CODING FUNDAMENTALS

ACROSS LANGUAGES

A Practical Guide to Core Programming Concepts in PHP, Node.js, Go, Kotlin, and Java

PHP • NODE.JS • GO • KOTLIN • JAVA

Author

SANTOKH SINGH SAGGU

CODING FUNDAMENTALS

ACROSS LANGUAGES

A Practical Guide to Core Programming Concepts
in PHP, Node.js, Go, Kotlin, and Java

PHP · NODE.JS · GO · KOTLIN · JAVA

SANTOKH SINGH SAGGU

July 2026

Author : Santokh Singh Saggu

Table of Contents

Preface

Every programming language, no matter how different it looks on the surface, is built on the same small set of ideas: you store data, you make decisions, you repeat actions, and you organize logic into reusable pieces. Once you understand these ideas in the abstract, learning a *new* language becomes mostly a matter of learning new syntax for old concepts — not learning to think all over again.

This book is organized around that principle. Each chapter covers one core programming concept in **general, language-agnostic terms first**, explaining *why* it exists and *how* it's typically used. Then the same concept is shown side-by-side in five real languages:

- **PHP** — a server-side scripting language, dynamically typed, widely used for web backends.
- **Node.js (JavaScript)** — a dynamically typed language that runs both in browsers and on servers via the Node.js runtime.
- **Go (Golang)** — a statically typed, compiled language designed by Google for simplicity and concurrency.
- **Kotlin** — a statically typed language that runs on the JVM, designed as a modern, safer alternative to Java.
- **Java** — a statically typed, compiled (to bytecode) language that has powered enterprise software for decades.

Seeing the same idea expressed five different ways makes the *underlying concept* easier to see, because you stop confusing "how PHP does it" with "what a variable actually is."

Chapter 1: Variables and Constants

1.1 The General Idea

A **variable** is a named container for a piece of data that can change over the life of a program. When you write `age = 25`, you're telling the computer: "reserve a spot in memory, call it `age`, and put the value `25` there." Later, you can update that spot — `age = 26` — and every part of the program that reads `age` will see the new value.

A **constant** is the same idea, except the value is locked in once it's set. Constants exist because some values genuinely should never change during a program's execution — the value of pi, a maximum retry count, an API base URL, a tax rate. Marking something as a constant is a signal to both the compiler/interpreter and to other programmers: "if you try to change this, something is wrong, and the language should tell you so."

Underneath, most languages distinguish variables along two more dimensions:

- **Typing**: Is the variable's data type fixed when it's declared (statically typed, e.g., Go, Kotlin, Java), or can it hold any type and change types later (dynamically typed, e.g., PHP, JavaScript)?
- **Scope**: Where in the program is this variable visible — the whole file, just a function, just a block of code inside `{ }`?

Good naming and disciplined use of constants over "magic numbers" (unexplained literal values scattered through code) are two of the cheapest ways to make code easier to read and maintain.

1.2 PHP

```
<?php
// Variable — starts with a dollar sign, no type declaration needed
$age = 25;
$age = 26; // reassignment is allowed

// Constant — defined once, cannot be changed afterward
define('MAX_RETRIES', 3);
// or, since PHP 7.1, using the const keyword (works at top level or in a
class)
const TAX_RATE = 0.18;

echo $age;         // 26
echo MAX_RETRIES;  // 3
```

1.3 Node.js (JavaScript)

```
// let — a variable that can be reassigned
let age = 25;
age = 26; // fine

// const — cannot be reassigned after its initial value is set
const MAX_RETRIES = 3;
const TAX_RATE = 0.18;

// var also exists (older syntax) but is generally avoided in modern code
// because of confusing scoping rules compared to let/const.

console.log(age);         // 26
console.log(MAX_RETRIES); // 3
```

1.4 Go

```
package main

import "fmt"

func main() {
    // Variable declared with explicit type
    var age int = 25
    age = 26 // reassignment allowed

    // Shorthand declaration, type is inferred
    name := "Santokh"

    // Constant — value is fixed at compile time
    const maxRetries = 3
    const taxRate = 0.18

    fmt.Println(age, name, maxRetries, taxRate)
}
```

1.5 Kotlin

```
fun main() {
    // var — a mutable variable
    var age: Int = 25
    age = 26 // allowed

    // val — a read-only reference, assigned exactly once
```

```
    val maxRetries = 3
    val taxRate = 0.18

    println("$age $maxRetries $taxRate")
}
```

1.6 Java

```
public class Main {
    public static void main(String[] args) {
        // Regular variable — type must be declared
        int age = 25;
        age = 26; // reassignment allowed

        // Constant — combination of final (cannot be reassigned)
        // and typically static when it belongs to the class rather than an
instance
        final int MAX_RETRIES = 3;
        final double TAX_RATE = 0.18;

        System.out.println(age + " " + MAX_RETRIES + " " + TAX_RATE);
    }
}
```

Chapter 2: Data Types

2.1 The General Idea

A **data type** tells the computer what *kind* of value a variable holds and, therefore, what operations make sense on it. You can add two numbers, but "adding" two pieces of text usually means joining them, and "adding" a number to a boolean doesn't make sense at all. Types prevent whole categories of bugs and let the computer store and process data efficiently.

Common categories across almost every language:

- **Numbers** — often split into integers (whole numbers) and floating-point/decimal numbers.
- **Text** — usually called a string.
- **Boolean** — true or false.
- **Composite types** — arrays/lists, objects/maps, and custom structures built from the basic types.
- **Null / absence of a value** — a special marker meaning "no value here."

Languages differ in *when* they check types. **Statically typed** languages (Go, Kotlin, Java) check types at compile time, before the program ever runs — catching many errors early. **Dynamically typed** languages (PHP, JavaScript) check types at run time, which is more flexible but pushes some errors later into execution.

2.2 PHP

```php
<?php
$count = 10;              // integer
$price = 19.99;           // float
$name = "Santokh";        // string
$isActive = true;         // boolean
$nothing = null;          // null

var_dump($count);   // int(10)
var_dump($price);   // float(19.99)
```

2.3 Node.js (JavaScript)

```javascript
let count = 10;          // number (JS does not separate int/float)
let price = 19.99;       // also just "number"
let name = "Santokh";    // string
let isActive = true;     // boolean
let nothing = null;      // null
```

```
let notDefined;          // undefined

console.log(typeof count, typeof name, typeof isActive);
// "number" "string" "boolean"
```

2.4 Go

```
package main

import "fmt"

func main() {
    var count int = 10        // integer
    var price float64 = 19.99 // floating-point
    var name string = "Santokh"
    var isActive bool = true

    fmt.Println(count, price, name, isActive)
}
```

2.5 Kotlin

```
fun main() {
    val count: Int = 10
    val price: Double = 19.99
    val name: String = "Santokh"
    val isActive: Boolean = true
    val nothing: String? = null // "?" marks it as nullable

    println("$count $price $name $isActive $nothing")
}
```

2.6 Java

```
public class Main {
    public static void main(String[] args) {
        int count = 10;
        double price = 19.99;
        String name = "Santokh";
        boolean isActive = true;
        String nothing = null;

        System.out.println(count + " " + price + " " + name + " " + isActive);
    }
```

```
}
```

Chapter 3: Operators

3.1 The General Idea

Operators are the symbols that act on values: `+`, `-`, `==`, `&&`, and so on. They fall into a few families:

- **Arithmetic** — `+`, `-`, `*`, `/`, `%` (remainder/modulo).
- **Comparison** — `==` (equal), `!=` (not equal), `<`, `>`, `<=`, `>=` — these produce a boolean result.
- **Logical** — `&&` (AND), `||` (OR), `!` (NOT) — used to combine boolean conditions.
- **Assignment** — `=` and its shortcuts like `+=`, `-=`, which update a variable based on its current value.

A subtlety worth knowing: some dynamically typed languages distinguish "loose" equality (which converts types before comparing) from "strict" equality (which does not). JavaScript's `==` vs `===` is the classic example, and using `===` is generally the safer default.

3.2 PHP

```
<?php
$a = 10;
$b = 3;

echo $a + $b;   // 13
echo $a % $b;   // 1 (remainder)
echo $a == "10"; // true (loose comparison, type juggling)
echo $a === "10"; // false (strict comparison, checks type too)
echo ($a > 5 && $b < 5); // true (logical AND)
```

3.3 Node.js (JavaScript)

```
let a = 10;
let b = 3;

console.log(a + b);      // 13
console.log(a % b);      // 1
console.log(a == "10");  // true  — loose equality, converts types
console.log(a === "10"); // false — strict equality, no conversion
console.log(a > 5 && b < 5); // true
```

3.4 Go

```
package main

import "fmt"

func main() {
    a, b := 10, 3

    fmt.Println(a + b)        // 13
    fmt.Println(a % b)        // 1
    fmt.Println(a == 10)      // true — Go has no "loose" equality; types must
match
    fmt.Println(a > 5 && b < 5) // true
}
```

3.5 Kotlin

```
fun main() {
    val a = 10
    val b = 3

    println(a + b)          // 13
    println(a % b)          // 1
    println(a == 10)        // true — structural equality by default
    println(a > 5 && b < 5) // true
}
```

3.6 Java

```
public class Main {
    public static void main(String[] args) {
        int a = 10;
        int b = 3;

        System.out.println(a + b);          // 13
        System.out.println(a % b);          // 1
        System.out.println(a == 10);        // true
        System.out.println(a > 5 && b < 5); // true
    }
}
```

Chapter 4: Conditional Statements

4.1 The General Idea

A conditional statement lets a program take different paths depending on whether some condition is true or false. The most universal form is `if / else if / else`: check a condition, run one block if it's true, optionally check further conditions, and fall back to a default block if nothing matched.

Nearly every language also offers a compact form for simple "pick one of two values" logic — usually called a **ternary operator** — which lets you write a small conditional expression inline rather than as a full block.

Conditionals are how programs encode rules: "if the user is logged in, show the dashboard; otherwise, show the login page." Nesting and combining conditions with logical operators (from Chapter 3) is how more complex rules get built up.

4.2 PHP

```php
<?php
$age = 20;

if ($age < 13) {
    echo "Child";
} elseif ($age < 20) {
    echo "Teenager";
} else {
    echo "Adult";
}

// Ternary shorthand
$status = ($age >= 18) ? "Adult" : "Minor";
echo $status;
```

4.3 Node.js (JavaScript)

```javascript
let age = 20;

if (age < 13) {
  console.log("Child");
} else if (age < 20) {
  console.log("Teenager");
} else {
  console.log("Adult");
}
```

```
// Ternary shorthand
const status = age >= 18 ? "Adult" : "Minor";
console.log(status);
```

4.4 Go

```
package main

import "fmt"

func main() {
    age := 20

    if age < 13 {
        fmt.Println("Child")
    } else if age < 20 {
        fmt.Println("Teenager")
    } else {
        fmt.Println("Adult")
    }

    // Go has no ternary operator by design — an if/else block is used instead
    status := "Minor"
    if age >= 18 {
        status = "Adult"
    }
    fmt.Println(status)
}
```

4.5 Kotlin

```
fun main() {
    val age = 20

    if (age < 13) {
        println("Child")
    } else if (age < 20) {
        println("Teenager")
    } else {
        println("Adult")
    }

    // Kotlin has no separate ternary operator — if/else is itself an
expression
    val status = if (age >= 18) "Adult" else "Minor"
    println(status)
```

```
}
```

4.6 Java

```
public class Main {
    public static void main(String[] args) {
        int age = 20;

        if (age < 13) {
            System.out.println("Child");
        } else if (age < 20) {
            System.out.println("Teenager");
        } else {
            System.out.println("Adult");
        }

        // Ternary shorthand
        String status = (age >= 18) ? "Adult" : "Minor";
        System.out.println(status);
    }
}
```

Chapter 5: Loops

5.1 The General Idea

A loop repeats a block of code, either a fixed number of times, while some condition holds, or once for each item in a collection. Loops exist because computers are good at doing tedious repetitive work, and writing out the same instruction a thousand times by hand would be absurd.

The common families:

- **Counting loop (`for`)** — runs a set number of times, typically driven by a counter variable (`i = 0; i < 10; i++`).
- **Condition loop (`while`)** — checks a condition *before* each pass; keeps going as long as it's true.
- **Post-condition loop (`do...while`)** — like `while`, but the condition is checked *after* the first pass, guaranteeing the body runs at least once.
- **Collection loop (`for...of`, `foreach`, `for...in`, range-based `for`)** — iterates directly over the items in an array, list, or map, without manually managing an index.

Two control keywords show up inside loops in nearly every language: `break` (exit the loop immediately) and `continue` (skip to the next iteration).

5.2 PHP

```
<?php
// Counting for loop
for ($i = 0; $i < 5; $i++) {
    echo $i . " ";
}
echo "\n";

// while loop
$i = 0;
while ($i < 5) {
    echo $i . " ";
    $i++;
}
echo "\n";

// do...while loop — runs at least once
$i = 0;
do {
    echo $i . " ";
    $i++;
```

```
} while ($i < 5);
echo "\n";

// foreach – iterating a collection
$fruits = ["apple", "banana", "cherry"];
foreach ($fruits as $fruit) {
    echo $fruit . " ";
}
```

5.3 Node.js (JavaScript)

```
// Counting for loop
for (let i = 0; i < 5; i++) {
  process.stdout.write(i + " ");
}
console.log();

// while loop
let i = 0;
while (i < 5) {
  process.stdout.write(i + " ");
  i++;
}
console.log();

// do...while loop
i = 0;
do {
  process.stdout.write(i + " ");
  i++;
} while (i < 5);
console.log();

// for...of – iterating a collection
const fruits = ["apple", "banana", "cherry"];
for (const fruit of fruits) {
  process.stdout.write(fruit + " ");
}
```

5.4 Go

```
package main

import "fmt"

func main() {
```

```
    // Go has only one loop keyword: for — but it covers all the classic cases

    // Counting for loop
    for i := 0; i < 5; i++ {
        fmt.Print(i, " ")
    }
    fmt.Println()

    // "while" style — for with just a condition
    i := 0
    for i < 5 {
        fmt.Print(i, " ")
        i++
    }
    fmt.Println()

    // "do...while" style is emulated since Go has no dedicated keyword
    i = 0
    for {
        fmt.Print(i, " ")
        i++
        if i >= 5 {
            break
        }
    }
    fmt.Println()

    // range — iterating a collection
    fruits := []string{"apple", "banana", "cherry"}
    for _, fruit := range fruits {
        fmt.Print(fruit, " ")
    }
}
```

5.5 Kotlin

```
fun main() {
    // Counting loop using a range
    for (i in 0 until 5) {
        print("$i ")
    }
    println()

    // while loop
    var i = 0
    while (i < 5) {
        print("$i ")
        i++
```

```
    }
    println()

    // do...while loop
    i = 0
    do {
        print("$i ")
        i++
    } while (i < 5)
    println()

    // for — iterating a collection directly
    val fruits = listOf("apple", "banana", "cherry")
    for (fruit in fruits) {
        print("$fruit ")
    }
}
```

5.6 Java

```
public class Main {
    public static void main(String[] args) {
        // Counting for loop
        for (int i = 0; i < 5; i++) {
            System.out.print(i + " ");
        }
        System.out.println();

        // while loop
        int i = 0;
        while (i < 5) {
            System.out.print(i + " ");
            i++;
        }
        System.out.println();

        // do...while loop
        i = 0;
        do {
            System.out.print(i + " ");
            i++;
        } while (i < 5);
        System.out.println();

        // enhanced for-each loop — iterating a collection
        String[] fruits = {"apple", "banana", "cherry"};
        for (String fruit : fruits) {
            System.out.print(fruit + " ");
```

```
        }
    }
}
```

Chapter 6: Switch Statements

6.1 The General Idea

A switch statement is a cleaner alternative to a long chain of `if / else if / else` when you're comparing one value against many possible exact matches — for example, checking a day-of-week number, a status code, or a menu choice. Instead of repeating the same variable in condition after condition, you write the value once and list the possible cases beneath it.

Historically, many switch statements required an explicit `break` after each case to stop execution from "falling through" into the next case — a common source of bugs when forgotten. Newer languages either avoid fall-through entirely or offer switch as an *expression* that directly produces a value.

6.2 PHP

```
<?php
$day = 3;

switch ($day) {
    case 1:
        echo "Monday";
        break;
    case 2:
        echo "Tuesday";
        break;
    case 3:
        echo "Wednesday";
        break;
    default:
        echo "Some other day";
}
// Note: without "break", execution falls through into the next case.
```

6.3 Node.js (JavaScript)

```
const day = 3;

switch (day) {
  case 1:
    console.log("Monday");
    break;
  case 2:
    console.log("Tuesday");
```

```
    break;
  case 3:
    console.log("Wednesday");
    break;
  default:
    console.log("Some other day");
}
// Same fall-through behavior as PHP if "break" is omitted.
```

6.4 Go

```
package main

import "fmt"

func main() {
    day := 3

    switch day {
    case 1:
        fmt.Println("Monday")
    case 2:
        fmt.Println("Tuesday")
    case 3:
        fmt.Println("Wednesday")
    default:
        fmt.Println("Some other day")
    }
    // Go does NOT fall through by default — each case breaks automatically.
    // Use the "fallthrough" keyword explicitly if that behavior is wanted.
}
```

6.5 Kotlin

```
fun main() {
    val day = 3

    // Kotlin uses "when" instead of "switch", and it works as an expression
    val name = when (day) {
        1 -> "Monday"
        2 -> "Tuesday"
        3 -> "Wednesday"
        else -> "Some other day"
    }
    println(name)
    // No fall-through at all — each branch is self-contained.
```

```
}
```

6.6 Java

```
public class Main {
    public static void main(String[] args) {
        int day = 3;

        switch (day) {
            case 1:
                System.out.println("Monday");
                break;
            case 2:
                System.out.println("Tuesday");
                break;
            case 3:
                System.out.println("Wednesday");
                break;
            default:
                System.out.println("Some other day");
        }
        // Traditional Java switch falls through without "break",
        // though modern Java (14+) also supports an arrow-based,
        // non-fall-through form: switch (day) { case 3 -> ...; }
    }
}
```

Chapter 7: Arrays and Collections

7.1 The General Idea

An array (or list, or collection) stores multiple values under a single variable name, accessed by position (an index) or, in the case of maps/dictionaries, by a key. Arrays are the backbone of almost every real program — a list of users, a shopping cart's items, a set of configuration options.

Two broad shapes appear across languages:

- **Ordered lists** — values are accessed by numeric position, starting at 0 in most languages.
- **Key-value maps** (also called dictionaries, associative arrays, or objects) — values are accessed by a named key rather than a position.

Statically typed languages usually require every element in an array to be the same type; dynamically typed languages are typically more permissive.

7.2 PHP

```
<?php
// Indexed array
$fruits = ["apple", "banana", "cherry"];
echo $fruits[1]; // banana

// Associative array (key-value map)
$person = ["name" => "Santokh", "age" => 45];
echo $person["name"]; // Santokh
```

7.3 Node.js (JavaScript)

```
// Array
const fruits = ["apple", "banana", "cherry"];
console.log(fruits[1]); // banana

// Object as a key-value map
const person = { name: "Santokh", age: 45 };
console.log(person.name); // Santokh
```

7.4 Go

```
package main
```

```
import "fmt"

func main() {
    // Slice (Go's flexible, growable array type)
    fruits := []string{"apple", "banana", "cherry"}
    fmt.Println(fruits[1]) // banana

    // Map (key-value collection)
    person := map[string]interface{}{"name": "Santokh", "age": 45}
    fmt.Println(person["name"]) // Santokh
}
```

7.5 Kotlin

```
fun main() {
    // List
    val fruits = listOf("apple", "banana", "cherry")
    println(fruits[1]) // banana

    // Map
    val person = mapOf("name" to "Santokh", "age" to 45)
    println(person["name"]) // Santokh
}
```

7.6 Java

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        // Array
        String[] fruits = {"apple", "banana", "cherry"};
        System.out.println(fruits[1]); // banana

        // Map
        Map<String, Object> person = new HashMap<>();
        person.put("name", "Santokh");
        person.put("age", 45);
        System.out.println(person.get("name")); // Santokh
    }
}
```

Chapter 8: Functions

8.1 The General Idea

A function is a named, reusable block of code that performs a task, optionally accepts input values (**parameters**), and optionally sends back a result (**return value**). Functions exist so that logic is written once and reused everywhere it's needed, rather than copy-pasted — which makes programs shorter, easier to test, and easier to fix (one bug fix in one place, instead of a dozen scattered fixes).

A function's **signature** — its name, the parameters it takes, and what it returns — acts as a contract: "give me these inputs, and I promise to hand back this kind of output." Statically typed languages enforce that contract at compile time; dynamically typed languages check it, if at all, only when the function actually runs.

8.2 PHP

```
<?php
function greet($name) {
    return "Hello, " . $name . "!";
}

echo greet("Santokh"); // Hello, Santokh!
```

8.3 Node.js (JavaScript)

```
function greet(name) {
  return `Hello, ${name}!`;
}

console.log(greet("Santokh")); // Hello, Santokh!

// Arrow function shorthand, common in modern JS
const greetArrow = (name) => `Hello, ${name}!`;
```

8.4 Go

```
package main

import "fmt"

// Go requires explicit parameter and return types
func greet(name string) string {
    return "Hello, " + name + "!"
```

```
}

func main() {
    fmt.Println(greet("Santokh")) // Hello, Santokh!
}
```

8.5 Kotlin

```
fun greet(name: String): String {
    return "Hello, $name!"
}

fun main() {
    println(greet("Santokh")) // Hello, Santokh!
}
```

8.6 Java

```
public class Main {
    static String greet(String name) {
        return "Hello, " + name + "!";
    }

    public static void main(String[] args) {
        System.out.println(greet("Santokh")); // Hello, Santokh!
    }
}
```

Chapter 9: Comments and Code Documentation

9.1 The General Idea

Comments are text in the source code that the computer ignores but that humans read — explanations of *why* code does something, notes for future maintainers (often yourself, six months later), or temporary annotations while debugging. Comments don't change what a program does; they change how easily a person can understand it.

Nearly every language supports two comment styles:

- **Single-line comments** — everything after a marker to the end of the line is ignored.
- **Multi-line / block comments** — everything between an opening and closing marker is ignored, useful for longer explanations or temporarily disabling a chunk of code.

Many languages also support a more structured "documentation comment" format (like PHP's PHPDoc, Java's Javadoc, or Kotlin's KDoc) that tools can read to auto-generate reference documentation.

9.2 PHP

```
<?php
// This is a single-line comment

/*
 * This is a
 * multi-line comment
 */

/**
 * A documentation comment (PHPDoc), often used to describe functions
 * @param string $name
 * @return string
 */
function greet($name) {
    return "Hello, $name!";
}
```

9.3 Node.js (JavaScript)

```
// This is a single-line comment
```

```
/*
 * This is a
 * multi-line comment
 */

/**
 * A JSDoc comment, describing the function for tooling and editors
 * @param {string} name
 * @returns {string}
 */
function greet(name) {
  return `Hello, ${name}!`;
}
```

9.4 Go

```
// This is a single-line comment

/*
   This is a
   multi-line comment
*/

// greet returns a friendly message for the given name.
// Go convention: doc comments start with the function's name.
func greet(name string) string {
    return "Hello, " + name + "!"
}
```

9.5 Kotlin

```
// This is a single-line comment

/*
 * This is a
 * multi-line comment
 */

/**
 * A KDoc comment, describing the function for tooling and editors
 * @param name the name to greet
 * @return a greeting message
 */
fun greet(name: String): String {
    return "Hello, $name!"
}
```

9.6 Java

```
// This is a single-line comment

/*
 * This is a
 * multi-line comment
 */

/**
 * A Javadoc comment, describing the method for tooling and generated docs
 * @param name the name to greet
 * @return a greeting message
 */
static String greet(String name) {
    return "Hello, " + name + "!";
}
```

Chapter 10: Classes and Objects (OOP Basics)

10.1 The General Idea

Object-oriented programming groups related data and the functions that act on it into a single unit called a **class**. A class is a blueprint — it doesn't hold real data itself, it just describes what shape an object of that type will have. An **object** (or **instance**) is a concrete thing built from that blueprint, with its own copy of the data.

A class typically defines:

- **Properties / fields** — the data each object carries (a `Person` has a `name` and an `age`).
- **Methods** — functions that belong to the class and usually act on its own properties (a `Person` can `greet()`).
- **Constructor** — a special method that runs when a new object is created, used to set up its initial data.

This matters because it lets code mirror the real-world things it models: instead of separate loose variables and functions passed around everywhere, a `Person` object bundles a person's data and behavior together, and you can create as many independent `Person` objects as you need.

Not every language expresses this the same way. Go, notably, has no classes at all — it uses **structs** (plain data groupings) combined with functions that are explicitly attached to that struct type, achieving much the same effect without traditional class syntax or inheritance.

10.2 PHP

```
<?php
class Person {
    public string $name;
    public int $age;

    // Constructor — runs automatically when a new object is created
    public function __construct(string $name, int $age) {
        $this->name = $name;
        $this->age = $age;
    }

    public function greet(): string {
        return "Hi, I'm {$this->name} and I'm {$this->age} years old.";
    }
}
```

```
$person = new Person("Santokh", 45);
echo $person->greet();
```

10.3 Node.js (JavaScript)

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  greet() {
    return `Hi, I'm ${this.name} and I'm ${this.age} years old.`;
  }
}

const person = new Person("Santokh", 45);
console.log(person.greet());
```

10.4 Go

```
package main

import "fmt"

// Go has no classes — a struct holds the data
type Person struct {
    Name string
    Age  int
}

// A method is a function with a "receiver", attaching it to the struct type
func (p Person) Greet() string {
    return fmt.Sprintf("Hi, I'm %s and I'm %d years old.", p.Name, p.Age)
}

func main() {
    person := Person{Name: "Santokh", Age: 45}
    fmt.Println(person.Greet())
}
```

10.5 Kotlin

```
class Person(val name: String, val age: Int) {
    fun greet(): String {
        return "Hi, I'm $name and I'm $age years old."
    }
}

fun main() {
    val person = Person("Santokh", 45)
    println(person.greet())
}
```

10.6 Java

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String greet() {
        return "Hi, I'm " + name + " and I'm " + age + " years old.";
    }

    public static void main(String[] args) {
        Person person = new Person("Santokh", 45);
        System.out.println(person.greet());
    }
}
```

Chapter 11: Error Handling

11.1 The General Idea

Things go wrong at run time no matter how careful you are: a file might not exist, a network call might time out, a user might type text where a number was expected. Error handling is the set of tools a language gives you to detect these situations and respond deliberately, instead of letting the program crash or silently produce a wrong result.

Two broad philosophies exist:

- **Exceptions** — when something goes wrong, the program "throws" an error object, which immediately stops normal execution and jumps to the nearest matching `catch` block that knows how to handle it. This is the approach PHP, JavaScript, Kotlin, and Java all take.
- **Explicit error values** — the function simply returns an error alongside (or instead of) its normal result, and the caller is expected to check it. Go deliberately avoids exceptions in favor of this style, treating error-checking as an ordinary, visible part of the code rather than a hidden jump.

Most exception-based languages also support a `finally` block — code that runs whether or not an error occurred, typically used for cleanup like closing a file or a database connection.

11.2 PHP

```
<?php
function divide($a, $b) {
    if ($b === 0) {
        throw new Exception("Cannot divide by zero");
    }
    return $a / $b;
}

try {
    echo divide(10, 0);
} catch (Exception $e) {
    echo "Error: " . $e->getMessage();
} finally {
    echo "\nDivision attempt finished.";
}
```

11.3 Node.js (JavaScript)

```
function divide(a, b) {
  if (b === 0) {
    throw new Error("Cannot divide by zero");
  }
  return a / b;
}

try {
  console.log(divide(10, 0));
} catch (error) {
  console.log("Error: " + error.message);
} finally {
  console.log("Division attempt finished.");
}
```

11.4 Go

```
package main

import (
    "errors"
    "fmt"
)

// Go returns errors as ordinary values, typically as the last return value
func divide(a, b float64) (float64, error) {
    if b == 0 {
        return 0, errors.New("cannot divide by zero")
    }
    return a / b, nil
}

func main() {
    result, err := divide(10, 0)
    if err != nil {
        fmt.Println("Error:", err)
    } else {
        fmt.Println(result)
    }
    fmt.Println("Division attempt finished.")
}
```

11.5 Kotlin

```
class DivisionException(message: String) : Exception(message)
```

```
fun divide(a: Double, b: Double): Double {
    if (b == 0.0) {
        throw DivisionException("Cannot divide by zero")
    }
    return a / b
}

fun main() {
    try {
        println(divide(10.0, 0.0))
    } catch (e: DivisionException) {
        println("Error: ${e.message}")
    } finally {
        println("Division attempt finished.")
    }
}
```

11.6 Java

```
public class Main {
    static double divide(double a, double b) {
        if (b == 0) {
            throw new ArithmeticException("Cannot divide by zero");
        }
        return a / b;
    }

    public static void main(String[] args) {
        try {
            System.out.println(divide(10, 0));
        } catch (ArithmeticException e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            System.out.println("Division attempt finished.");
        }
    }
}
```

Chapter 12: Asynchronous Code

12.1 The General Idea

Most code runs **synchronously** — one line finishes before the next one starts. That works fine for pure computation, but it breaks down for anything that involves waiting: reading a file from disk, calling a remote API, querying a database. If the program simply froze on every one of those waits, it would feel sluggish and would waste the time it could have spent on other work.

Asynchronous code is a way of saying: "start this slow operation, but don't block everything else while it finishes — let me know when the result is ready." Three patterns have evolved to manage this, largely in this historical order:

- **Callbacks** — pass a function to be called once the slow operation completes. Simple, but can become hard to read when many async steps are chained ("callback hell").
- **Promises / Futures** — an object representing a value that will exist *eventually*, with methods to react once it resolves or fails. This is cleaner to chain than raw callbacks.
- **async/await** — syntax that lets asynchronous code be *written* like synchronous code (top to bottom, no nested callbacks) while still running asynchronously underneath. This is the modern, preferred style in most languages that support it.

Go takes a distinct approach: rather than async/await, it uses lightweight **goroutines** (concurrent functions) and **channels** (pipes for passing data between them), reflecting a concurrency model built around communicating processes rather than promises.

12.2 PHP

```
<?php
// PHP executes synchronously by default; there is no built-in Promise/await.
// The concept is still shown here using a callback, PHP's native mechanism
// for "run this once the other work is done".
function fetchData($callback) {
    // Imagine this represents a slow operation, like a network call
    $data = "user data";
    $callback($data);
}

fetchData(function ($result) {
    echo "Received: " . $result;
});
```

```
// True non-blocking async in PHP typically requires an extension
// or framework such as ReactPHP, Swoole, or Amp.
```

12.3 Node.js (JavaScript)

```
// Node.js is built around asynchronous I/O from the ground up.

function fetchData() {
  return new Promise((resolve) => {
    setTimeout(() => resolve("user data"), 1000); // simulates a slow call
  });
}

// Modern style: async/await, reads top-to-bottom like synchronous code
async function main() {
  console.log("Fetching...");
  const result = await fetchData();
  console.log("Received:", result);
}

main();
```

12.4 Go

```
package main

import (
    "fmt"
    "time"
)

// Go uses goroutines (lightweight concurrent functions) and channels
// instead of async/await.
func fetchData(resultChan chan string) {
    time.Sleep(1 * time.Second) // simulates a slow call
    resultChan <- "user data"
}

func main() {
    resultChan := make(chan string)
    go fetchData(resultChan) // "go" starts this function concurrently

    fmt.Println("Fetching...")
    result := <-resultChan // waits here for the value to arrive
    fmt.Println("Received:", result)
}
```

12.5 Kotlin

```
import kotlinx.coroutines.*

// Kotlin uses coroutines: "suspend" functions that can pause without
// blocking the underlying thread.
suspend fun fetchData(): String {
    delay(1000) // simulates a slow call, non-blocking
    return "user data"
}

fun main() = runBlocking {
    println("Fetching...")
    val result = fetchData()
    println("Received: $result")
}
```

12.6 Java

```
import java.util.concurrent.CompletableFuture;

public class Main {
    static CompletableFuture<String> fetchData() {
        return CompletableFuture.supplyAsync(() -> {
            try {
                Thread.sleep(1000); // simulates a slow call
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
            return "user data";
        });
    }

    public static void main(String[] args) throws Exception {
        System.out.println("Fetching...");
        String result = fetchData().get(); // waits for the result
        System.out.println("Received: " + result);
    }
}
```

Closing Notes

Across five very different languages — a loosely typed web scripting language, a runtime built for JavaScript, a compiled systems language, a modern JVM language, and the long-standing enterprise standard — the same handful of ideas kept reappearing: name a piece of data, decide between paths, repeat work, and organize logic into functions. The syntax around those ideas differs — curly braces versus indentation, `var` versus `val` versus `$`, fall-through switches versus `when` expressions — but the underlying thinking is portable.

The practical implication: once these fundamentals are solid in one language, picking up a sixth, seventh, or eighth language is far more about learning new syntax and new library conventions than about learning to reason like a programmer all over again. That reasoning — breaking a problem into stored values, decisions, repetition, and reusable pieces — is the actual skill, and it's the same skill in every language in this book.

About the Author

Santokh Singh Saggu is a founder of Riasys Technology and a technologist who works at the intersection of thinking and making. He builds software, applications, and electronic systems by blending the technology of the mind—design, psychology, and philosophy—with the technology of matter—computers, programming languages, and machines.

He enjoys reading and studying books across multidisciplinary fields and learning from people and nature. Through this, he seeks knowledge, wisdom, insights, and patterns, which he analyzes and synthesizes into writing, design, and practical systems.

He believes most challenges are not purely technical but stem from confusion and lack of clarity. By focusing on simplicity and structure, he aims to make work and learning more manageable.

His work supports entrepreneurs and professionals in solving real business problems, and helps learners move from uncertainty to understanding in areas such as programming and electronic device design.

Visit : https://www.riasys.co.in