, but they have different purposes and behave differently in expressions.

```
number answer = 42
string answer_text = "42"
print(answer + 1)
print(answer_text)
```

2.10 Working with strings

Strings become more useful when a program can measure, combine, search, and transform them. The `count` keyword reports the number of characters in a string, including spaces and punctuation.

```
string title = "YASS"
number title_length = count(title)
print(title_length)
```

The concatenation operator `&` combines strings. It is useful for constructing messages from several smaller values, but the pieces should be joined in a way that remains readable.

```
string first_name = "Ada"
string language = "YASS"
string message = first_name & " is learning " & language
print(message)
```

2.10.1 Inline documents and heredocs

Long strings are difficult to read when they are written as one quoted line with many escaped quotation marks and newline characters. ZPE's inline documents, sometimes

called heredocs, allow a multi-line document to be written directly in the source. They are useful for HTML fragments, reports, templates, and other blocks of text.

The exact delimiter form should be checked against the installed ZPE version, because the opening and closing markers must match. Inline documents can also contain values from the surrounding program, allowing a template to be built without manually concatenating every line.

```
string name = "Ada"
string report = <<
Welcome, name.
This is a multi-line report.
>>
print(report)
```

The example is deliberately shown as a document rather than a conventional quoted string: the line breaks remain visible, and the document can be edited like the text it represents. Before using an inline document in a project, test its delimiter and interpolation behaviour with the version of ZPE being used.

2.10.2 String functions

ZPE provides predefined functions for common string operations. These functions make the operation and the value being processed explicit.

```
string raw_title = "  Programming Notes  "
string title = raw_title.trim()
boolean mentions_yass = title.contains("YASS")
print(title)
print(mentions_yass)
```

Reference methods can be combined in larger expressions, while the value being operated on remains visible at the start of the expression. Check the official ZPE documentation for the methods available in the installed version.

Exercises

1. Count the characters in a user-entered string and report its length.
2. Build a greeting by concatenating a first name and a message.
3. Create a multi-line library notice using an inline document.
4. Trim a title and check whether it contains a chosen word using string functions.

2.11 The boolean type

The `boolean` type has two values: `true` and `false`. Booleans are especially useful when a program needs to record an answer to a yes-or-no question.

```
boolean logged_in = true
boolean has_permission = false
print(logged_in)
print(has_permission)
```

Comparison expressions produce boolean values. For example, a comparison can determine whether one number is greater than another.

```
number score = 75
number pass_mark = 50
boolean passed = score >= pass_mark
print(passed)
```

The common comparison operators are equality (`==`), inequality (`!=`), less than (`<`), greater than (`>`), less than or equal to (`<=`), and greater than or equal to (`>=`). A comparison should be read as a question: is `score` greater than or equal to `pass_mark`?

2.11.1 Try it: boolean values

1. Create a boolean recording whether a user is logged in.
2. Compare two numbers and store the result in a boolean variable.

2.12 Lists of values

A `list` stores an ordered collection of values. Lists are useful when several values belong together and their order matters.

```
list subjects = ["mathematics", "history", "computing"]
print(subjects)
```

The values in a list can be accessed by position. Positions are indexed from zero in the usual programming style, so the first element has index zero.

```
list subjects = ["mathematics", "history", "computing"]
string first_subject = subjects[0]
print(first_subject)
```

A list can contain numbers as well as strings:

```
list temperatures = [18, 21, 19, 23]
number morning = temperatures[0]
print(morning)
```

When a list is used in a larger program, its intended contents should be made clear by the surrounding names and operations. Later chapters will cover iteration and the standard functions for adding, removing, and searching for list values.

2.13 Maps of named values

A `map` stores values associated with keys. A list is usually selected by position; a map is selected by a meaningful key. In a map literal, the `=>` operator connects each key to its value. This tells ZPE that the bracketed expression is a map rather than an ordinary list.

```
map person = [
    "name" => "Ada",
    "language" => "YASS",
    "active" => true
]
print(person)
```

Maps are useful when the parts of a value have names. A record describing a book, for example, is easier to understand when its title and author are named rather than remembered as positions two and three in a list.

```
map book = [
    "title" => "Programming Notes",
    "pages" => 128,
    "available" => true
]
```

The exact operations for reading and changing map entries will be introduced with the rest of YASS data handling. The important distinction at this stage is conceptual: lists answer "which value is at this position?" while maps answer "which value belongs to this key?"

2.14 Alternative values

Map lookups and other expressions may not always produce the value a program wants. A key might not exist, an optional value might be unset, or an input might be empty. YASS provides the `??` alternative-value operator for these situations. It evaluates the expression on its left and uses the expression on its right as a fallback when the left-hand value is unavailable.

```
string search = input("Please insert the username you are looking for.")

map users = [
    "jamie1" => "Jamie Balfour",
    "john1" => "John Balfour"
]

print(users[search] ?? "User not found")
```

If the user enters `jamie1`, the map lookup produces "Jamie Balfour". If the key is not present, the lookup is unavailable and the program prints "User not found" instead. This keeps a simple fallback close to the expression that needs it. For several different actions or more involved conditions, use an `if` statement; alternative values are intended for choosing between a primary value and a default.

2.14.1 Try it: alternative values

1. Add a third user to the map and test both a known and an unknown username.
2. Change the fallback message so that it includes the search term.

2.14.2 Try it: collections

1. Create a list of five temperatures and print its first value.
2. Create a map describing a book with a title, author, and page count.

2.15 Changing a variable

A variable can be assigned a new value of the same type. This is useful when a value represents a changing state.

```
number attempts = 0
attempts = attempts + 1
attempts = attempts + 1
print(attempts)
```

The right-hand side is evaluated before the assignment takes effect. Thus `attempts = attempts + 1` means: read the current value, add one, and store the result back under the same name.

Typed variables make accidental changes easier to notice. If a name represents a number, assigning an unrelated string to it is a sign that the program's design needs attention. Explicit types therefore serve as documentation as well as a source of compiler or runtime checks.

2.16 Constants and stable values

Some values should not change after they have been established. YASS provides the `const` declaration for this purpose. A constant is named once and cannot be reassigned later in the program. For example, a pass mark is normally a stable value during one run of a program.

```
const pass_mark = 70
number score = 72
boolean passed = score >= pass_mark
print(passed)
```

The value assigned to `pass_mark` gives the constant its number value, while `const` prevents it from being reassigned. If a later statement attempted to assign a new value to `pass_mark`, ZPE would report that the constant cannot be changed.

Separating a constant from the calculation gives it a name and makes the program easier to adapt. If the pass mark changes between versions of a program, one declaration can be changed instead of searching through several unrelated expressions.

2.17 Expressions and precedence

An expression may contain several operators. YASS evaluates multiplication and division before addition and subtraction, while parentheses can be used to make a different grouping explicit.

```
number a = 2 + 3 * 4
number b = (2 + 3) * 4
print(a)
print(b)
```

The two variables have different values because the parentheses change the grouping. Good programming style favours expressions that can be understood quickly. If an expression requires a long explanation, split it into named intermediate values.

```
number width = 12
number height = 8
number area = width * height
number border = 2 * (width + height)
print(area)
print(border)
```

The names `area` and `border` give the two calculations a meaning that would be hidden in one large expression.

2.18 Conversions and mismatched types

Programs often receive text from a user, a file, or a network connection. Such input may need to be converted before it can be used as a number. Conversion should be explicit: a string containing digits is not automatically the same thing as a number in every context.

```
string input_text = "27"
number quantity = value(input_text)
number total = quantity * 3
print(total)
```

When conversion can fail, a robust program should validate the input and decide what to do when it is not a valid number. Error handling and input validation receive fuller treatment in later chapters. The principle is worth learning now: do not silently assume that external text has the type your calculation requires.

2.19 Scope and lifetime

A name is available only in the part of a program where it has been declared. This region is its scope. Keeping names close to where they are used reduces the number of values a reader must remember at once.

```
number width = 6
number height = 4
number area = width * height
print(area)
```

As programs grow, functions and structures create additional scopes. A name declared inside one function should not normally be treated as though it were available everywhere. This separation prevents unrelated parts of a program from changing one another's data accidentally.

2.20 Choosing useful names

Naming is a design activity, not a cosmetic one. A useful name answers a question for the reader. Prefer `temperature` to `t` when the value represents a temperature, and prefer `maximum_attempts` to `n` when the value limits repetition.

```
number maximum_attempts = 3
number attempts_used = 0
boolean can_try_again = attempts_used < maximum_attempts
print(can_try_again)
```

Names should also be consistent. If a program uses `first_name`, it should not later switch to `forename` for the same concept without a reason. Consistency makes searching, reading, and maintaining a program easier.

2.21 A complete example

The following example brings together several ideas from this chapter. It keeps the data explicitly typed, gives each value a useful name, and uses intermediate results to make the calculation readable.

```
string customer = "Ada"
number notebooks = 2
number notebook_price = 4.50
number delivery_charge = 2
number goods_total = notebooks * notebook_price
number final_total = goods_total + delivery_charge

print(customer)
print(final_total)
```

The program is small, but its structure scales. A larger program may obtain the customer and quantities from input, calculate several charges, and write a receipt to a file. The names and types still provide the same foundation.

2.22 Chapter-wide review exercises

Build one typed order-and-receipt program, extending the same data model throughout.

1. Store a customer's name and one product using typed variables.
2. Add quantity, unit price, and delivery charge.
3. Calculate the goods total and final total.
4. Add a pass/fail or budget boolean to the order.
5. Represent several products with a list or map.
6. Test the receipt with a free-delivery order and an order with delivery.

Chapter 3

Decisions and Repetition

Most useful programs must choose between alternatives and repeat actions. YASS provides conditional statements and loops for these purposes.

3.1 Conditions and truth tables

An `if` statement makes its decision from a condition. A condition is an expression whose result is either `true` or `false`. Comparisons such as `>` and `==` produce boolean values, and logical operators combine those values into a larger condition.

Truth tables provide a precise way to examine a logical condition. The table below shows the result of the `and` operation: both inputs must be true for the combined condition to be true.

`left`	`right`	`left and right`
false	false	false
false	true	false
true	false	false
true	true	true

The `or` operation is true when at least one input is true, while `not` reverses a boolean value. Thinking in terms of a truth table is especially useful when a decision contains several conditions, because it helps reveal cases that might otherwise be missed.

The three basic logical operations can be summarised together. The `and` operator requires both inputs to be true; `or` requires at least one input to be true; and `not` changes true to false and false to true. The following table shows the two-input operations side by side.

`left`	`right`	`left and right`	`left or right`
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

It is often helpful to read a truth table as a list of every possible input combination. For two boolean inputs there are four rows. For three inputs there would be eight rows, which

is one reason that naming intermediate conditions can make a complicated decision easier to inspect.

3.1.1 Exclusive or (XOR)

Exclusive or, usually abbreviated to XOR, is true when exactly one of two inputs is true. It is false when both inputs are false, and it is also false when both inputs are true. This makes XOR different from ordinary `or`: ordinary `or` accepts the case where both inputs are true, whereas XOR excludes it.

`left`	`right`	`left XOR right`
false	false	false
false	true	true
true	false	true
true	true	false

YASS can express XOR by combining the operators already introduced: one input or the other must be true, but they must not both be true.

```
boolean left = true
boolean right = false
boolean exclusive = (left or right) and not (left and right)
print(exclusive)
```

This form is deliberately written out because it makes the rule visible and does not depend on a separate operator name. It can be placed in a function later if a program needs to perform the same test repeatedly.

```
boolean has_ticket = true
boolean is_invited = false
boolean may_enter = has_ticket or is_invited
print(may_enter)
```

Exercises

1. Draw a truth table for `or`.
2. Write a condition that requires a user to have both a ticket and an invitation.
3. Use the completed XOR truth table above to explain why only the two rows with exactly one true input produce `true`.
4. Translate your XOR truth table into YASS using `and`, `or`, and `not`. Test all four input combinations and check that the output matches your table.

3.2 Making a decision

```
number temperature = 21

if (temperature > 20)
    print("It is warm")
else
    print("It is cool")
end if
```

Exercises

1. Change the temperature so that the other branch runs.
2. Add a third branch for a temperature exactly equal to 20.

3.2.1 Try it: simple decisions

1. Print whether a number is positive, negative, or zero.
2. Use an `if-else` statement to report whether a person may enter an event.

3.3 Ternary expressions

When a decision only needs to choose one of two values, a ternary expression can be more compact than a complete `if-else` statement. It places the condition first, followed by the value for the true case and the value for the false case.

```
number temperature = 21
string description = temperature > 20 ? "warm" : "cool"
print(description)
```

The ternary expression is useful when selecting a label, a message, or another small value. It should not be used to hide several statements or complicated logic: when the decision needs actions, an ordinary `if-else` statement is clearer.

Exercises

1. Use a ternary expression to choose “adult” or “child” from an age.
2. Rewrite a short two-branch value selection using a full `if-else` statement, then compare the two forms.

3.4 Several alternatives: elseif

An `if` statement becomes especially useful when there are more than two possible outcomes. YASS uses `elseif` to test another condition only when the earlier conditions were

false. The order matters: the first condition that succeeds selects the branch, and the remaining branches are skipped.

```
number score = 75

if (score >= 90)
    print("Excellent")
elseif (score >= 70)
    print("Pass")
elseif (score >= 50)
    print("Resit")
else
    print("Fail")
end if
```

This is clearer than writing several independent `if` statements. Independent statements could all run, while an `if-elseif-else` chain expresses one choice between alternatives. It is also important to put the most restrictive tests first. If the test for `score >= 50` came first, a score of 95 would be labelled `Resit` and the later tests would never be reached.

Use `elseif` when the alternatives are conditions or ranges. When the program is choosing between fixed values, `switch` and `when-is` provide more specialised forms of the same general idea. They will be introduced alongside their exact syntax after the basic `if` family is familiar.

Exercises

1. Add a distinction for scores of 100 or above.
2. Explain why the tests should be ordered from highest to lowest.

3.5 Match statements

The `match` statement is a more advanced way to select a result from several fixed cases. It is especially useful when one value must be compared with several possible literals. Unlike a long chain of conditions, the value being examined is written once and each branch states the result associated with it.

```
number status_code = 404

string message = match(status_code :
    200 => "OK";
    404 => "Not found";
    500 => "Server error";
    else => "Unknown status";
)

print(message)
```

The selected expression is evaluated immediately and assigned to `message`. The `else` branch is optional, but it is usually a good idea when the input may have a value not

listed explicitly. Match branches end with semicolons, including the fallback branch. Use `match` for immediate selection; a reusable `matcher` value is a separate, more advanced feature that belongs later in the book.

Exercises

1. Use `match` to turn a number from 1 to 7 into the name of a day.
2. Add a fallback for an invalid day number.

3.6 Matcher expressions

A `matcher` is a reusable callable value built from fixed match branches. Unlike `match`, which selects a result immediately, a matcher can be stored and called repeatedly with different input values. Its keys must be literals, and an optional `else` branch handles values that have no matching key.

```
status_lookup = matcher(
    200 => "OK";
    404 => "Not found";
    else => "Unknown";
)

print(status_lookup(404))
print(status_lookup(200))
```

The matcher returns “Not found” for 404 and “OK” for 200. Because the matcher is a value, it can be passed to a function, returned from a function, or stored as an object property. Use `match` when one immediate selection is needed; use `matcher` when the same fixed lookup is needed repeatedly.

Exercises

1. Create a matcher that converts the numbers 1, 2, and 3 into the words “one”, “two”, and “three”.
2. Add an `else` branch for an unsupported number.
3. Write a function that accepts a matcher and a value, then returns the matcher’s result.

3.7 Repeating a task

```
for (number i = 1; i <= 5; i++) {
    print(i)
}
```

Loops are valuable because they let a short description stand for many operations. The program above prints five numbers without requiring five separate instructions.

3.8 The while loop

A `while` loop repeats its body as long as its condition is true. The condition is checked before each iteration, so the body may run zero times.

The update is essential. A loop whose condition never becomes false will continue indefinitely. It is often helpful to identify the value that controls the loop, initialise it before the loop, and update it visibly inside the body.

```
number attempt = 1

while (attempt <= 3)
    print("Attempt " & attempt)
    attempt = attempt + 1
end while
```

This is a good place for a `while` loop because the program does not need a collection of values. It needs to continue until a changing condition is no longer true. The same pattern can be used for polling a service, reading records until the end of a file, or repeating validation until input is acceptable.

Unlike a `for` loop, a `while` loop does not automatically advance a counter. The programmer must make progress explicit. Before running a while loop, ask three questions: what value controls the condition, what makes the condition eventually false, and what happens if the condition is already false when the loop begins?

YASS also provides related forms for situations where the body must run before the condition is tested, or where the program should continue until a condition becomes true. Those forms are useful, but the ordinary `while` loop is the clearest starting point and should be mastered first.

3.8.1 Do–while loops

A do–while loop is useful when the body must run at least once. It places the condition after the body, so the first iteration happens before the test is made. The loop then repeats while the condition remains true.

```
number attempt = 0

do
    attempt = attempt + 1
    print("Attempt " & attempt)
loop while (attempt < 3)
```

This differs from an ordinary `while` loop. If the condition were false before an ordinary `while` loop began, its body would be skipped. A do–while loop performs its body once before it considers whether another iteration is needed. It is therefore a good fit for menus, retry prompts, and other interactions that must be shown at least once.

3.8.2 Loop-until form

The related `loop until` form expresses the same general idea from the opposite direction: the body continues until its condition becomes true. The condition describes the stopping event rather than the condition for continuing.

```
number attempt = 0

loop until (attempt >= 3)
    attempt = attempt + 1
    print("Attempt " & attempt)
end loop
```

Read the condition carefully. This loop stops when `attempt` reaches 3; it does not mean "loop while attempt is at least 3". Writing the stopping condition directly can make retry and polling code easier to read, but the update that moves the program toward that condition is still essential.

Exercises

1. Change the do–while example so that it makes five attempts.
2. Rewrite the do–while example using an ordinary `while` loop. Explain how you would ensure that the body still runs at least once.
3. Change the `loop until` example so that it stops after five attempts.
4. Write a `loop until` program that increases a number until it reaches 100.

Exercises

1. Change the loop so that it makes five attempts.
2. Write a while loop that counts down from 10 to 1.

3.9 The for-to loop

The `for-to` form is useful when a loop follows a numeric range. It keeps the counter and its range together.

```
for (number i = 1 to 3) {
    print(i)
}
```

Use a range loop when the program is concerned with a sequence of numbers. Use a `while` loop when the stopping condition depends on an event or a changing state rather than on a simple range.

Exercises

1. Use a `for-to` loop to print the numbers from 1 to 10.
2. Use a range loop to print room numbers from 101 to 105.

3.9.1 Try it: numeric ranges

1. Use a `for-to` loop to print the numbers from 1 to 10.
2. Use a range loop to print the room numbers from 101 to 105.

3.10 The for-each loop

The `for-each` form visits the values in a list or another iterable collection. Unlike a numeric loop, it does not require the program to manage an index.

```
list colours = ["red", "green", "blue"]

for each (colour in colours)
    print(colour)
end for
```

The loop variable is supplied by the loop construct and takes the type of the values being visited. This makes `for each` a natural choice when the program needs the values themselves rather than their positions.

Exercises

1. Print every item in a list of shopping tasks.
2. Print a greeting for each name in a list.

3.10.1 Try it: visiting values

1. Use `for each` to print every colour in a list.
2. Use `for each` to print a greeting for every name in a list.

3.11 Inline iteration

Sometimes a program needs the values produced by a loop inside an expression rather than a separate block of statements. YASS supports inline iteration for this purpose. The `from` expression creates a range of values that can be consumed immediately.

```
print(from 1 to 6)
```

The upper bound is exclusive, so extttfrom 1 to 6 produces 1 through 5. This is useful for passing a generated range to another operation or for applying a short transformation. An inline iteration can also use a function to describe how each value should be changed:

```
print(from 1 to 6 : function(number value) : number
    return value * 2
end function)
```

Inline iteration is concise, but it should not replace an ordinary loop when the body needs several statements, changes program state, or contains important decisions. Use a named loop when the steps need to be read and understood individually.

Exercises

1. Print the values from 10 to 15 using inline iteration.
2. Generate the first five doubled values from a range.

3.12 The repeat loop

YASS also provides the newer `repeat` loop for repeating a block a specified number of times.

```
repeat 3 times
    print("Again")
end repeat
```

The repeat form is useful when the number of repetitions is the main idea and no counter is needed by the body. A `for` loop is usually clearer when the body needs the counter; a `while` loop is usually clearer when the program should continue until a condition changes.

The repeat loop is deliberately introduced here in its count-controlled form. Condition-controlled repetition will be treated alongside the other loop-until forms once the basic loop families are established.

Exercises

1. Change the loop to display the reminder five times.
2. Explain when a repeat loop is clearer than a for loop.

3.12.1 Try it: decisions

1. Use an `elseif` chain to classify a score as excellent, pass, resit, or fail.
2. Write a temperature classifier with at least three ranges.

3.13 Adding a running total

One of the most common uses of a loop is to combine many values into one result. For example, a small shop might need the total value of several items in a basket. The accumulator is initialised before the loop and updated once for each item.

```
list prices = [3.50, 8.00, 2.25, 4.75]
number total = 0

for each (price in prices)
    total = total + price
end for

print(total)
```

The initial value matters. A total should begin at zero, because zero is the identity value for addition. The loop then has a simple responsibility: take the next price and add it to the total.

3.14 Calculating an average

The same pattern can calculate an average. The program keeps both a running total and a count of the values it has visited.

```
list temperatures = [18, 21, 19, 23, 20]
number total = 0
number item_count = 0

for each (temperature in temperatures)
    total = total + temperature
    item_count = item_count + 1
end for

number average = total / item_count
print(average)
```

This is a useful general pattern for measurements, scores, prices, and sensor readings. In production code, also consider the empty-list case before dividing: a list with no values has no meaningful average.

3.15 Finding a value

Loops are also useful for searching. The following program looks for an item in a list of permitted roles. The boolean variable records what the loop has discovered.

```
list roles = ["author", "editor", "reviewer"]
string requested_role = "editor"
boolean found = false

for each (role in roles)
    if (role == requested_role)
        found = true
    end if
end for

if (found)
    print("Role is available")
else
    print("Role is not available")
end if
```

Once a match has been found, a later version of this program could stop the loop early. The important design decision is to keep the search state explicit rather than relying on an unexplained change to the list.

3.16 Filtering a collection

A loop can also select values that satisfy a rule. This example prints only the low-stock products from a list of stock levels. The parallel lists make the example concrete; later chapters introduce structures that keep related fields together.

```
list stock_levels = [12, 2, 0, 7]
list products = ["pens", "notebooks", "folders", "rulers"]
number position = 0

for each (stock in stock_levels)
    if (stock < 5)
        print(products[position])
    end if
    position = position + 1
end for
```

Here the loop walks through the stock levels while a separate counter identifies the corresponding product. This works, but it also shows why a map or structure is often a better representation when several values belong to the same item.

3.17 Processing a list of tasks

In a task-processing program, the body of a loop often performs an operation rather than merely printing a value. The following example represents a list of tasks and reports each task as it is processed.

```
list tasks = ["backup files", "send report", "check messages"]

for each (task in tasks)
    print("Processing: " & task)
end for
```

The loop does not need to know how many tasks there are. Adding another task to the list automatically gives the program another item to process.

Exercises

1. Add two more tasks to the list.
2. Change the message so that it reports the task number as well as its name.

3.18 Repeating a check

The repeat loop is useful when an action must be attempted a known number of times. For example, a program might check a service three times before reporting that it needs attention.

```
repeat 3 times
    print("Checking service")
end repeat
```

In a real program, the print statement would be replaced by a status check. If the check can succeed early, a later example can combine the repeat loop with a condition and a break statement. Keeping the count separate from the action makes the retry policy easy to change.

3.19 Chapter-wide review exercises

Build one small task-processing program, adding each new control-flow idea to the same program.

1. Add a decision that reports whether a task is urgent.
2. Add an `elseif` chain for task priority.
3. Process several tasks with a `for each` loop.
4. Add a retry using a `while` or `repeat` loop.
5. Count completed tasks and report the total.
6. Test the program with an empty task list and a list containing urgent tasks.

Chapter 4

Collections

Loops become especially useful when they operate on collections. A collection is a value that holds several related values. ZPE provides several collection types, each suited to a different problem: lists, arrays, maps, and ordered maps. Choosing the right one makes both the program and its intent easier to understand.

4.1 Checking whether a collection is empty

The `empty` keyword checks whether a collection contains no values. It can be used with lists, maps, strings, and other values that have a meaningful notion of emptiness. This is often clearer than comparing a count with zero, because it states the question the program is asking directly.

```
list tasks = []

if (empty(tasks))
    print("There are no tasks")
else
    print("Tasks are waiting")
end if
```

An empty check is particularly useful before processing input or entering a loop. It lets a program distinguish between "there is nothing to do" and "there are values to process" without assuming that every collection uses the same representation.

Exercises

1. Check whether a list of library books is empty before printing its contents.
2. Check whether a user's search string is empty before using it as a map key.

4.1.1 Lists

A list is an ordered, flexible collection. It is a good choice when the program needs to keep a sequence of values and may add or remove elements as it runs.

```
list tasks = ["backup files", "send report", "check messages"]

for each (task in tasks)
    print(task)
end for
```

The position of each value is significant, and the list can be traversed from beginning to end. Lists are often used for search results, queues of work, lines read from a file, and groups of values returned by a function.

4.1.2 List subscripting

Subscript notation uses square brackets to select an element from a list. The subscript is normally a numeric position, making it possible to read a particular value without traversing the entire collection.

```
list books = ["YASS Guide", "Programming Notes", "Algorithms"]
string second_book = books[1]
print(second_book)
```

List subscripts are zero-based, so the second list item is at position 1. A program should still consider whether a requested list position exists before relying on the result.

Exercises

1. Use `is_set` before reading the last possible index of a list.
2. Check whether a map contains a requested username before printing its value.
3. Add a missing-key message to the map example.

Exercises

1. Print the first and last values in a list of three book titles.
2. Use a list subscript to print the middle value in a list of three temperatures.

4.1.3 Counting values

The `count` keyword is useful whenever a program needs to know how many values a collection contains. It is an expression rather than an ordinary library function, so it can be used directly in calculations and loop conditions.

```
list tasks = ["backup files", "send report", "check messages"]
number task_count = count(tasks)
print("There are " & task_count & " tasks")

string message = "YASS"
number character_count = count(message)
print("The message has " & character_count & " characters")
```

Although lists are a common use, `count` also works with strings: it reports the number of characters in the string. This makes it useful for checking input length, measuring labels, and deciding whether a message is empty or large enough to process. Remember that `count` is a language keyword, so a variable should not be named `count`; use a name such as `item_count` instead.

4.1.4 Maps

A map associates keys with values. It is the natural choice when the program needs to answer a question such as "what is this person's email address?" or "what price belongs to this product?"

```
map product = [
    "name" => "Notebook",
    "price" => 4.50,
    "in_stock" => true
]
print(product)
```

The keys give meaning to the values, so a map is usually clearer than a list of fields whose positions must be memorised. Maps are useful for records, configuration data, JSON-like data, and look-up tables. The order of map entries should not be treated as part of the meaning unless an ordered map is used.

4.1.5 Typed collections

Collection types can include angle brackets that describe the types of the values they contain. A typed list states the type of every element, while a typed map states the type of its keys followed by the type of its values.

```
list<number> scores = [10, 20, 30]
map<number, string> students = [
    1 => "Ada",
    2 => "Grace"
]
```

The extttlist<number> declaration says that extttscores contains numbers. The extttmap<number, string> declaration says that extttstudents uses numbers as keys and strings as values. These declarations describe the collection's intended shape and allow ZPE to perform additional type checking.

4.1.6 Map subscripting

A map is accessed with the same square-bracket notation, but the subscript is a key rather than a numeric position.

```
map prices = [
    "notebook" => 4.50,
    "pen" => 1.20
]
number notebook_price = prices["notebook"]
print(notebook_price)
```

The key identifies the entry, which is why a map lookup remains readable even when entries are reordered. A program should consider whether the key exists before relying on the result.

4.1.7 Testing whether an index or key exists

The `is_set` keyword checks whether a subscripted value is present. It is useful before reading a list position or map key when the program cannot assume that the requested item exists.

```
list books = ["YASS Guide", "Programming Notes"]
map prices = ["pen" => 1.20]

if (is_set(books[1]))
    print(books[1])
end if

if (is_set(prices["pen"]))
    print(prices["pen"])
end if
```

For a list, the expression tests whether the index is available. For a map, it tests whether the key is present. This is different from testing whether the stored value is true, non-zero, or non-empty: an existing entry may legitimately contain `false`, 0, or an empty string. Use `is_set` when the question is whether the path exists at all.

Exercises

1. Use `is_set` before reading the last possible index of a list.
2. Check whether a map contains a requested username before printing its value.
3. Add a missing-key message to the map example.

4.1.8 Iterating through a map

Maps are often processed with a loop that provides both the key and its associated value. The following form makes the two parts of each map entry explicit:

```
mp = [
    "jamie1" => "Jamie Balfour",
    "john1" => "John Balfour"
]

for each (mp as key => value)
    print(key & ": " & value)
end for
```

The key identifies the entry and the value contains the data stored under that key. The same idea can be expressed with the unbound naming style used elsewhere in this book; the important point is that a map loop exposes both pieces of each entry. This is more informative than looping over values alone when the key is needed for a label, report, or lookup result.

Exercises

1. Add a third user to the map and print all three key-value pairs.
2. Change the loop so that it prints only entries whose value contains the surname "Balfour".

4.1.9 Arrays

An array is an indexed collection intended for values of a consistent kind and a more fixed structure. Arrays are useful when a program needs efficient indexed storage, such as a row of measurements, a set of positions, or a block of numeric data. Unlike a general list, an array communicates that the collection has a stronger shape and element-type expectation.

Arrays are particularly useful in larger programs and performance-sensitive code. They should not be introduced merely because the word "array" is familiar: if the program needs a flexible sequence, use a list; if it needs named fields, use a map. In YASS, an array is a distinct fixed-size collection rather than simply another name for a list. Its declaration and element rules should therefore be checked against the version of YASS being used; do not silently replace an array with a list when the fixed-size behaviour is important.

4.1.10 Ordered maps

An ordered map combines named lookup with a guaranteed insertion order. This is useful when the keys identify values but the order must also be preserved, for example when generating a menu, writing table headings, or serialising configuration in a predictable order. An ordered map can be written directly with braces:

```
values = {"first" => 10, "second" => 20}
```

This preserves the order in which the entries are written.

An ordered map should be chosen deliberately. If the program only needs access by key, an ordinary map expresses that requirement more directly. If it needs both key-based

lookup and stable presentation order, an ordered map records both requirements in the data structure itself.

4.1.11 Choosing a collection

The four collection types can be summarised as follows:

- use a `list` for a flexible ordered sequence;
- use an `array` for a more strongly shaped indexed collection;
- use a `map` for key-to-value lookup where entry order is not important;
- use an ordered map when keys matter and insertion order must be preserved.

The choice affects how the rest of the program is written. A list is normally traversed by value or position, while a map is normally accessed by key. An ordered map adds a presentation guarantee, and an array adds stronger expectations about indexed data. These distinctions become increasingly important when collections are passed to functions or exchanged with files and other programs.

4.1.12 Try it: loops and collections

1. Use `for each` to print every item in a shopping list.
2. Use a loop to calculate the total of a list of prices.

4.2 Choosing a loop

The loop forms overlap, but they communicate different intentions:

- use `for` when a compact counter-controlled loop is appropriate;
- use `for-to` when the program walks through a numeric range;
- use `for each` when the program visits values in a collection;
- use `while` when the condition is checked before each possible iteration;
- use `repeat` when repetition itself, or a repeat-until condition, is the central idea.

The best loop is usually the one that makes its stopping rule obvious to the reader. A shorter loop is not automatically a clearer loop.

4.3 Collection review exercises

Build one small library catalogue, changing its representation as the requirements become clearer.

1. Store several book titles in a list and print them.
2. Add a map containing one book's title, author, and availability.
3. Store several book maps in a list and search for a title.
4. Add an array for a fixed set of shelf numbers.
5. Use an ordered map to produce a catalogue in a predictable order.
6. Build a final catalogue report that uses all the required collections.

Chapter 5

Functions

Functions let a program give a name to a useful operation. Instead of repeating the same instructions, a function packages them into a reusable unit. Functions can accept parameters, calculate results, and hide details that do not need to be repeated throughout the rest of the program.

5.1 Returning a value

The `return` statement ends the current function and sends a value back to the code that called it. The returned expression can be a literal value, a variable, or a larger calculation. Once a `return` statement has run, the remaining statements in that function are not executed.

```
function get_message() : string
    return "Hello from a function"
end function

string message = get_message()
print(message)
```

A function can also return the result of a calculation. Giving the result a name at the call site makes it possible to use that value later, pass it to another function, or store it in a collection. A function whose purpose is only to perform an action can return without a useful value; functions that calculate something should document an appropriate return type.

5.2 Defining a function

The following function accepts two numbers and returns their total. The parameter types and return type make the contract visible to the reader.

```
function add(number left, number right) : number
    return left + right
end function

number total = add(12, 8)
print(total)
```

A parameter is a value supplied by the caller. A return value is the result sent back by the function. A function should have one clear responsibility; small functions are easier to test, name, and reuse.

5.3 Parameters and return values

Parameters allow one function to work with many different values. The function below converts a number of minutes into seconds.

```
function to_seconds(number minutes) : number
    return minutes * 60
end function

number duration = to_seconds(5)
print(duration)
```

The name of a parameter belongs to the function. It does not have to be the same as the name used by the caller. This separation allows a function to describe its own calculation without depending on the names chosen elsewhere.

5.4 Union return types

Some functions deliberately return one of several kinds of value. YASS supports union return types, written by separating the permitted types with a vertical bar. This is useful when a function can return a calculated value on success or the original text when the input is not numeric.

```
function parse_value(string text) : number | string
    if (string_is_numeric(text))
        return 10
    end if
    return text
end function

print(parse_value("10"))
print(parse_value("unknown"))
```

The caller must understand both permitted results. A union return type is not a substitute for a clear interface: if callers would have to guess what a result means, return a record or another structured value instead. Union types are especially useful at boundaries where a function naturally represents either a value or a textual explanation.

5.5 Named and default parameters

Arguments are normally supplied in the same order as the parameters in the function declaration. Named arguments make the call more explicit and allow the caller to provide values by parameter name. This is particularly helpful when a function has several parameters of the same type.

```
function describe_person(string name, number age) : string
    return name & " is " & age & " years old"
end function

string description = describe_person(age = 36, name = "Ada")
print(description)
```

A parameter can also have a default value. The default is used when the caller does not supply that argument.

```
function greeting(string name = "Guest") : string
    return "Hello, " & name
end function

print(greeting())
print(greeting("Ada"))
```

Default values are useful when one choice is suitable for most calls but the caller may still override it. Required parameters should come before parameters with defaults, and a default should make sense as a complete, useful value rather than hiding a missing piece of essential information.

5.5.1 Try it: parameter choices

1. Add a default subject to a function that creates a welcome message.
2. Call a two-parameter function once positionally and once with named arguments.

5.6 Functions that produce text

Not every function performs a numerical calculation. A function can construct a message or format a result.

```
function greeting(string name) : string
    return "Hello, " & name
end function

string message = greeting("Ada")
print(message)
```

Returning a value is often more flexible than printing inside the function. The caller can print the result, store it, combine it with another string, or pass it to another function.

5.7 The test function

YASS supports test functions for checking that a program's functions behave as expected. A test uses assertions to state an expectation. The following test checks the result of the earlier function.

```
test function()
    assert(add(2, 3) == 5)
end function
```

Tests turn an assumption into an executable check. A useful test includes ordinary cases, boundary cases, and cases that previously caused a bug. Tests should be small enough that a failure gives a clear clue about what needs attention.

5.8 Functions and collections

Functions can accept and return collections. This keeps the operation separate from the data on which it operates.

```
function count_items(list items) : number
    number item_count = 0
    for each (item in items)
        item_count = item_count + 1
    end for
    return item_count
end function

list tasks = ["backup", "report", "review"]
print(count_items(tasks))
```

The function does not need to know where the list came from. That makes it useful for lists read from a file, entered by a user, or returned by another function.

5.9 Functions as values

YASS functions are values, so a function can be stored in a variable and passed to another function. This is the foundation of functional programming and makes it possible to describe an operation separately from the code that applies it.

```
fn operation = function(number value) : number
    return value * 2
end function

print(operation(6))
```

The `fn` type describes a value that contains a function. YASS also accepts `function` in this type position, so the following declaration could equivalently begin with `function operation =`. The shorter `fn` spelling is convenient when a function is being treated as a value, while `function` can make the meaning more obvious to readers. This is different

from declaring a named function: the operation is created as a value and can be replaced with another compatible operation.

5.10 Lambda functions

A lambda function is an anonymous function, usually written close to the place where it is used. Lambdas are useful for short transformations and callbacks where giving the operation a permanent top-level name would add unnecessary ceremony.

```
fn triple = function(number value) : number
    return value * 3
end function

print(triple(4))
```

Use a named function when the operation is important enough to be reused or documented independently. Use a lambda when the operation is short, local, and clearer when kept beside the code that consumes it.

5.11 Arrow functions

YASS also provides a compact arrow form. The arrow separates the parameters on the left from the expression or action on the right.

```
fn double = (number value) => value * 2
print(double(9))
```

An arrow function with parameters is useful when the operation can be expressed as one short expression. For a no-argument action, a normal function declaration is clearer and is the form used elsewhere in this book. Longer behaviour should normally use a full function block so that the structure remains easy to read.

5.12 Choosing a function design

Good function design is mostly about clear boundaries. A function should have a meaningful name, a small set of inputs, a predictable result, and a responsibility that can be explained in one sentence. Avoid functions that silently change unrelated state or combine input, calculation, formatting, and file output without a reason.

5.13 Chapter-wide review exercises

Build one collection of reusable utilities, adding a new function to the same program at each step.

1. Write a typed function that creates a formatted book label.

2. Add a function that counts books in a list.
3. Add a test function for both ordinary and empty lists.
4. Store a short transformation as a lambda.
5. Use an arrow function for one simple collection operation.
6. Build a tested catalogue utility from the functions you have written.

Chapter 6

Records

Records provide a typed way to group related fields. They are useful when a program needs data with a known shape but does not need the full behaviour of a structure and object. A record describes data; a structure can additionally contain methods and other object behaviour.

6.1 Defining a record

Every field in a YASS record must have a type. Fields may be given default values, or they may be declared without values and supplied when the record is created.

```
record pupil
    string forename = "Jamie",
    string surname = "Balfour"
end record
```

The comma after each field separates it from the next field in the record declaration. Unlike a map, the record documents its fields and their types in the declaration itself.

6.2 Creating a record

Record structures are instantiated with `new`. The record name can then be used as the type of a variable containing a record instance.

```
record book
    string title,
    string author,
    boolean available
end record

library_book = new book("Programming Notes", "Ada", true)
print(library_book.title)
```

The constructor values follow the order of the declared fields. A record with no default values must be given suitable values before its fields are used.

6.3 Reading and changing fields

Record fields are accessed with the dot operator. This makes record code concise while keeping the fields typed and named.

```
library_book.available = false
boolean can_borrow = library_book.available
print(can_borrow)
```

Records are a good fit for rows from a data file, measurements, addresses, catalogue entries, and other pieces of information that have a stable set of fields.

6.4 Records and maps

A map is flexible: new keys can represent different shapes of data. A record is more deliberate: its fields are declared in advance and each field has a type. Use a map when the keys are naturally dynamic; use a record when the shape is part of the program's design.

```
record measurement
    string sensor,
    number value,
    string unit
end record

measurement reading = new measurement("outside", 21.5, "C")
print(reading.sensor & ": " & reading.value & reading.unit)
```

6.5 Records and structures

Records and structures both group related values, but they communicate different intentions. A record is primarily typed data. A structure is a template for objects that may contain methods, constructors, and more involved behaviour. Starting with a record is often clearer when no behaviour is needed yet.

6.6 Exercises

Continue the library project from the previous chapter. Replace suitable map-like data with records where the fields have a stable, known shape.

1. Define a typed `BookRecord` with title, author, and availability fields.
2. Create a book record with the `new` keyword and print its title.
3. Add a `StudentRecord` with a name and library-card number.
4. Create a list of book records and print each title with a loop.

5. Decide which library values should remain maps and which should become records.
6. Extend the library report so that it reads typed fields from both record types.

Chapter 7

Structures and Objects

Structures allow a programmer to define a reusable shape containing data and functions. An object is an instance created from that structure. Together they provide a way to keep related state and behaviour together.

7.1 Defining a structure

The following structure describes a person. Its fields provide the state that every person object will have.

```
structure Person
    string name = ""
    number age = 0

    public function description() : string
        return name & " is " & age & " years old"
    end function
end structure
```

A structure is a template rather than one particular person. It describes the fields and functions available to objects created from it.

7.2 Creating objects

Use `new` to create an object from a structure. Each object has its own values for the fields defined by the structure.

```
Person ada = new Person()
Person grace = new Person()

ada->name = "Ada"
ada->age = 36
grace->name = "Grace"
grace->age = 28

print(ada->description())
print(grace->description())
```

The accessor arrow, `->`, selects a field or function belonging to an object. It is different from `=>`, which associates a key with a value in a map.

7.3 Creating simple objects directly

Not every object needs a named structure first. YASS can create a simple object directly from a set of named properties. This is useful for small pieces of related data whose shape is local to one part of a program.

Property names may be written as quoted strings:

```
object x = {"name" : "Jamie", "age" : 35}
print(x)
```

When a property name is a valid identifier, the quotation marks may be omitted. The result has the same two properties:

```
object x = {name : "Jamie", age : 35}
print(x)
```

These are object literals, not map literals. A map associates keys and values using the map syntax with `=>`; an object literal describes an object with named properties using `:`. The distinction matters when code later accesses properties, passes the value to an object-aware function, or relies on object behaviour. For a reusable design with methods, constructors, inheritance, or access control, define a `structure` and create an instance with `new` instead.

The syntax is deliberately familiar to anyone who has worked with JSON. A simple ZPE object can often be read as a JSON object: property names are followed by colons, values may be strings, numbers, booleans, lists, or nested objects, and commas separate properties. This makes it natural to move data between a YASS program and a web service or configuration file. The similarity is not a promise that every YASS object is valid JSON, however. Objects with methods, special runtime behaviour, or values that JSON cannot represent may need to be converted before they are serialised.

7.4 Built-in objects

ZPE also provides objects that are ready to use from YASS. Examples include the `Date` object for working with dates and times, the `Exception` object for representing errors,

and the `Calculator` object for calculator-style operations. Other built-in objects support areas such as databases, HTML documents, colours, points, processes, and streams.

The installed runtime can list the objects it knows about. Run the following command in a terminal:

```
zpe -h objects
```

This is preferable to relying on an old list in a book: the command reports the objects available in the particular ZPE installation. The documentation and the object descriptions can then be used together to learn the methods and properties exposed by an object.

7.5 Constructors

A constructor gives an object its initial values at the moment it is created. In YASS, a structure can provide a special `_construct` function.

```
structure Book
    string title = ""
    string author = ""

    function _construct(string new_title, string new_author)
        title = new_title
        author = new_author
    end function
end structure

Book book = new Book("Programming Notes", "Ada")
print(book->title)
```

Constructors are useful because they prevent an object from being created in an incomplete state. The caller supplies the essential data once, and the structure establishes the initial object.

7.6 Methods and object behaviour

Functions inside a structure are often called methods because they describe behaviour belonging to an object. A method can read the object's fields, update them, or return a calculated result.

```
structure Counter
    number value = 0

    public function increment()
        value = value + 1
    end function

    public function current() : number
        return value
    end function
end structure
```

The structure keeps the counter's state and the operations that change it together. This is the central idea behind object-oriented programming.

7.7 Encapsulation

Encapsulation means keeping an object's state and the operations that control it together, while exposing only the interface that other code needs. A private balance, for example, should be changed through a method that can enforce the account's rules rather than being changed freely by every caller. Public methods form the usable interface; protected members support derived structures; private members hide implementation details.

```
structure Account
    private number balance = 0

    public function deposit(number amount)
        if (amount > 0)
            balance = balance + amount
        end if
    end function

    public function get_balance() : number
        return balance
    end function
end structure
```

The caller can ask the account to deposit money and can read the balance through its public methods, but cannot directly replace the private field. Encapsulation is therefore about protecting invariants and making the permitted operations clear, not about hiding every detail for its own sake.

7.8 Inheritance

Inheritance allows one structure to build on another. The child structure receives the fields and methods of its parent and can add new behaviour or provide a more specialised implementation. Use inheritance when the child really is a specialised form of the parent; otherwise, composition is often easier to understand.

```
structure Person
    string name = ""

    public function description() : string
        return this->name
    end function
end structure

structure Student inherits Person
    string course = ""

    public function description() : string
        return super->description() & " studies " & this->course
    end function
end structure

Student ada = new Student()
ada->name = "Ada"
ada->course = "computing"
print(ada->description())
```

The keyword `inherits` declares the relationship. Inside the child structure, `super` refers to the parent implementation, so `super->description()` calls the version defined by `Person`. This is useful when the child wants to extend a parent's behaviour rather than replace it completely. YASS also supports `extends` as an alternative spelling for inheritance.

7.9 Scope and access control

Scope determines where a variable or method can be seen and used. Within a structure, YASS provides three access levels: `public`, `protected`, and `private`. These keywords make the intended boundary of an object explicit.

A public field or method is available to code that can access the object. A protected member is available to the object and related descendant code, while a private member is restricted to the structure's own methods. If no access keyword is written, the member uses YASS's default protected or friendly behaviour. The following example shows the declarations together:

```
structure Account
    public string owner = ""
    protected number balance = 0
    private string security_code = ""

    public function name() : string
        return owner
    end function

    private function valid_code(string supplied) : boolean
        return supplied == security_code
    end function
end structure
```

The access level is part of the design, not merely decoration. Public members form the interface that other code is expected to use. Protected members support cooperation between a structure and related code without exposing every detail to the whole program. Private members protect implementation details, such as a security code, that callers should never change directly. Keeping a small public interface usually makes an object easier to use and safer to modify.

Scope also applies to ordinary variables. A variable declared inside a function normally belongs to that function and its descendants. A variable declared at global level can be visible more broadly, while a private declaration can keep a local value from being confused with a value in an enclosing function. These rules become especially important when methods update object state or when nested functions share values.

7.10 Structures and collections

Objects can contain lists and maps, which makes them useful for modelling larger real-world entities.

```
structure Student
    string name = ""
    list subjects = []

    function subject_count() : number
        number item_count = 0
        for each (subject in subjects)
            item_count = item_count + 1
        end for
        return item_count
    end function
end structure
```

A structure can therefore represent both an individual item and the operations that make sense for that item.

7.11 Structures as types

Once a structure has been declared, its name can be used as a type for variables that should contain objects created from that structure. This makes the relationship between a variable and its expected object explicit.

The foundations introduced herestructures, objects, encapsulation, inheritance, access control, and methodsare the vocabulary needed for the later chapters on scope, records, exceptions, and reusable components.

7.12 Chapter-wide review exercises

1. Build a complete program using the chapter's main ideas.
2. Explain the most important design choice in your solution.

3. Test the program with at least three different inputs.
4. Improve the program's names and structure.
5. Describe one edge case and handle it.
6. Extend the program with one useful feature.

Chapter 8

Interfaces

An interface describes a contract that a structure must satisfy. It names the methods that an implementing structure must provide without deciding how those methods work. Interfaces are useful when different structures should be usable through the same set of operations.

8.1 Declaring an interface

An interface can declare a method without giving it a body. A structure that implements the interface must provide a compatible method with the same name, parameters, and return type.

```
interface Readable
    public function read() : string
end interface

structure Book implements Readable
    public string content = "A short book"

    public function read() : string
        return this->content
    end function
end structure

Book book = new Book()
print(book->read())
```

If a structure does not provide all of the required methods, ZPE reports the incomplete implementation during compilation. This lets a program use the interface as a reliable promise: any value presented as `Readable` will provide the `read` operation.

8.2 Multiple interfaces and checking

A structure may implement more than one interface. List the interface names after `implements`, separated by commas, and provide every required method. YASS can also

use the `implements` expression to check whether a value satisfies an interface before calling its methods.

```
if (book implements Readable)
    print("This book can be read")
end if
```

Interfaces describe capabilities rather than family relationships. Use inheritance when one structure is a specialised form of another; use an interface when unrelated structures should share a usable contract.

8.3 Interface exercises

Extend the library project with shared capabilities.

1. Define a `Borrowable` interface with a method for borrowing an item.
2. Make a book structure implement `Borrowable`.
3. Create a second structure that implements the same interface.
4. Add a function that accepts an interface-compatible object and calls its required method.
5. Check an object with the `implements` expression before using it.

Chapter 9

Exceptions

Programs do not always receive valid input or complete the operations they attempt. A file may be missing, a value may have the wrong type, or a calculation may not be possible. An exception is an object that represents this kind of exceptional situation. Exception handling lets a program respond deliberately instead of stopping without an explanation.

9.1 Trying an operation

The `try` construct marks code that may fail. A matching `catch` block describes what the program should do if an exception is raised.

```
try
    throw_error(new Exception("The calculation could not be completed"))
catch Exception problem
    print(problem->message())
end try
```

If the operation in the `try` block succeeds, the catch block is skipped. If an exception is raised, execution moves to the catch block. The exception variable, `problem` in this example, contains information about what went wrong.

Exercises

1. Put an operation that may fail inside a `try` block and print a friendly message from `catch`.
2. Add a successful operation before the failing operation and observe which statements run.

9.2 The Exception object

YASS represents errors as instances of the built-in `Exception` object. An exception can carry a message and other information about the failure. Code can inspect the caught object and display its message to help the user or developer understand the problem.

```
try
    throw_error(new Exception("The account could not be opened"))
catch Exception problem
    print(problem->message())
end try
```

The `throw_error` function raises an exception and transfers control to the nearest suitable catch block. Modern ZPE expects `throw_error` to receive an `Exception` object rather than an arbitrary string. This keeps failures structured and allows code to handle them consistently.

9.3 Creating useful failures

A function can validate its input and raise an exception when the input violates its contract. The caller can then decide whether to catch the problem, report it, or allow it to propagate to a higher-level part of the program.

```
function withdraw(number balance, number amount) : number
    if (amount < 0)
        throw_error(new Exception("Withdrawal cannot be negative"))
    end if

    if (amount > balance)
        throw_error(new Exception("Insufficient funds"))
    end if

    return balance - amount
end function

try
    number remaining = withdraw(100, 125)
    print(remaining)
catch Exception problem
    print(problem->message())
end try
```

The function does not print an error or decide how the application should recover. It reports the invalid situation by throwing an exception, leaving the caller responsible for choosing an appropriate response. This separation makes the function reusable in a command-line program, a graphical application, or a web service.

9.4 Choosing what to catch

Catch an exception when the program can do something useful with it: display a helpful message, ask for new input, choose a fallback, or record the failure. Do not catch every error merely to hide it. A swallowed exception can make a program appear to work while silently producing incomplete or incorrect results.

9.4.1 Try it: the library project

Continue the library program used throughout the book.

1. Write a function that throws an exception when a student tries to borrow an unavailable book.
2. Catch the exception and print a clear explanation to the student.
3. Add a second catch scenario for returning a book that is not currently borrowed.
4. Preserve the original exception message while adding a more helpful user-facing message.
5. Test both successful and unsuccessful borrowing operations.
6. Decide which failures should be handled by the immediate caller and which should be allowed to propagate.

9.5 Chapter-wide review exercises

Extend the library program into a small, reliable service that validates its operations.

1. Add an exception for an unknown student.
2. Add an exception for an unknown book.
3. Add an exception for attempting to borrow an unavailable book.
4. Catch each failure and report a different useful message.
5. Add a function that validates a book record before it is added to the catalogue.
6. Test the complete program with successful operations, invalid input, and repeated borrowing.

Chapter 10

Enumerations

An enumeration gives a program a small, named set of related values. Enums are useful when a value should be one of a known collection of states, rather than an arbitrary number or string. Each member is assigned a value and can be accessed through the enum name and the static-member operator `::`.

10.1 Declaring an enum

Use `enum` or `enumerator` to declare an enumeration. The members may use supported values such as numbers, strings, or booleans.

```
enum Status
    OK = 1
    FAILED = 2
    UNKNOWN = 3
end enum

number state = Status::OK

if (state == Status::OK)
    print("The operation succeeded")
end if
```

The named members make the program easier to read than comparisons with unexplained numeric literals. An enum member is also a public static value, so it can be passed to functions and stored in other values like any compatible value.

10.2 Nested enums

An enum may be declared inside a structure. Its name is then qualified through the containing structures.

```
structure UI
    public structure Button
        static enum Action
            click = "click"
            mouseover = "mouseover"
        end enum
    end structure
end structure

string action = UI::Button::Action.click
print(action)
```

Use an enum when the allowed values are known and stable. Use a map or object when the program needs to associate arbitrary keys with changing data.

10.3 Enumeration exercises

Extend the library project with named states.

1. Define an enum for the availability of a book.
2. Store one enum member in a variable and print it.
3. Use an `if` statement to respond to one enum value.
4. Use `match` to produce a message for each availability state.
5. Create a nested enum for the kind of library item being catalogued.

Chapter 11

Components and Templating

In ZPE, a component is a special type that produces a string when it is invoked. Components are designed for efficient reusable output and are prepared by the component system before execution. They are especially useful for generating HTML: the markup stays in a reusable component while values are supplied when the component is called.

11.1 Component declaration and use

A component declaration begins with `component` followed by its name and ends with `end component`. The name is simply the name of the component. In this example, `product` is the component name and `id`, `title`, and `description` are input values. Each assignment after the declaration invokes the component and stores the resulting string.

```
component product
<div class="product" id="{{id}}">
  <div class="title">{{title}}</div>
  <div class="description">{{description}}</div>
</div>
end component

aftershave = product([
  "id" => "aftershave_product",
  "title" => "Aftershave",
  "description" => "A fantastic smelling aftershave"
])

butter = product([
  "id" => "butter_product",
  "title" => "Butter",
  "description" => "A tub of regular butter"
])
```

The double braces mark values that are inserted into the component's markup. Each map supplies the values for one product, so each invocation produces a separate HTML string. The same markup is reused for both products, while their data remains separate from the component declaration.

The same component can therefore produce many HTML strings without duplicating its markup. Keep the names in each input map consistent with the expressions used by the component.

11.2 Backtick templates

For smaller pieces of generated text, ZPE also provides backtick templating. A backtick-delimited template can contain ordinary text together with template expressions. This is convenient when a complete component page would be unnecessary.

```
string title = "My website"
string heading = `<h1>{{title}}</h1>`
print(heading)
```

Backtick templates are useful for short fragments, while components are the main tool for reusable markup. Templates are not a substitute for escaping or validating data: values supplied by users must still be handled safely before insertion.

11.3 Exercises

1. Create a component that renders a person's name and email address as HTML.
2. Create a component that renders a book title and author.
3. Use the same component with three different input maps.
4. Add a backtick template for a small HTML fragment.
5. Explain which values should be validated or escaped before insertion.

Chapter 12

Scope

Scope describes where a variable or function can be seen and used. We have already met scope in earlier chapters: a variable declared inside a function belongs naturally to that function, and a field declared inside a structure belongs to its objects. This chapter brings those ideas together and explains how ZPE manages global, local, function, object, and block scope.

12.1 Global and local variables

Code outside a named function runs in the global function. A variable declared there has global scope and can be used by other code that can see the global environment. A variable declared inside a function is local to that function and its descendants.

```
string application_name = "Library Desk"

function show_application()
    string message = application_name & " is ready"
    print(message)
end function

show_application()
```

Here, `application_name` is global, while `message` is local to `show_application`. The function can read the global value, but code outside the function cannot use `message` after the function has finished. Global variables can be convenient for genuine application-wide configuration, but local variables are usually easier to understand because their lifetime and purpose are visible near their declaration.

Exercises

1. Create a global library name and print it from two different functions.
2. Create a local message in one function and explain why another function cannot use it directly.

12.2 Function scope

Every function creates a scope for its parameters and local variables. A parameter is available inside the function that declares it. A nested function may be able to read values from its containing function, but deliberately creating local names avoids accidental changes to an enclosing value.

```
function report_total(number first, number second) : number
    number total = first + second
    return total
end function

number result = report_total(12, 8)
print(result)
```

The names `first`, `second`, and `total` belong to the function's scope. They are implementation details of `report_total`; the caller only needs to know the function's parameters and return value. This separation is one reason functions make larger programs easier to maintain.

12.3 Object scope and the `this` keyword

An object has its own scope for fields and methods. When a method accesses a field without an object prefix, YASS can resolve it as a member of the current object. The `this` keyword makes that relationship explicit: it refers to the object whose method is currently running.

```
structure Counter
    number value = 0

    function increment()
        this->value = this->value + 1
    end function

    function current() : number
        return this->value
    end function
end structure

Counter counter = new Counter()
counter->increment()
print(counter->current())
```

The accessor arrow selects a member from an object. Thus `this->value` means the `value` field belonging to the current object, while `counter->value` would refer to the field belonging to the object stored in `counter`. The keyword is especially useful when a structure inherits fields or when a method needs to make the owner of a value unmistakable.

Exercises

1. Add a `reset` method that sets the counter's value to zero through `this`.
2. Create two counters and show that their fields are kept in separate object scopes.

12.4 Scope blocks

A scope block creates a smaller, temporary environment inside a function. In keyword-ending syntax it begins with `block` and ends with `end block`. Scope blocks are useful when a short calculation needs temporary names that should not become part of the surrounding function's working scope.

```
function calculate_total(number price, number quantity) : number
    block
        number subtotal = price * quantity
        print("Subtotal: " & subtotal)
    end block

    return price * quantity
end function
```

A scope block can read values from its surrounding scope, such as `price` and `quantity`. Its temporary names should not be used as though they were ordinary names in the surrounding function. The same construct can be written with the brace form when using YASS's C-style syntax, but the keyword-ending form makes the boundary explicit for readers.

12.5 Public, protected, and private scope

Scope also controls access to members of structures. A public member is available to code using the object. A protected member is available to the object and related descendant code. A private member is restricted to the structure's own methods. These access levels were introduced with structures and objects earlier, but they are best understood as scope rules: each one defines who can see or change a member.

12.6 Choosing a scope

Prefer the narrowest scope that expresses the design. Keep calculations local to their functions, keep object state inside the object, use `this` when member ownership needs to be clear, and reserve global variables for values that genuinely belong to the whole program. Narrow scope reduces accidental changes and makes it easier to reason about a program one part at a time.

12.6.1 Scope review exercises

Continue the library project and make its ownership boundaries explicit.

1. Identify which values in the library program should be global configuration.
2. Move a temporary calculation into a function-local variable.
3. Add a method that updates a book through `this`.
4. Use a scope block for a temporary receipt subtotal.
5. Mark one object member public, one protected, and one private, then describe who can access each.
6. Test the program after attempting to access a private member from outside its structure.

Chapter 13

GoTo and Labels

Labels and `goto` allow a program to jump directly from one point to another. A label gives a position in the program a name, and `goto` transfers execution to that label.

13.1 A labelled loop

The following example repeats a small section until the counter reaches ten. The label identifies the beginning of the repeated section, while the `goto` returns execution to it.

```
number i = 0

label openloop
print(i)
i = i + 1

if (i < 10)
    goto openloop
end if

print("Done")
```

The label does not perform an action by itself. It marks a destination. The `goto` statement names that destination, and the program continues from the labelled line. A label should be unique within the part of the program where it can be reached.

Exercises

1. Change the example so that it prints the numbers from 1 to 5.
2. Add a second label and use it to skip a message under a particular condition.

13.2 Why structured loops are usually better

Although labels and jumps are useful tools, a `for`, `while`, or `repeat` loop usually communicates repetition more clearly. A structured loop keeps its starting point, stopping

condition, and body together. With `goto`, the reader must search for the destination and check every possible path back to it.

```
number i = 0

while (i < 10)
    print(i)
    i = i + 1
end while
```

This has the same general behaviour as the labelled example, but its control flow is visible in one block. Prefer a structured loop when the task is naturally a repetition. Reserve `goto` for situations where a named jump genuinely makes the program clearer, such as carefully controlled state transitions or low-level compatibility code.

13.3 Labels and decisions

A jump can be placed inside a decision, allowing the program to leave one path and continue at a named destination. This can be useful, but it can also create difficult-to-follow paths when many labels and jumps are mixed together.

```
string command = "quit"

if (command == "quit")
    goto finished
end if

print("The command was not quit")

label finished
print("Finished")
```

In this example the jump skips the first message. A function that returns, or an ordinary `if-else` structure, may be clearer in a larger program. The important skill is not simply knowing how to jump, but being able to choose the control-flow form that makes the program's intention easiest to verify.

13.3.1 Try it: the library project

Continue the library project and compare structured and labelled control flow.

1. Create a labelled search through a short list of book titles.
2. Stop the search by jumping to a label when the title is found.
3. Rewrite the same program using a structured loop and a boolean flag.
4. Compare which version makes the stopping condition easier to understand.
5. Add a second label for the case where no title is found.
6. Explain why a function might be preferable to either version in a larger program.

13.4 Chapter-wide review exercises

Build a small command processor that uses labels only where they add a clear benefit.

1. Print a menu with three commands.
2. Use a label to return to the menu after an invalid command.
3. Add a structured loop version of the same menu.
4. Use a label to reach a common finishing message.
5. Test the program with valid, invalid, and repeated commands.
6. Refactor one jump into a function or structured conditional and explain the improvement.

Chapter 14

Modules

As programs grow, related functions, constants, and structures need a clear home. A module provides a named boundary for reusable code. It helps prevent unrelated names from colliding and gives a program a more deliberate organisation than one very large source file.

14.1 Why use a module?

A module can group the operations belonging to one area of a program, such as catalogue management, input handling, or mathematics. Module members can be accessed through the module rather than being placed into the global namespace without explanation.

The following is a module-file fragment. It is meant to be compiled as part of a module-aware project, rather than pasted into an ordinary one-file script.

```
module Catalogue
    public function format_label(string title, string author) : string
        return title & " by " & author
    end function
end module
```

The exact members exposed by a module form part of its public interface. Keep that interface small and meaningful so that other programs do not need to know how the module performs its work internally. The function can then be called through the module's qualified name:

```
string book_label = Catalogue::format_label("Programming Notes", "Ada")
print(book_label)
```

The name `format_label` is deliberately used instead of `label`: `label` is reserved by YASS and should not be used as a function or variable name.

14.2 Built-in modules

ZPE includes built-in modules for common areas of work. Examples include `Math` for mathematical operations, `System` for system-level services, `Input` for input-related operations, and `CLI` for command-line behaviour. The `ZPE` and `Utilities` modules provide access to facilities belonging to the environment itself.

These modules are part of the ZPE ecosystem rather than code that must be copied into every YASS file. Their exact functions and members should be checked in the version of the ZPE documentation being used.

The installed runtime can list the available modules too:

```
zpe -h modules
```

This command is useful when a new ZPE release adds a module or moves a facility from an older global function into a module. It also provides a quick way to check what is available before designing a program around a particular module.

```
number distance = Math::absolute_value(-12)
print(distance)
```

The double-colon notation identifies a static member of a module. It makes the owner of the operation visible at the call site and reduces ambiguity when two modules provide similarly named operations.

Exercises

1. Find one useful operation in the `Math` module and use it in a typed program.
2. Identify a task in the library project that would make sense as a separate module.

14.3 Namespaces

A namespace adds another level of organisation inside a structure or module. It is particularly valuable when two libraries use the same structure name. The following example places `HTMLBuilder` in the `jsmith/packages` namespace and then constructs it using its qualified name.

```
structure HTMLBuilder
    namespace jsmith/packages

    string title = ""
end structure

jsmith/packages/HTMLBuilder builder = new jsmith/packages/HTMLBuilder()
```

The qualified name identifies the structure belonging to that namespace. Without the namespace, it could collide with another library's `HTMLBuilder`. Namespaces are therefore about identity and organisation, while modules are about grouping reusable members.

Exercises

1. Give the library project's structures a namespace based on the project name.
2. Write down two possible collisions that namespaces could prevent.

Chapter 15

Directives and Annotations

Directives and annotations provide information to the compiler, runtime, editor, or documentation tools. They are written with the `@` syntax and are not ordinary function calls.

15.1 Documenting a function

The `@doc`, `@author`, and `@date` annotations attach information to a function. This information can be retained when a compiled program is exported as a library.

```
@doc "format_label : Creates a printable label for a book"
@author "Jamie Balfour"
@date "October 2026"
function format_label(string title) : string
    return "Book: " & title
end function
```

Documentation annotations are useful because the explanation travels with the code. A library user can inspect the compiled documentation rather than having to search through the library's implementation.

15.2 Compiler directives

The `@directive` form controls compiler behaviour. For example, `prevent_auto_includes` can stop a compiled program from automatically including files at the start, while `requires_explicit_decla` asks the compiler to reject variables that have not been explicitly declared.

```
@directive requires_explicit_declaration : true

string title = "YASS"
print(title)
```

The `@requires return` annotation documents and enforces that a function must provide a return value. It is useful for functions whose callers rely on a result, especially when the function contains several branches.

```
@requires return
function make_label(string title) : string
    if (title == "")
        return "Untitled"
    end if
    return "Book: " & title
end function
```

Every path through a function marked this way should return a value of the declared type. The annotation makes that requirement visible to the reader and gives ZPE an opportunity to report an incomplete function during analysis or compilation.

15.3 Permission requirements

An annotation can also record the minimum permission level needed by a function. This is important for operations that access files, network resources, devices, or other protected capabilities. The following example documents that the function requires permission level 4:

```
@requires permission 4
function read_stream()
    return true
end function
```

The permission requirement is part of the function's contract. A caller or administrator can see that the operation needs elevated access before running it. For example, ZPE's `Stream::open` method requires permission level 4, which can be checked with `zpe -man Stream::open`. The exact level should always be confirmed for the installed ZPE version rather than guessed from the operation's name.

The `@compile` annotation can control whether a following function is included in a compiled application. This is useful for development helpers that are needed while working on source code but should not be emitted in the final application.

```
@compile false
function development_message()
    print("Development helper")
end function
```

Directives should be used deliberately. They change how tools interpret the program, so a directive deserves an explanation when its purpose is not obvious from the surrounding code.

15.3.1 Try it: documenting the library project

1. Add documentation annotations to one library function.
2. Turn on explicit declarations and fix any resulting undeclared names.
3. Add a development-only helper and exclude it from a compiled application.

Chapter 16

Importing Libraries and Including Code

Modules and namespaces organise code, but a program also needs a way to use code stored elsewhere. YASS provides importing for libraries and including for source files. These operations have different purposes and should not be treated as interchangeable.

16.1 Importing a library

The `import` command loads a library so that its public functionality can be used. An `as` clause can also create a local import alias when that is useful.

```
import stdLib
import catalogue as books
```

The precise members available depend on that library's documented interface. Importing is normally the right choice when code is intended to be a reusable library with a defined interface, documentation, and ownership boundary. Do not confuse this form with object construction: `new` is not part of the import declaration.

Importing `stdLib` makes the standard-library functions available to the YASS program. The same mechanism can also load plugins. A plugin may be implemented in native code, allowing it to provide functions and objects whose underlying work is performed outside YASS while still exposing a documented interface to the program.

16.2 Including source code

Including code places another source file into the current program's compilation context. This can be useful for a small group of shared declarations or a local helper file that is maintained alongside the main program.

```
include "library_helpers.yas"

string label = make_library_label("Programming Notes")
print(label)
```

An included file is part of the consuming program rather than an independent library boundary. Keep include files focused, avoid surprising side effects when they are read, and document the names they provide. For larger or reusable components, prefer a module or imported library.

This is a two-file example: create a file named textttlibrary_helpers.yas containing the helper declaration before running the consuming file. The include statement cannot succeed if that file does not exist.

16.3 Choosing between import and include

Use `import` when the code has a public interface and may be reused by several programs. Use `include` when a small project needs to share source declarations directly. The distinction helps a reader understand whether a dependency is a separately managed component or simply another part of the current program.

16.3.1 Try it: split the library project

1. Move the catalogue-label function into a helper source file.
2. Include that file from the main library program.
3. Identify which part of the project would be better designed as an imported module.
4. Add documentation annotations to the reusable function.
5. Give a helper structure a namespace to prevent a naming collision.
6. Explain which names should be public and which should remain internal.

16.4 Chapter-wide review exercises

Organise the library project as several cooperating source components.

1. Import `stdLib` and identify two functions it makes available.
2. Include a small helper file containing a formatting function.
3. Use the included function from the main catalogue program.
4. Import a library or built-in module needed by the program.
5. Investigate one native plugin and record which functions or objects it provides.
6. Compile and test the complete project, recording which names come from each imported or included component.

Chapter 17

ZEQL

ZEQL, the *ZPE Expression Query Language*, provides a compact way to filter and transform collections. It is useful when the task is naturally described as "take values from this source, keep the ones matching a condition, and select a result". ZEQL works with lists, maps, records, and other iterable values.

17.1 A first query

A query uses the keywords `from`, `in`, and `select`. The query variable is local to the query, so it does not add another name to the surrounding program.

```
list $numbers = [1, 2, 3, 4, 5]
$doubled = from $number in $numbers
    select $number * 2

print($doubled.to_list())
```

The expression following `select` is evaluated for each source item. It may return the original item, a calculated value, or a field taken from a record or object. Queries are lazy: declaring one does not process every value immediately. Enumeration begins when a result method such as `to_list()`, `length()`, or `first()` is used.

17.2 Filtering with where

The optional `where` clause keeps only the items for which its condition is true. It runs before the selection expression.

```
list $numbers = [1, 2, 3, 4, 5, 6]
$even_squares = from $number in $numbers
    where $number % 2 == 0
    select $number * $number

print($even_squares.to_list())
```

This produces the list `[4, 16, 36]`. Use `first()` when only the first matching value is

needed; there is no reason to build a complete list in that situation. Use `length()` when the number of matching values is the result of interest.

17.3 Predicate lambdas

A predicate lambda is a small unnamed function written beside the operation that uses it. In a collection pipeline, the parameter before the arrow represents the current item. The result of the lambda becomes the condition for `where`, or the transformed value for `select`.

```
fn is_even = (number value) => value % 2 == 0

$numbers = [1, 2, 3, 4, 5, 6]
$even_squares = from $number in $numbers
    where is_even($number)
    select $number * $number

print($even_squares.to_list())
```

The predicate function answers "should this item be kept?" The query's selection expression answers "what value should the result contain?" Predicate lambdas are particularly useful when a query rule is short and local. If a rule becomes long or needs to be reused, give it a name with a function instead.

17.4 Queries and ordinary loops

A loop is often clearer when each item requires several statements or produces side effects. A ZEQL query is often clearer when the operation is a pipeline of filtering and transformation. Both approaches are valuable, and the best choice is the one that makes the program's intention easiest to read.

17.4.1 Try it: query a collection

1. Create a list of prices and select only prices below a chosen limit.
2. Transform a list of names into a list containing their upper-case forms.
3. Use `first()` to find the first value satisfying a condition.

17.5 Exercises

1. Select the odd numbers from a list.
2. Produce the cubes of the numbers from 1 to 5.
3. Select the names containing the letter "a".
4. Query a list of records and select one field from each matching record.

5. Use `length()` to count how many values pass a filter.
6. Rewrite a short collection-processing loop as a query, then explain which version is easier to maintain.

Chapter 18

Unit Testing

Unit tests check small parts of a program automatically. They are especially useful for functions: each test can provide known input and verify that the result is the one the program promises to return. A failing test gives a precise place to start investigating when a later change breaks earlier behaviour.

18.1 Assertions

ZPE provides assertion statements for expressing expectations. The simplest assertions check whether an expression is true or false.

```
function double(number value) : number
    return value * 2
end function

assert(double(4) == 8)
```

An assertion succeeds silently when its condition is satisfied. If it fails, ZPE reports the failed test so that the program can be corrected. Keep each test focused: a test that checks one behaviour is easier to understand than one that combines several unrelated expectations.

18.2 Comparing expected and actual values

The assertion is useful when a function should return a particular value. The two expressions are compared directly, so the expected value can be written beside the expression being tested.

```
function greeting(string name) : string
    return "Hello, " & name
end function

assert(greeting("Ada") == "Hello, Ada")
```

Tests should include ordinary cases as well as boundary cases. For `double`, that might

mean testing zero, a negative number, and a decimal value. For `greeting`, it might mean testing an empty name and a name containing a space. These tests document the intended behaviour while also protecting it during refactoring.

For collection-valued results, the assertion can compare the complete result with the expected collection. This is useful for algorithms whose order matters, such as a Fibonacci sequence:

```
list a = fibonacci(9)
assert(a == [5, 8, 13, 21, 34])
```

The assertion passes only when the returned list has the expected values in the expected order. If the function returns an extra value, omits one, or places a value incorrectly, the test exposes the difference.

18.2.1 Try it: test a small function

1. Write a function that converts a temperature in Celsius to Fahrenheit.
2. Add an equality assertion for the freezing point.
3. Add a second assertion for a negative temperature.
4. Add an assertion for a second known conversion result.

18.3 A small test suite

A useful test suite names its tests clearly in comments or surrounding sections, keeps setup small, and tests the public behaviour rather than the implementation details. When a test fails, read the reported expression first; then decide whether the function or the test contains the mistake.

```
function is_adult(number age) : boolean
    return age >= 18
end function

assert(is_adult(18) == true)
assert(is_adult(35) == true)
assert(is_adult(17) == false)
```

Unit testing is not a replacement for trying a complete program. It is one layer of confidence: assertions check individual behaviours, while later integration tests can check that functions, collections, files, and imported libraries work together.

18.4 Exercises

1. Test a function that adds two numbers.
2. Test the same function with a negative number.
3. Test a function that returns the larger of two values.

4. Test both branches of a pass-or-fail function.
5. Test a string function with an empty string.
6. Write a short collection of assertions for one function from an earlier chapter.

Chapter 19

YASS and the Web

ZPE can run YASS as part of a web page. Its YWP format combines ordinary HTML with server-side YASS enclosed by `<zwp>` and `</zwp>` tags. The server sends the HTML to the browser, while the YASS inside the page runs first and can generate content or return a response.

19.1 A first YWP page

Save the following as `index.ywp`. The HTML is ordinary document content, while the YASS block is evaluated on the server.

```
<!doctype html>
<html>
  <head>
    <title>Simple YASS web page</title>
  </head>
  <body>
    <h1>Welcome to YASS</h1>
    <zwp>
      if (is_set(REQUEST->get('name')))
        print("Hello, " & REQUEST->get('name'))
      else
        print("Hello, visitor")
      end if
    </zwp>
  </body>
</html>
```

The page can be opened with a URL such as `/index.ywp?name=Jamie`. The browser supplies the query-string value, and the page reads it through the `REQUEST` object.

19.2 The REQUEST object

Every YWP page receives a `REQUEST` object. It represents information supplied with the current HTTP request. Query-string values are available through `REQUEST->get`, submit-

ted form data through `REQUEST->post`, and raw submitted data through `REQUEST->post_raw`. Headers, server information, and session values are exposed through the corresponding request methods.

```
<zwp>
  string query = REQUEST->get('q')

  if (is_set(query))
    print("Searching for: " & query)
  else
    print("Enter a search term.")
  end if
</zwp>
```

Treat request data as untrusted input. Validate it before using it in calculations, database queries, file names, or generated markup. A missing query value should be a normal case that the page handles deliberately, not an unexpected crash.

19.3 Forms and POST data

An HTML form normally sends its fields using POST. YASS reads those fields through the request object.

```
<form method="post">
  <label>Name <input name="name"></label>
  <button type="submit">Send</button>
</form>

<zwp>
  if (is_set(REQUEST->post('name')))
    print("Welcome, " & REQUEST->post('name'))
  end if
</zwp>
```

The page should still validate the submitted value. Checking whether a field exists is not the same as checking whether it is suitable for the operation the program is about to perform.

19.4 The RESPONSE object

The `RESPONSE` object controls the HTTP response sent to the browser. It can return JSON, set response headers, select a status code, redirect the browser, and store session values.

```
<zwp>
  map result = ["name" => "Jamie", "role" => "teacher"]
  return RESPONSE->json(result)
</zwp>
```

Other useful operations include `RESPONSE->set_header(...)`, `RESPONSE->set_http_response(...)`, `RESPONSE->redirect(...)`, and `RESPONSE->set_session_variable(...)` . Choose

the response deliberately: a browser page, a JSON API, and a redirect communicate different results to the client.

19.5 Reusable pages

Larger sites should not place every piece of markup in one page. A YWP page can render another page while preserving the active request, route values, and page context.

```
<zwp>
  render "/partials/navigation.ywp"
</zwp>
```

Use `render` when the included page needs the current web context. Ordinary `include` remains useful for isolated source inclusion, but it should not be assumed to receive the same request and route context.

19.6 Routers

Traditional YWP pages are selected directly by their file paths. A router provides an application-style alternative: the root page selects a router, and route functions handle matching requests. Routing is optional. The `@zpe router` directive selects a router other than the default root router.

```
<ywp>
@zpe router /router.ywp

@get("/products/{id}")
function product(number id)
  return RESPONSE->json(["id" => id])
end function

@post("/search")
function submit_search(string q)
  return "Searching for " & q
end function
</ywp>
```

The route path matches the URL path only. The dynamic segment `id` is passed to the function, while query-string and form values can be supplied by name or read through `REQUEST`. Route declarations can use `@get`, `@post`, `@put`, and `@delete`. Keep route functions small: validate the inputs, call application functions, and return the appropriate response.

19.7 Running a local web site

ZPE includes a local development server for testing YWP and related web resources. Give it the directory containing the site's `index.ywp`:

```
zpe -z /path/to/site
zpe -z /path/to/site --port 8080
```

The local server is intended for development and testing. Production deployment should use the appropriate web-server configuration, HTTPS, permissions, and operational controls.

19.8 Web chapter exercises

1. Create an `index.ywp` page that prints a welcome message.
2. Add a query-string name and display a different greeting when it is present.
3. Add a form that accepts a subject and displays it after submission.
4. Validate an empty form field and display a helpful message.
5. Return a map as JSON through `RESPONSE->json(...)`.
6. Set a plain-text response header and compare it with an HTML response.
7. Set a not-found status code for a missing resource.
8. Redirect a visitor from one page to another.
9. Store and read a session value.
10. Move the navigation into a reusable rendered page.
11. Add a route for a product identifier.
12. Add a POST route for a search form.
13. Read a route parameter and a query-string parameter in the same handler.
14. Run the site using the local development server.
15. Add a second route and explain why the two handlers should remain separate.

Appendix A

Running ZPE

This appendix collects the most useful ZPE modes. The exact options available depend on the installed version, so use `zpe -h` when a mode needs more detail.

A.1 Running a program

Run a `.yas` source file by passing its path to ZPE:

```
zpe hello.yas
```

Interactive mode is useful for trying short expressions and checking ideas immediately:

```
zpe -i
```

A.2 Useful modes

The tree mode displays the parsed structure of a program, while web mode processes a web page containing YASS. Debug mode connects a program to a debugger such as ZIDE.

```
zpe -t hello.yas
zpe -w index.ywp
zpe -d hello.yas
```

The local web-development server uses `-z` and a document-root directory:

```
zpe -z /path/to/site
```

Use the help system to inspect modes, configuration, built-in objects, modules, and functions:

```
zpe -h
zpe -h config
zpe -h objects
zpe -h modules
zpe -h functions
```

A.3 Version and documentation information

The installed version can be checked before diagnosing a difference between two machines:

```
zpe --version
```

Appendix B

The ZPE formal specification

The complete grammar is maintained in the official ZPE/YASS documentation as the formal specification. It is the right place to consult the BNF when a question concerns the language grammar rather than the teaching examples in this book. The specification also provides a useful reference when implementing tools such as syntax highlighters, formatters, parsers, and language plugins.

The grammar should be read alongside the documentation for the installed runtime. ZPE supports several compatible writing styles, and newer releases may add productions while preserving existing programs. The official documentation is available at https://www.jamiebalfour.scot/projects/zpe/documentation/; follow its Technical reference section to the formal specification.

When using the BNF, remember that a grammar describes what can be written, not whether a complete program has a useful purpose. A syntactically valid program may still fail at runtime because it refers to missing files, invalid input, unavailable modules, or an object used outside its permitted scope.

About the Examples

The examples in this edition are intentionally short. As the book develops, they will grow into complete programs and will introduce functions, data structures, files, modules, objects, and optional typing with TYPO.