**Index**

| **Chapter** | **Title** | **Page** |
| --- | --- | --- |
| **1** | **Introduction to .g** | **2-5** |
| **2** | **Language Basics** | **6-10** |
| **3** | **Control Flow** | **11-15** |
| **4** | **Functions** | **16-20** |
| **5** | **Logic and Conditions** | **21-26** |
| **6** | **Arrays and Objects** | **27-31** |
| **7** | **Built-in Functions** | **32-37** |
| **8** | **Error Handling** | **38-41** |
| **9** | **The Standard Library (std@1.0/)** | **42-47** |
| **10** | **Modules and Imports** | **48-52** |
| **11** | **User-Defined Types** | **53-57** |
| **12** | **Building CLI and File Tools** | **58-63** |
| **13** | **Future Features & Language Design** | **64-67** |
| **A** | **Appendix A: Grammar & Syntax Reference** | **68-71** |
| **B** | **Appendix B: Keywords, Operators, Built-ins** | **72-74** |
| **C** | **Future Updates & Feedback** | **75** |

# **Chapter 1 — Introduction to .g**

**1.1 Overview**

**.g is a lightweight, cross-platform scripting language designed for simplicity, clarity, and modular development. It is built entirely in JavaScript and executed on the Node.js runtime. The language emphasizes expression-based design, minimal syntax, and extensibility through a built-in standard library.**

**.g is ideal for:**

* **Writing automation scripts**
* **Prototyping tools**
* **Educational interpreters**
* **Extending apps with custom logic**

**Its syntax avoids noise (no semicolons), encourages readable code, and provides powerful standard features such as file I/O, string manipulation, math utilities, and structured control flow.**

**1.2 Motivation**

**.g was created to solve a personal need: a scripting language that is fully transparent, customizable, and embeddable into other projects. Unlike larger languages with complex runtimes or legacy syntax, .g offers a controlled environment where every rule, token, and feature is built from scratch.**

**Core motivations include:**

* **Full ownership of language behavior and structure**
* **Simplicity without sacrificing functionality**
* **Integration into Node.js-based tools and scripts**
* **Extensibility through modular standard libraries**

**Whether you're writing scripts or building a custom REPL, .g gives you a solid foundation with minimal overhead.**

## **1.3 Installation**

**.g is distributed via** [**npm**](https://www.npmjs.com/) **under the package name @singhgurjot/g-lang.**

**To install it globally:**

****

**This provides a global g command that can be used from any terminal session.**

**Requirements:**

* **Node.js v14+**
* **npm (Node Package Manager)**

## **1.4 Your First .g Program**

**Create a file named hello.g:**

****

**Then execute it using:**

****

**Expected output:**

****

**This command invokes the .g interpreter using the global CLI, parses your code, and executes it in a scoped environment.**

## **1.5 Features at a Glance**

**The .g language supports a growing set of built-in capabilities:**

**Core Language**

* **Variables via var**
* **Expression-based flow (if, else, while)**
* **Functions with function and return**
* **Arrays, objects, indexing, and property access**
* **Logical and comparison operators (==, !=, &&, ||, etc.)**

### **Standard Library: std@1.0/**

**The standard library is bundled by default and includes:**

* **math: Random, rounding, trigonometry, constants (math.PI, math.floor(), etc.)**
* **string: Case manipulation, splitting, searching, replacing**
* **array: Push, pop, length, inclusion checks**
* **fs: Read, write, append, delete files**
* **time: Current timestamp, ISO time, blocking sleep**
* **cli: input() prompts and args() for command-line arguments**
* **json: toJSON() and parseJSON() for serialization**

**All modules are natively available — no extra setup needed.**

**1.6 Cross-Platform Compatibility**

**Because .g is implemented in JavaScript and runs on Node.js, it works on:**

* **Windows**
* **macOS**
* **Linux**
* **Any system that supports Node.js**

**Once installed via npm, it behaves like any standard CLI tool.**

## **1.7 Summary**

**.g is a script-first, logic-focused language built for practical use. It’s simple to learn, transparent by design, and ready to scale with your projects. With its minimal syntax, modular standard library, and clean execution model, .g gives you the control of a scripting language — without the complexity.**

#

#

# **Chapter 2 — Language Basics**

**This chapter introduces the foundational syntax and core constructs of the .g language. You’ll learn how to declare variables, print output, work with different data types, and build expressions — everything you need to start writing real .g programs.**

##

## **2.1 Variable Declarations**

**Variables in .g are declared using the var keyword. A variable holds a value and can be reassigned later.**

### **Syntax:**



**Variables are dynamically typed — the type is inferred from the value assigned.**

##

##

## **2.2 Built-in Data Types**

**.g supports several primitive and compound data types:**

| **Type** | **Example** |
| --- | --- |
| **Number** | **var a = 42, var pi = 3.14** |
| **String**  | **var s = "hello"** |
| **Boolean** | **var flag = true** |
| **Null** | **var x = null** |
| **Array** | **var items = [1, 2, 3]** |
| **Object** | **var person = { name: "Aman" }** |

**These types are compatible with built-in functions like type(), length(), and operators such as +, ==, and &&.**

## **2.3 Output with say**

**Use the say statement to print output to the console.**

### **Example:**

****

**Output:**

****

**2.4 Arithmetic Expressions**

**.g supports standard arithmetic operations:**

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| **+** | **Addition** | **X + Y** |
| **-** | **Subtraction** | **X - Y** |
| **\*** | **Multiplication** | **X \* Y** |
| **/** | **Division** | **X / Y** |

**Example:**

****

**All expressions follow standard mathematical precedence.**

**2.5 Logical and Comparison Operators**

**Logic and conditionals are built-in and follow familiar patterns.**

| **Operator** | **Meaning** | **Example** |
| --- | --- | --- |
| **==** | **Equal too** | **x == 5** |
| **!=** | **Not equal to** | **x != 10** |
| **>** | **Greater than** | **x > 2** |
| **<** | **Less then** | **x < 5** |
| **>=** | **Greater than or Equal too** | **x >= 10** |
| **<=** | **Less than or Equal too** | **x <= 10** |
| **&&** | **Logical AND** | **a && b** |
| **!** | **Logical OR** | **!isReady** |

**Booleans are strictly evaluated, and null is treated as falsy.**

## **2.6 Comments**

**Use // to write comments:**

****

**Comments are ignored by the interpreter.**

## **2.7 Code Formatting**

**.g has no required indentation rules, but clean formatting is encouraged.**

* **No semicolons are needed**
* **Curly braces {} are used for blocks**
* **Expressions and statements can span multiple lines if necessary**

### **Example:**

****

## **2.8 Summary**

**In this chapter, you’ve learned the essential building blocks of .g:**

* **Declaring variables with var**
* **Working with numbers, strings, booleans, arrays, and objects**
* **Printing output with say**
* **Performing arithmetic and logic operations**
* **Writing clean, readable code**

**These are the foundations for writing every .g program. In the next chapter, we’ll explore control flow in depth — including conditionals and loops.**

# **Chapter 3 — Control Flow**

**Control flow statements allow your program to make decisions and repeat actions based on conditions. .g includes two fundamental control structures: if statements and while loops. This chapter covers their syntax, usage, and best practices.**

**3.1 Conditional Execution: if and else**

## **Use if to execute code only when a condition is true. You can optionally include an else block for alternative behavior.**

## **Syntax:**



**Example:**

****

**Output:**

****

**Conditions can use comparison operators (==, >, etc.) or any expression that returns a boolean.**

## **3.2 Chained Conditions with else if**

**.g does not have a dedicated else if keyword. However, you can chain if statements inside else blocks:**

****

**3.3 Repeating Code with while**

**Use a while loop to repeat a block of code as long as a condition remains true.**

### **Syntax:**



**Example:**

****

**Output:**

****

**Be cautious: a while loop will continue forever if the condition never becomes false.**

**3.4 Nesting and Scope**

**Both if and while support nested blocks:**

****

**Output:**

****

**Variables declared with var inside the loop persist in the outer scope unless re-declared.**

## **3.5 Null and Boolean Conditions**

**Any valid .g expression can be used as a condition. Some notes:**

* **true and false are native boolean literals.**
* **null is treated as false.**
* **Strings, numbers, and arrays are truthy unless explicitly compared.**

##

## **3.6 Common Pitfalls**

* **Always use {} braces — no implicit indentation-based blocks.**
* **Ensure that loop variables are updated inside the loop to avoid infinite loops.**
* **Boolean operators (&&, ||, !) must be written correctly — use parentheses if in doubt.**

## **3.7 Summary**

**This chapter covered .g's control flow capabilities:**

* **Conditional execution with if, else, and nested blocks**
* **Looping with while**
* **Boolean logic and expression-based decisions**
* **Proper scoping and structure**

**In the next chapter, we’ll dive into functions — how to define, call, and return values in reusable logic blocks.**

# **Chapter 4 — Functions**

**Functions allow you to group code into reusable blocks that can be executed multiple times with different inputs. In .g, functions are first-class citizens and support parameters, return values, and modular usage.**

## **4.1 Function Declarations**

**Functions are defined using the function keyword, followed by a name, a parameter list in parentheses, and a code block in {} braces.**

### **Syntax:**



**Example:**

****

**This defines a function named greet that accepts a single parameter.**

##

##

## **4.2 Calling Functions**

**Once defined, a function can be called by using its name followed by parentheses.**

### **Example:**



**Arguments are passed in the same order as parameters.**

## **4.3 Returning Values**

**Use return inside a function to send back a value to the caller.**

### **Example:**



**If a function has no return, it returns null by default.**

##

##

##

## **4.4 Local Scope**

**Variables defined inside a function are local to that function.**

****

**The variable result is not accessible outside the function.**

## **4.5 Nested Function Calls**

**Functions can be nested or used inside expressions.**

****

**Functions are evaluated from the inside out, like regular expressions.**

## **4.6 Function Composition & Reuse**

**Functions can call each other, or themselves (for recursion), though .g currently lacks tail-call optimization or stack guards. You can safely structure helper functions like this:**

****

## **4.7 Exporting and Importing Functions**

**Functions can be shared across .g files using export and import. For now, we’ll just note that this feature exists — a full breakdown is in Chapter 12.**

**Import.g:**

****

**Export.g:**

****

## **4.8 Summary**

**Functions in .g are compact, expressive, and easy to reuse. This chapter introduced:**

* **Declaring functions with function**
* **Passing parameters and arguments**
* **Using return to return values**
* **Calling and composing functions**
* **Function scope rules**

**Next, we’ll move into the logic system of .g — covering equality, logical operators, and how to build expressive conditions with clean syntax.**

# **Chapter 5 — Logic and Conditions**

**The logic system in .g allows you to evaluate conditions, compare values, and control the flow of your program using boolean expressions. This chapter covers equality checks, logical operators, and how they interact with other core features like if, while, and functions.**

## **5.1 Boolean Values**

**.g includes two native boolean literals:**

****

**These are used in conditionals, comparisons, and logical expressions.**

****

##

##

##

##

## **5.2 Comparison Operators**

**Comparison operators return a boolean result and are commonly used in conditions.**

| **Operator** | **Meaning** | **Example** |
| --- | --- | --- |
| **==** | **Equal to** | **X == 5** |
| **!=** | **Not Equal to** | **name != "Grey"** |
| **>** | **Greater than** | **x > 3** |
| **<** | **Less than** | **x < 10** |
| **>=** | **Greater or equal** | **score >= 80** |
| **<=** | **Less or equal** | **score <= 50** |

**Example:**

****

## **5.3 Logical Operators**

**Logical operators combine boolean values and expressions.**

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| **&&** | **Logical AND (both true)** | **x > 0 && x < 10** |
| **!** | **Logical NOT (negation)** | **!isReady** |

**Example:**

****

## **5.4 Truthy and Falsy Values**

**In .g, expressions are strictly evaluated. The following values are treated as falsy:**

* **false**
* **null**

**Everything else (including 0, "", []) is considered truthy by default unless compared explicitly.**

****

**Output:**

****

## **5.5 Logical Grouping with Parentheses**

**Use parentheses () to group logical conditions and control evaluation order.**

****

**Logical expressions without parentheses follow standard operator precedence: ! > && > ||**

## **5.6 Negation with !**

**You can invert any boolean expression using the ! (NOT) operator.**

****

**Output:**

****

## **5.7 Combining Conditions**

**Logical expressions can be nested and combined for clarity.**

****

## **5.8 Summary**

**This chapter introduced .g's logical and comparison system, including:**

* **Boolean literals (true, false)**
* **Comparison operators (==, !=, >, etc.)**
* **Logical operators (&&, ||, !)**
* **Grouping expressions with parentheses**
* **Understanding truthy and falsy values**

**In the next chapter, we’ll explore arrays and objects — and how they’re used to structure and organize data in .g.**

# **Chapter 6 — Arrays and Objects**

**Data structures are essential for organizing and managing values in any language. .g supports two primary compound data types: arrays and objects. This chapter explores how to create, access, and manipulate them effectively.**

## **6.1 Arrays**

**An array is an ordered list of values. Use square brackets [] to define them.**

### **Example:**

****

**Arrays can contain any value type, including other arrays or objects.**

### **6.1.1 Accessing Array Elements**

**Use zero-based indexing with square brackets:

**

**Accessing an invalid index returns null.**

### **6.1.2 Modifying Arrays**

**Use assignment to update values:**

****

**Note: this requires the index to already exist.**

### **6.1.3 Built-in Array Functions**

**The standard library provides array helpers under the array namespace:**

| **Function** | **Description** |
| --- | --- |
| **array.length(arr)** | **Returns array length** |
| **array.push(arr, val)** | **Appends a value** |
| **array.pop(arr)** | **Removes and returns last item** |
| **array.includes(arr, v)** | **Returns true if value exists** |

**Example:**

****

## **6.2 Objects**

**Objects store key–value pairs, and are defined using curly braces {}.**

### **Example:**



**Keys are always strings; values can be any type.**

###

### **6.2.1 Accessing and Updating Properties**

**Use dot notation or bracket notation:**

****

**Dot notation is preferred for known keys.**

###

###

###

###

### **6.2.2 Nested Objects**

**Objects can be nested for structured data:**

****

## **6.3 Arrays of Objects**

**These structures are common when modeling collections:**

****

**You can combine loops with arrays to iterate through them (covered in Chapter 9).**

## **6.4 Objects as Function Arguments**

**Objects work well for passing structured parameters:**

****

## **6.5 Summary**

**In this chapter, you learned how to structure and manipulate data using:**

* **Arrays — ordered lists accessed via index**
* **Objects — key–value mappings accessed via property names**
* **Dot/bracket notation**
* **Built-in functions like array.length() and array.includes()**

**Next, we’ll cover built-in functions like type(), length(), and how to use .g’s utility functions to inspect and process data.**

# **Chapter 7 — Built-in Functions**

**The .g language provides a collection of built-in functions that handle common tasks like inspecting types, reading input, working with files, and manipulating data. These functions are available by default in every .g script and do not require any imports.**

## **7.1 type(value)**

**Returns the type of the given value as a string.**

### **Examples:**



## **7.2 length(value)**

**Returns the length of a string or array.**

### **Examples:**

****

**Throws an error for unsupported types.**

## **7.3 input(prompt)**

**Prompts the user for input. Returns a string. If no prompt is provided, it uses an empty one.**

****

## **7.4 args()**

**Returns an array of command-line arguments passed when running the .g script.**

### **Command:**

****

### **Script:**



##

##

##

## **7.5 File I/O Functions**

**These functions enable interaction with the file system.**

### **read(filename)**

**Reads and returns the contents of a file.**

****

### **write(filename, content)**

**Writes content to a file (overwrites existing file or creates a new one).**

****

### **append(filename, content)**

### **Appends content to an existing file.**

****

###

###

###

### **append(filename, content)**

**Appends content to an existing file.**

****

### **exists(filename)**

**Returns true if the file exists.**

****

### **delete(filename)**

**Deletes a file.**

****

## **7.6 JSON Utilities**

### **toJSON(value)**

**Serializes a .g value (object or array) into a JSON string.**

****

###

### **parseJSON(string)**

**Parses a JSON string and returns the corresponding .g object.**

****

## **7.7 Time Utilities**

### **now()**

**Returns the current time in ISO format.**

****

### **timestamp()**

**Returns the current time in milliseconds since the UNIX epoch.**

****

###

###

###

###

### **sleep(ms)**

**Pauses execution for a specified number of milliseconds. (Blocking)**

****

## **7.8 Summary**

**This chapter introduced the built-in functions available globally in .g. You now know how to:**

* **Inspect values using type() and length()**
* **Prompt user input and access CLI args**
* **Perform file I/O with read(), write(), append(), etc.**
* **Use toJSON() and parseJSON() for structured data**
* **Access current time and pause execution**

**In the next chapter, we’ll cover error handling in .g, including how to use try and catch to prevent crashes and gracefully handle unexpected failures.**

#

# **Chapter 8 — Error Handling**

**In any programming language, runtime errors are inevitable. The .g language supports structured error handling using try and catch blocks. This allows you to detect, isolate, and respond to failures — without crashing your entire program.**

## **8.1 The try / catch Statement**

**The try block lets you run a section of code that might fail. If an error occurs, the catch block executes instead.**

### **Syntax:**

****

**Both blocks are required. You cannot use try without a catch.**

**Example:**

****

**If missing.txt does not exist, the read() function will throw an error — and the catch block will handle it without stopping the script.**

## **8.2 Scope Inside try / catch**

**Variables declared inside the try block are local to that block — but values can be safely passed or reassigned:**

****

**This works because the outer result is updated in both paths.**

## **8.3 Common Error Sources**

**Some typical cases where try/catch is useful:**

| **Function** | **Error Condition** |
| --- | --- |
| **read(file)** | **File not found** |
| **delete(file)** | **File doesn’t exist or can’t be deleted** |
| **parseJSON(text)** | **Invalid JSON string** |
| **array.pop()** | **Popping from an empty array** |
| **type()** | **Unsupported type** |

**Use try/catch to isolate unsafe logic or file-dependent operations.**

## **8.4 No Error Object (yet)**

**In .g, the catch block does not receive an error object. You cannot access the exact message or stack trace at this time.**

**Future versions of .g may support:**

****

**But for now, error details are only printed to the console (if not caught).**

## **8.5 Nested Error Handling**

**You can nest try/catch blocks to handle different levels of failure.**

****

**Use sparingly — nested try blocks can make code harder to follow.**

## **8.6 Defensive Programming**

**Error handling isn’t just about reacting — it’s about anticipating. Use exists() and type() to avoid errors before they occur.**

****

**This keeps your code safer and cleaner.**

## **8.7 Summary**

**In this chapter, you’ve learned how to make your .g programs more reliable by using:**

* **try and catch blocks**
* **Defensive checks like exists() and type()**
* **Structuring fallback logic**
* **Avoiding common failure points**

**Error handling makes .g suitable for real-world scripting — where missing files, bad inputs, or failed operations shouldn't bring everything to a halt.**

**In the next chapter, we’ll dive into the .g Standard Library (std@1.0/) and explore its math, string, array, and utility modules in detail.**

# **Chapter 9 — The Standard Library (std@1.0/)**

**.g ships with a built-in standard library under the namespace std@1.0/. It includes powerful modules for math, strings, arrays, files, timing, CLI tools, and JSON — allowing you to write real-world scripts without external dependencies.**

**No import is required — these functions are available by default.**

## **9.1 math Module**

**Provides math utilities, constants, and conversions.**

### **Common Math Functions:**



**Constants and Trig:**

****

**9.2 string Module**

**Utilities for string transformation, searching, and slicing.**

### **Examples:**

****

**Advanced:**

****

## **9.3 array Module**

**Basic array manipulation functions.**

### **Examples:**

****

**Note: Arrays are passed by reference — modifications are permanent.**

## **9.4 fs Module (File I/O)**

**Built-in global functions, not namespaced under fs, but conceptually grouped here.**

**Read and Write:**

****

**File Existence and Deletion:**

****

##

## **9.5 time and sleep Utilities**

## **Timestamps and ISO Strings:**

****

**Blocking Sleep:**

****

## **9.6 CLI Tools**

**Use input() for prompts and args() for command-line arguments.**

### **Input Prompt:**

****

**Reading CLI Args:**

****

****

##

##

## **9.7 JSON Handling**

### **Convert Object to JSON:**



**Parse JSON String:**

****

**9.8 Directory Utilities**

****

**Throws error if the path is invalid or not a directory.**

##

##

##

##

## **9.9 Summary**

**The std@1.0/ library makes .g powerful out of the box, covering:**

* **Math utilities and constants**
* **String manipulation**
* **Array operations**
* **File system access**
* **Timing and delays**
* **Input and argument handling**
* **JSON parsing**
* **Directory listing**

**In the next chapter, we’ll explore .g's import/export system for organizing code across files and creating reusable modules.**

# **Chapter 10 — Modules and Imports**

**As your projects grow, organizing code across multiple files becomes essential. .g supports modular programming using import and export declarations — allowing functions to be shared between files and reused cleanly.**

**This chapter covers how .g handles modules, how the import/export system works, and how to write modular, maintainable .g programs.**

## **10.1 Exporting Functions**

**To make a function available to other .g files, use the export keyword before the function declaration.**

### **Example: math.g**

****

**Only exported functions can be imported into other scripts. Non-exported functions are local to their file.**

##

##

## **10.2 Importing Functions**

**Use the import statement to bring exported functions from another .g file into the current one.**

### **Syntax:**

****

**The path must be a string literal and must include the full file name, including the .g extension.**

**Example: main.g**

****

## **10.3 How Imports Work Internally**

**When an .g file is run, the interpreter does the following:**

1. **Resolves the file path (relative to the current script).**
2. **Reads and parses the imported file.**
3. **Look for export declarations.**
4. **Caches the parsed module to prevent repeated imports.**
5. **Makes selected exported functions available to the current script.**

**Only exported function declarations are currently supported. Variable or constant exports are not allowed.**

**10.4 Avoiding Name Collisions**

**Functions are imported into the global scope, so avoid name collisions:**

****

**Overwriting an imported name is possible but discouraged. If a function is already defined, .g will use the last assignment.**

## **10.5 Importing from Multiple Files**

**You can import from more than one .g file within the same script.**

****

**Each import must use its own import { ... } from "..." block.**

##

##

##

## **10.6 Import Errors**

**If a requested function isn’t exported by the module, the interpreter throws a clear error:**

****

**If the file does not exist or contains syntax errors, the import will fail at runtime.**

## **10.7 No Circular Imports**

**The interpreter prevents circular imports using internal caching. If a file tries to import itself (directly or indirectly), it is skipped silently to avoid infinite loops.**

## **10.8 Suggested File Structure**

**Here’s a suggested folder layout for modular .g projects:**

****

**Use relative paths in import statements to mirror this structure:**

****

## **10.9 Summary**

**.g modules let you split logic into files and reuse functions cleanly. You now know how to:**

* **Use export to expose functions from one file**
* **Use import to load them in another**
* **Resolve .g files with relative paths**
* **Prevent naming conflicts and circular imports**

**In the next chapter, we’ll explore user-defined data structures, including how to model structured information using nested objects and custom conventions.**

# **Chapter 11 — User-Defined Types**

**.g is a dynamically typed language with no explicit type declarations, but you can still design structured data using objects, arrays, and consistent naming patterns. This chapter explains how to model real-world entities and organize data using user-defined types and conventions.**

**11.1 Objects as Custom Types**

**You can represent structured entities (like users, products, or settings) using plain objects.**

### **Example: A "User" type**



**This object behaves like a "User" type — with named fields that can be accessed or modified.**

**11.2 Accessing and Updating Fields**

## **Use dot notation to access or update values:**

****

**You can also use bracket notation for dynamic access:**

****

## **11.3 Nested Structures**

**Objects can contain other objects or arrays, creating rich data models.**

**Example: A user with nested contact info**

****

**11.4 Creating and Using Custom Conventions**

**Since .g lacks class declarations or interfaces, structure is enforced by consistency:**

****

**You can think of makeUser() as a constructor function.**

## **11.5 Checking and Validating Fields**

**You can inspect structure using type() and in:**

****

**Currently .g does not include hasOwnProperty() or a full in operator — but field existence can be manually checked like this:**

****

**11.6 Arrays of Objects**

**Objects are often stored in arrays when managing collections:**

****

**This approach scales well in loops, filtering, and modular design.**

## **11.7 Utility Patterns**

## **1. Enum-like objects:**



**2. Struct-style fixed shapes:**

****

**You can clone or fill this shape per item.**

## **11.8 Summary**

**While .g doesn’t support classes or types natively, you can still design rich, structured data using:**

* **Plain objects with consistent field names**
* **Functions as pseudo-constructors**
* **Arrays for collections of objects**
* **Dot/bracket access for reading/writing fields**
* **Nested models for complex hierarchies**

**In the next chapter, we’ll simulate file system and command-line interaction by building file-based tools and CLI applications using .g.**

# **Chapter 12 — Building CLI and File Tools**

**.g includes built-in support for reading files, writing logs, and handling command-line arguments — making it well-suited for automation scripts, lightweight tools, and simple CLI utilities. This chapter walks through real examples and best practices for scripting with .g.**

## **12.1 Accessing CLI Arguments with args()**

**When you run a .g script with extra arguments, they are accessible via the global args() function.**

### **Example:**

****

****

**Output:**

****

**Arguments are passed as strings, and args() always returns an array.**

## **12.2 Prompting Input with input()**

**Use input() to ask the user for input during script execution.**

****

**This pauses execution and waits for the user to type a response.**

## **12.3 Writing and Appending to Files**

**.g allows you to write logs, configs, or data files with built-in file I/O.**

### **Example: Logging events**

****

**You can read back the file contents using:**

****

##

##

## **12.4 Checking for Files Before Reading**

**Use exists(filename) to verify that a file is present.**

****

**This prevents runtime errors and allows safe fallback logic.**

## **12.5 Deleting Files**

**To remove a file permanently:**

****

**Use with caution — this cannot be undone.**

##

##

##

##

## **12.6 Building a Real CLI Utility**

**Here’s a real CLI example: a .g script that counts words in a file.**

### **wordcount.g**



**Run it:**

****

##

##

##

## **12.7 File-Based Configuration**

**You can simulate config loading with parseJSON():**

****

**This makes .g useful for dynamic or scriptable tools where options are stored in files.**

## **12.8 Combining Everything: Simple Installer**

### **installer.g**

****

**Create a folder, copy this script, and .g becomes your project bootstrapper.**

**12.9 Summary**

**With built-in tools like read(), write(), args(), and input(), .g can be used for:**

* **Simple installers**
* **Config managers**
* **CLI-based editors**
* **File processors**
* **Scripting small DevOps tasks**

**In the next chapter — the final one — we’ll look ahead: what’s possible for .g in future versions, and how you can contribute, extend, or customize it to fit your own needs.**

# **Chapter 13 — Future Features & Language Design**

**The .g language was built from the ground up — not just to run code, but to be hacked, shaped, and extended by its users. This chapter offers a look ahead at possible future directions, features under consideration, and the philosophy guiding .g's continued development.13.1 Design Principles**

**.g is driven by a few key principles:**

* **Clarity over complexity**
* **Minimalism with power**
* **Everything is understandable**
* **Built with JS, embeddable in JS**
* **Script-first, tool-friendly**

**These values will continue to guide how new features are added and how breaking changes are avoided.**

## **13.2 Planned & Proposed Features**

**While .g is already usable for scripting and automation, several enhancements are being explored:**

### **In Progress / Experimental**

* **Namespaces & Scoped Imports
 Avoid global pollution and allow aliasing:**

****

* **Native Modules for std@1.1
 Adding env, http, os, crypto, and path modules.**
* **Improved catch support
 Allow accessing error messages inside catch blocks:**

****

### **Under Consideration**

* **Pattern Matching
 Simple switch-case alternative for object structures and conditions.**
* **Classes or Prototypes
 A lightweight way to define reusable object templates (optional).**
* **Lambda / Arrow Functions
 Short function syntax:**

****

* **Command Alias System
 Allow g to execute .g scripts globally from any folder using registered aliases.**
* **Code Editor Plugin
 VS Code support with syntax highlighting and .g language extensions.**

###

###

### **Won’t Add (At Least for Now)**

* **Complex type system**
* **Implicit returns**
* **Global mutation of built-ins**
* **Heavy class inheritance**

**.g will remain simple, flat, and learnable, even as it grows.**

## **13.3 Language Internals: Extend & Develop**

**Since .g is implemented entirely in JavaScript, you can:**

* **Modify the tokenizer, parser, or interpreter**
* **Add new keywords or operators**
* **Replace test.js with a custom CLI**
* **Add new standard modules in std@1.x/**

**The project was intentionally designed to be *understandable*, not magic.**

## **13.4 How to Contribute**

**Even if .g is your own private language, it can still be open to collaboration:**

* **Write and share .g modules**
* **Build tools that extend the CLI**
* **Contribute feature ideas to a roadmap**
* **Fork and remix the interpreter**
* **Create tutorials or share .g demos**

**The more you use .g, the more it evolves.**

##

##

## **13.5 Summary**

**This final chapter outlines where .g is headed and how its flexible foundation allows it to grow without becoming bloated. Whether you're writing automation, building CLI tools, or just learning language internals, .g gives you something rare:**

**A language you can own, understand, and extend without asking permission.**

**You're now equipped with everything needed to write .g scripts, build modules, handle files, process input, and shape your own programming environment.**

# **Appendix A — Grammar & Syntax Reference**

# **This appendix provides an overview of .g’s syntax rules and grammar structures. It’s not a full formal grammar, but rather a high-level summary for developers.**

#

## **A.1 General Structure**

# **Statements are line-based (no semicolons).**

# **Blocks are enclosed in { }.**

# **Indentation is optional but recommended.**

**A.2 Variable Declaration**

****

* **All variables declared with var**
* **Dynamic typing**

## **A.3 Expressions**

## **Expressions return values and can be nested:**

****

**Supported types: number, string, boolean, null, array, object**

## **A.4 Control Flow**

### **If/Else:**

****

**While Loop:**

****

## **A.5 Functions**

### **Declaration:**



**Return:**

****

**A.6 Arrays**

****

**A.7 Objects**

****

## **A.8 Import/Export**

### **Export:**



**Import:**

****

**A.9 Try/Catch**

****

#

#

#

#

#

#

#

#

# **Appendix B — Language Reference**

# **A compact reference sheet for .g developers.**

## **B.1 Reserved Keywords**



**B.2 Operators**

| **Symbol** | **Meaning** |
| --- | --- |
| **+** | **Add / Concatenate** |
| **-** | **Subtract** |
| **\*** | **Multiply** |
| **/** | **Divide** |
| **==** | **Equal** |
| **!=** | **Not Equal** |
| **<** | **Less Than** |
| **>** | **Greater That** |
| **<=** | **Less or Equal** |
| **>=** | **Greater or Equal** |
| **&&** | **Logical AND** |
| **!** | **Logical NOT** |
| **=** | **Assignment** |
| **.** | **Property Access** |
| **[]** | **Index Access** |

# **B.3 Built-in Functions (Global)**

| **Function** | **Description** |
| --- | --- |
| **say(...)** | **Print to console** |
| **type(x)** | **Get type of value** |
| **length(x)** | **Length of string or array** |
| **input(prompt)** | **Ask for user input** |
| **args()** | **Get CLI arguments as array** |
| **read(file)** | **Read file contents** |
| **write(file, c)** | **Write file (overwrite)** |
| **append(file, c)** | **Append to file** |
| **delete(file)** | **Delete file** |
| **exists(file)** | **Check file existence** |
| **toJSON(obj)** | **Serialize object** |
| **parseJSON(str)** | **Parse JSON string** |
| **now()** | **Current ISO time string** |
| **timestamp()** | **UNIX timestamp in ms** |
| **sleep(ms)** | **Pause execution** |
| **listdir(path)** | **List directory contents** |

#

**B.4 Built-in Namespaces**

| **Namespace** | **Functions** |
| --- | --- |
| **math** | **random(), round(), pow(), PI()** |
| **string** | **upper(), lower(), split(), slice()** |
| **array** | **push(), pop(), length(), includes()** |

#

#

#

#

#

#

#

# **Future Updates & Feedback**

# **The .g language is actively evolving — and your ideas, issues, and feedback can shape its direction.**

### **What's coming in future versions:**

* **Scoped imports and namespace aliasing**
* **New std@1.1/ modules (env, http, crypto)**
* **Lambda/arrow function support**
* **Community-driven examples and toolkits**

###  **Have suggestions or bugs to report?**

**You're invited to contribute to the language’s growth. If you've built a tool, found a bug, or have a feature request, reach out directly:**

**Contact:**

 **Gurjotpal Singh
 singhgurjotpal84@gmail.com**

**All feedback is welcome — whether you’re a beginner, hobbyist, or advanced script developer.**

**Thanks for choosing .g**