Essential Concepts of Computer Programming

A beginner-friendly guide to essential concepts in computer programming to design more robust, extensible, reusable, and efficient software.

alpharithms

Zack West

αlphαrithms

www.alpharithms.com

Table of Contents

Introduction

This short book is intended to serve as a quick reference and jumping off point for the most basic concepts in Computer Science, Programming, and Software Engineering. It is—by no means—mean to serve as a comprehensive survey, authoritative reference, or even something you would likely want to take very seriously—*seriously*.

This book is a "living document" in that it is constantly being updated, tweaked, and revised. The most current version can be downloaded freely from the following URL:

https://www.alpharithms.com/get/basic-programming-concepts/

For convenience, there is also a web-version of this book that will generally reflect the content here. This version is freely available on the alpharithms.com website at the following URL:

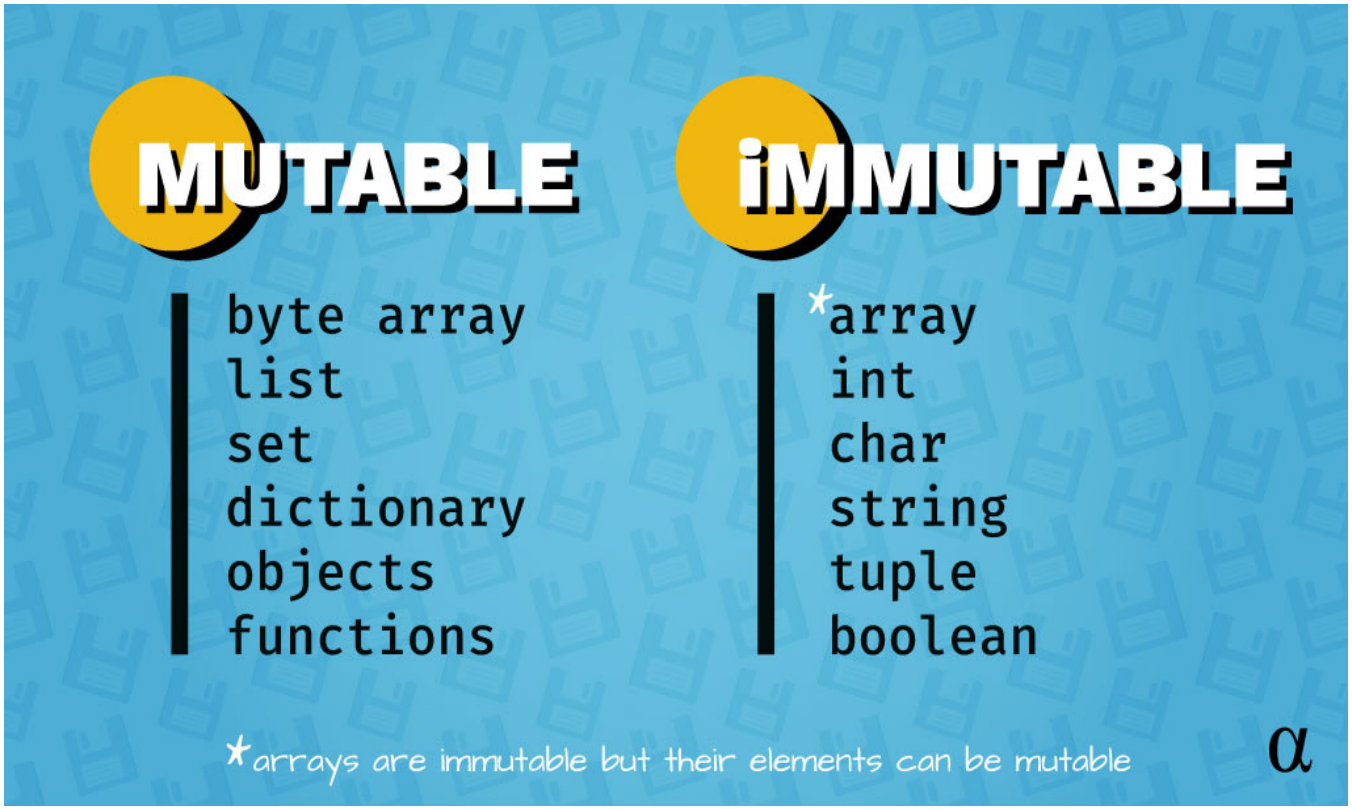
https://www.alpharithms.com/concepts-every-programmer-should-know-480916/

A few final thoughts before diving in: readers will benefit from taking a moment to acknowledge both the breadth and depth of learning opportunities within the fields of Computer Science, Programming, and Software Engineering.

Each field consists of functionally limitless numbers of sub-topics that one could devote a lifetime of study to—and still have work leftover. Seeking to "learn it all" is a perspective almost certain to drive one mad. A steady, appreciative, and mindful approach is likeliest to sustain the life-long love of learning that many successful computing professional possess.

That's it—enjoy the book!

Mutability



Mutability describes the ability of an object to be changed after its initial creation. Mutable objects allow such change while immutable objects can not be changed. These properties can be used to bound the behavior of programs by defining how the state of certain data is allowed to be altered. In some cases, mutability can restrict data from being changed at all!

Mutable Objects

Mutable objects are characterized by the inability to change their data. This property can be a direct result of a Data Type or an indirect result of a Data Structure. For example, an Array is immutable in the sense that its length is fixed in size but also immutable in that its elements can be replaced or, in some cases, even mutated. Here is a list of common mutable objects:

- ArrayLists (java)
- Byte Arrays
- Sets
- Hash Tables
- Dictionaries (Python)

Immutable Objects

Immutable objects are characterized by their inability to be changed once they have been created. These objects simplify many aspects of programming, compiling, and run-time goals. They are thread-safe, won't change state during thrown errors, don't need copy constructors, and serve as overtly predictable keys for Data Structures like hash tables, sets, and etc. Some common immutable data types are as follows:

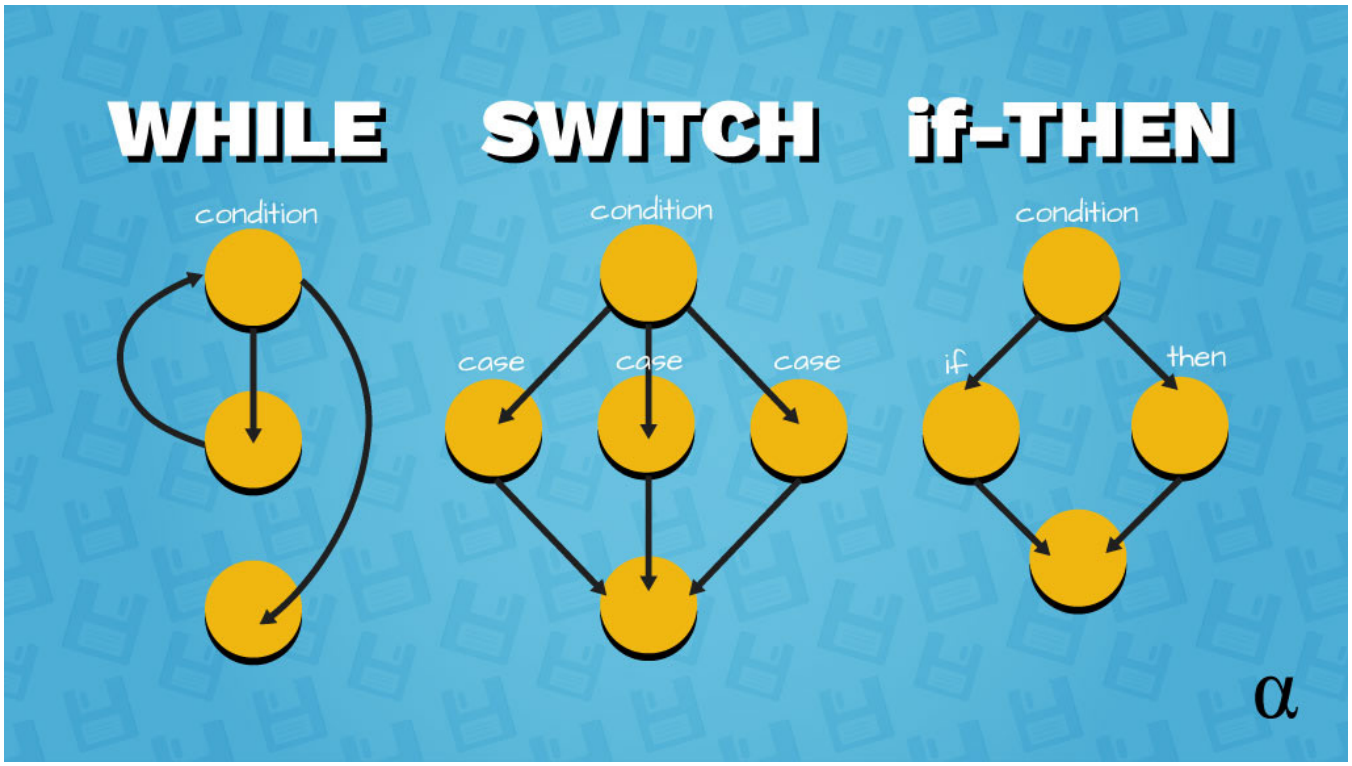
- Integers
- Characters
- Strings
- Tuples
- Floats

> Note: *The keywords final and const are often confused with mutability. These declarations define the behavior of references to objects but not an object's mutability without additional consideration.*

Recommended Course: *Java In-Depth: Become a Complete Java Engineer*

Recommended Reading: *The Art of Immutable Architecture: Theory and Practice of Data Management in Distributed Systems*

Control Flow



Control flow describes the order in which programming statements are evaluated. Control flow mechanisms exist from the lowest levels of CPU architecture in physical form to the highest levels of abstraction in computer language syntaxes. Examples of control flow mechanisms include the following:

- If-Then Statements
- Switch Cases
- Condition-controlled Loops (Do While, Break, Continue, etc.)
- Assertions
- Coroutines

Each of these represents logical tools by which programmers can control the flow of their programs' execution and/or interpretation. The canonical control flow example is seen with If-Then statements where a program checks if a condition is present and then acts accordingly.

For example, if a car is turned on then one can start driving. Control flow statements often present alternatives should an if condition not be met. Using our car example, after the if-then there might be an else statement that says: else turn the car on. Here's another example, written in Python, showing a series of control flow statements:

```
# Initial “if” statement checks if
# if the car is started.
if car_started:

    # If the “is_started” condition is True
    # (i.e. if the car is started) the following
    # code will be executed:
    start_driving()

# The “else” statement provides fallback
# logic should the condition not be met
else:

    # If the “is_started” condition is False
    # (i.e. the car is NOT started) the
    # Following code will be executed
    start_car()
```

If-Then statements aren’t the only means of control flow though certainly the most commonly recognized. A close alternative is the Switch statements where a variable is considered and multiple Case statements are provided to address each possible state. Here’s an example of a basic Switch statement written in Java:

```
public class ControlFlow {
    public static void main(String[] args) {
        String weekdayName = args[0];
        switch (weekdayName) {
            case “Monday”:
                System.out.println(“The First Day of the Week!”);
            case “Tuesday”:
                System.out.println(“The Second Day of the Week!”);
            case “Wednesday”:
                System.out.println(“The Third Day of the Week”);
            case “Thursday”:
                System.out.println(“The Fourth Day of the Week”);
            case “Friday”:
                System.out.println(“The Fifth Day of the Week”);
            case “Saturday”:
            case “Sunday”:
                System.out.println(“It’s the freakin’ Weekend!”);
            default:
                System.out.println(“That’s not a weekday name!”);
        }
    }
}
```

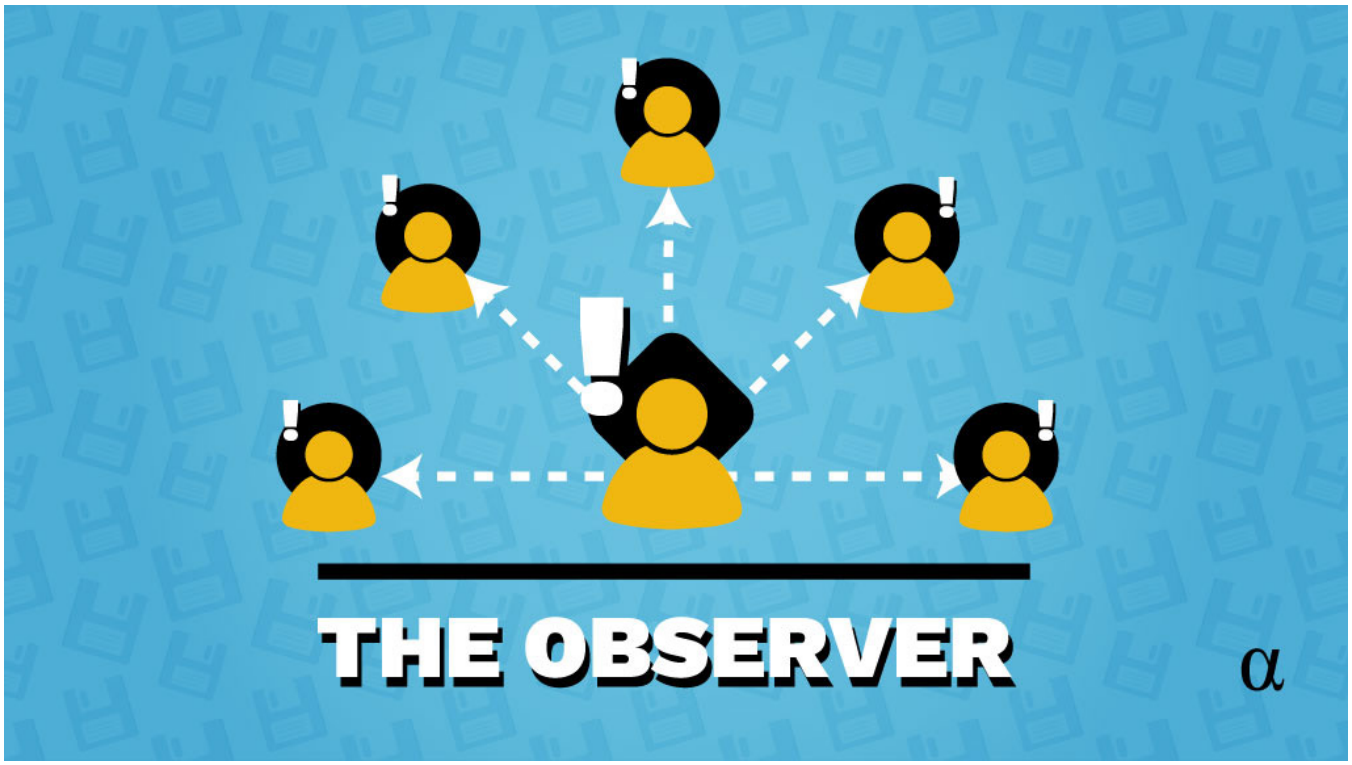
```
// Example Usage
java ControlFlow Fribsday

// Resulting output
That's not a weekday name!
```

Here we see a series of case statements that test the variable `weekdayName` and act accordingly. Note the "stacking" of case statements for which the same logic is applied (Saturday and Sunday) and also the ending default case that will catch input not addressed by other statements.

> **Note**: *Control Flow is not to be confused with Flow Control —another useful concept from the field of computer networking.*

Recommended Course: *Flow Controls – Fundamentals of Programming in Python*
Recommended Reading: *Effective Java – 3rd Edition*

Design Patterns



Design patterns are generalized, reusable approaches to solving common problems. There is an incredibly vast number of design patterns used across an incredibly vast number of software contexts.

The use of design patterns allows programmers and engineers to benefit from the collective work of others in solving common problems. Several popular and broadly applicable design patterns are as follows:

- Observer Pattern
- Factory Pattern
- Singleton Pattern
- Decorator

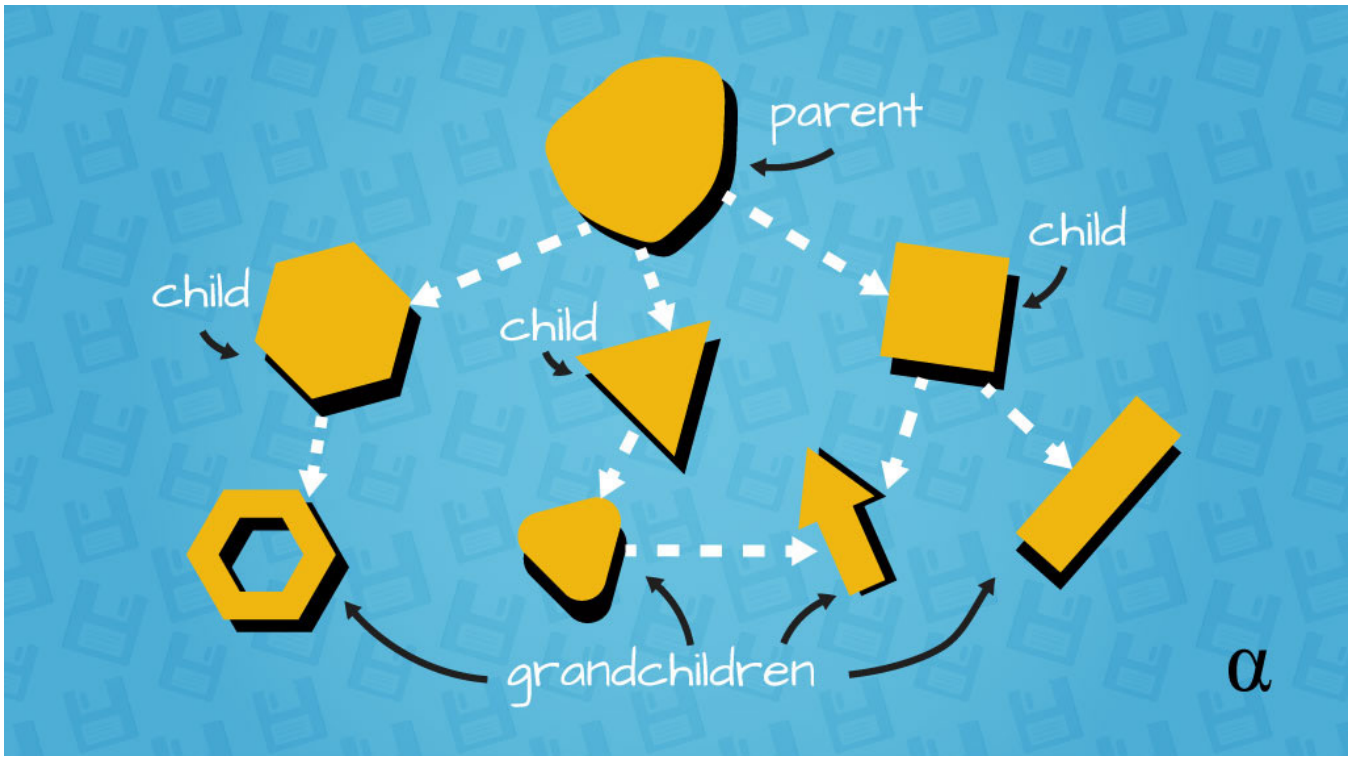
Design patterns can be created, adapted, or re-used to address common problems in many situations. Some are nuanced—only applicable maybe to a few common problems a single design team faces for a specific project. Others are near-universal—patterns that stand to benefit any team working on any software. Regardless of scope, design patterns fall into one of these four broad categories:

- Creational
- Structural
- Behavioral
- Concurrency

The origin of design patterns is credited to the book *Design Patterns Elements of Reusable Object-Oriented Software* published in 1994 by the now-infamous "Gang of Four." Design patterns aren't requirements of effective programming but almost certainly requirements of efficient programming efficiency.

Recommended Reading: *Head First Design Patterns: A Brain-Friendly Guide*
Recommended Course: *Design Patterns in Java*

Object-Oriented Design



Object-Oriented Programming (OOP) accounts for many core features of modern programming languages. It can be characterized as a programming paradigm, often contrasted with functional programming, where "objects" model real-world concepts.

OOP objects contain data and operations with which programmers can manipulate the data by changing the state of an object. OOP is a vast topic and one many have dedicated the focus of their academic and professional careers to studying. Important concepts of OOP include the following:

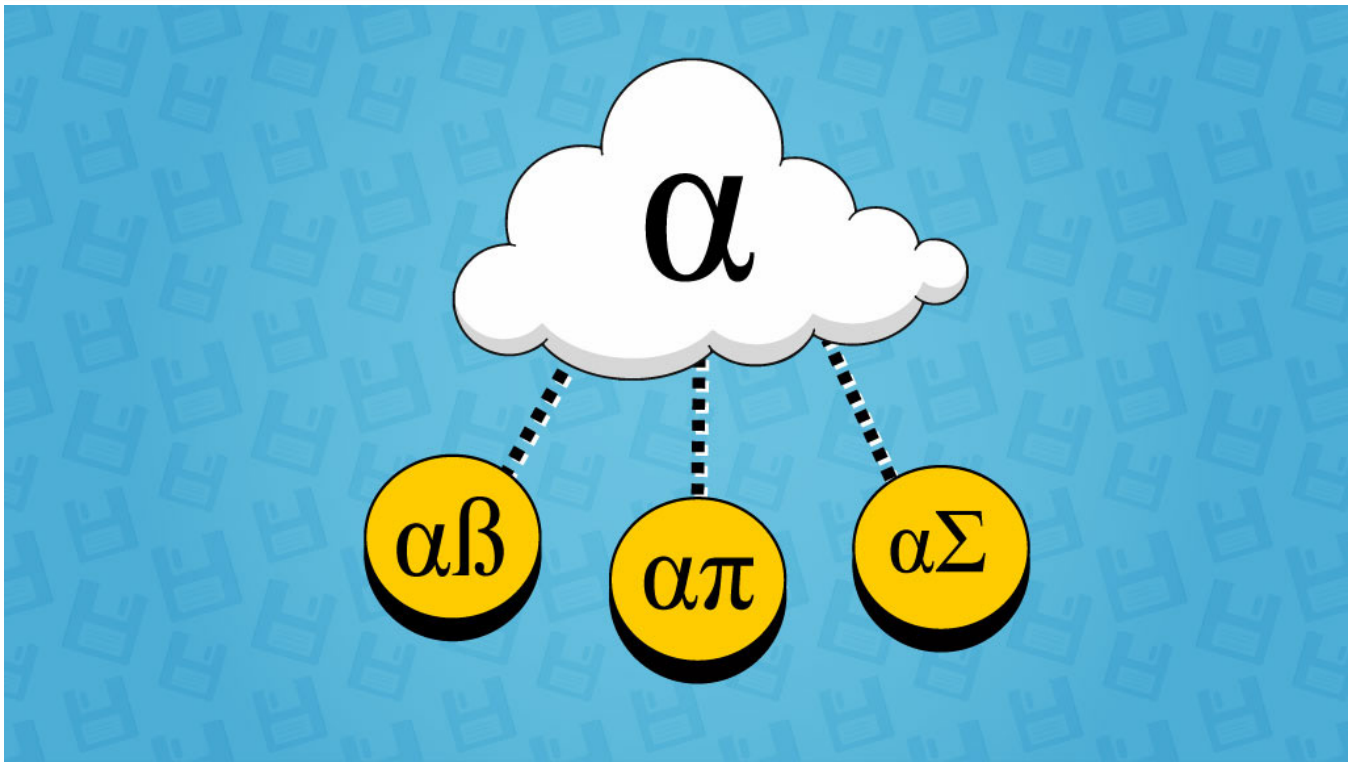
- Inheritance
- Polymorphism
- Abstract Classes
- Encapsulation
- Composition

Examples of primarily object-oriented programming languages include Java, C#, and C++ though other modern languages such as Python, JavaScript, and PHP all use object-oriented designs as well and, in most cases, are designed in an OOP fashion but do not require programmers to constrain themselves to the OOP paradigm.

Note: Python is an Object-Oriented language in how it is designed but allows programmers to use many other paradigms like functional programming. Languages like C# and Java are much more opinionated in their class-based OOP design and require an OOP-first approach.

Recommended Reading: *Object-Oriented Thought Process*
Recommended Course: *Java Programming for Complete Beginners*

Abstract Data Types & Data Structures



Abstract Data Types (ADTs) are high-level concepts that outline how data can be stored and manipulated. They provide the conceptual framework for what tasks should be addressed by programmers but offer little to no opinion on how those tasks should be addressed. The following are common ADTs:

- Array
- Linked List
- Hash Map
- Queue
- Tree

ADTs are outlined in formal syntax through a series of algebraic declarations including a **signature** and **axioms** outlining the operations by which the ADT performs the intended actions. These operations fall into one of several common categories:

- Constructor
- Transformer
- Observer
- Destructor
- Iterator

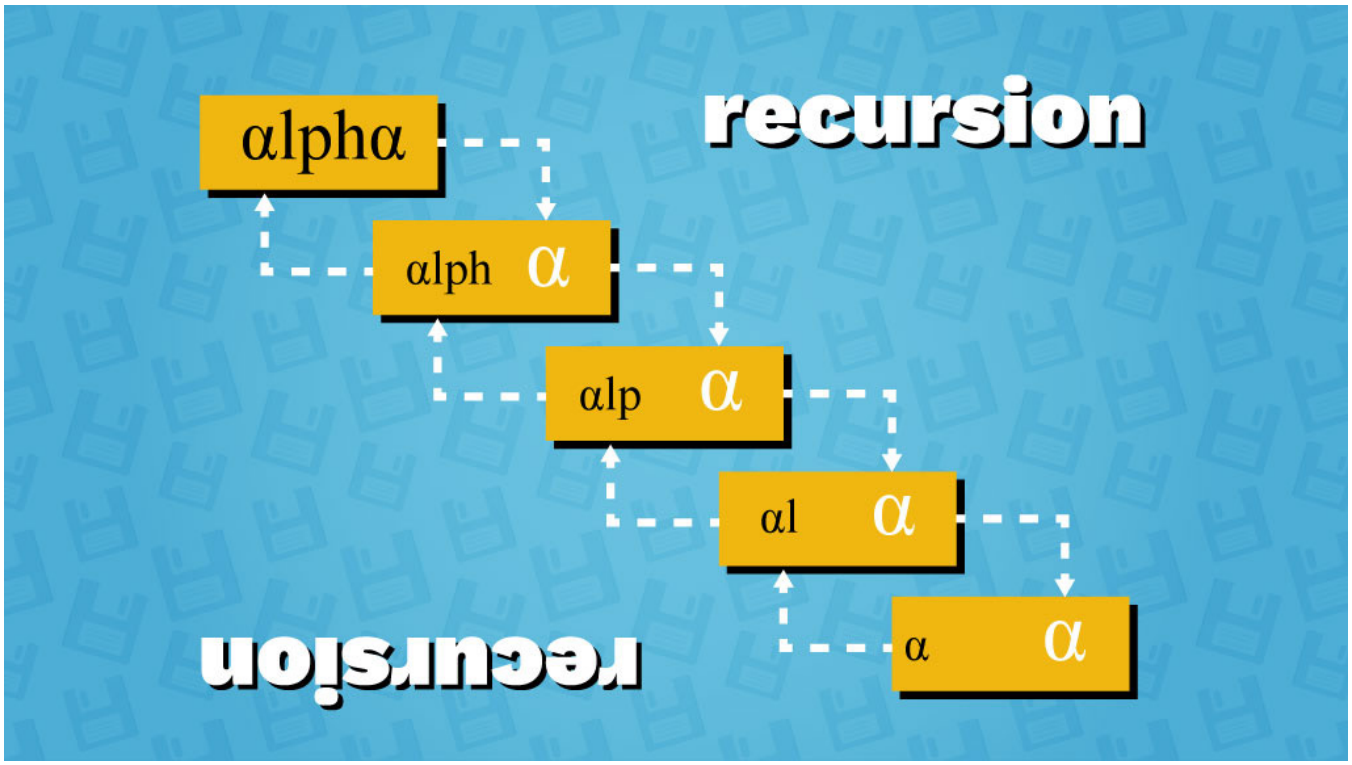
ADTs are, by definition, abstract in the sense they don't exist by rite. Rather, ADTs are implemented as Data Structures with language-specific and use-case-specific details. For example, the `Queue` ADT can be in Python, Java, C#, or any other language—but is then referred to as a **Data Structure**.

Data Structures reflect the theoretical underpinnings of ADTs but implement the specifics of their operations. These objects handle data in ways designed to provide optimal performance under certain conditions. These conditions may be a programming language, a processing load, a backlog of data, or any number of other foreseen and unforeseen conditions.

Data Structures are often taught alongside algorithmic design and analysis in many Computer Science programs. Understanding how the design and use of certain data structures impact the efficiency of algorithmic processes is essential. Most modern programming languages afford developers many data structures to either use explicitly or to adapt for specific use cases.

Recommended Reading: *Algorithms*
Recommended Course: *Data Structures and Algorithms: Deep Dive Using Java*

Recursion



Recursion is an eloquent concept by which functions and procedures tackle a problem by calling themselves repeatedly until a certain condition is met. Such functions and procedures are considered to be recursive and can be used to tackle problems such as navigating hierarchical structures like Trees. Recursive functions can also cause disaster when misused.

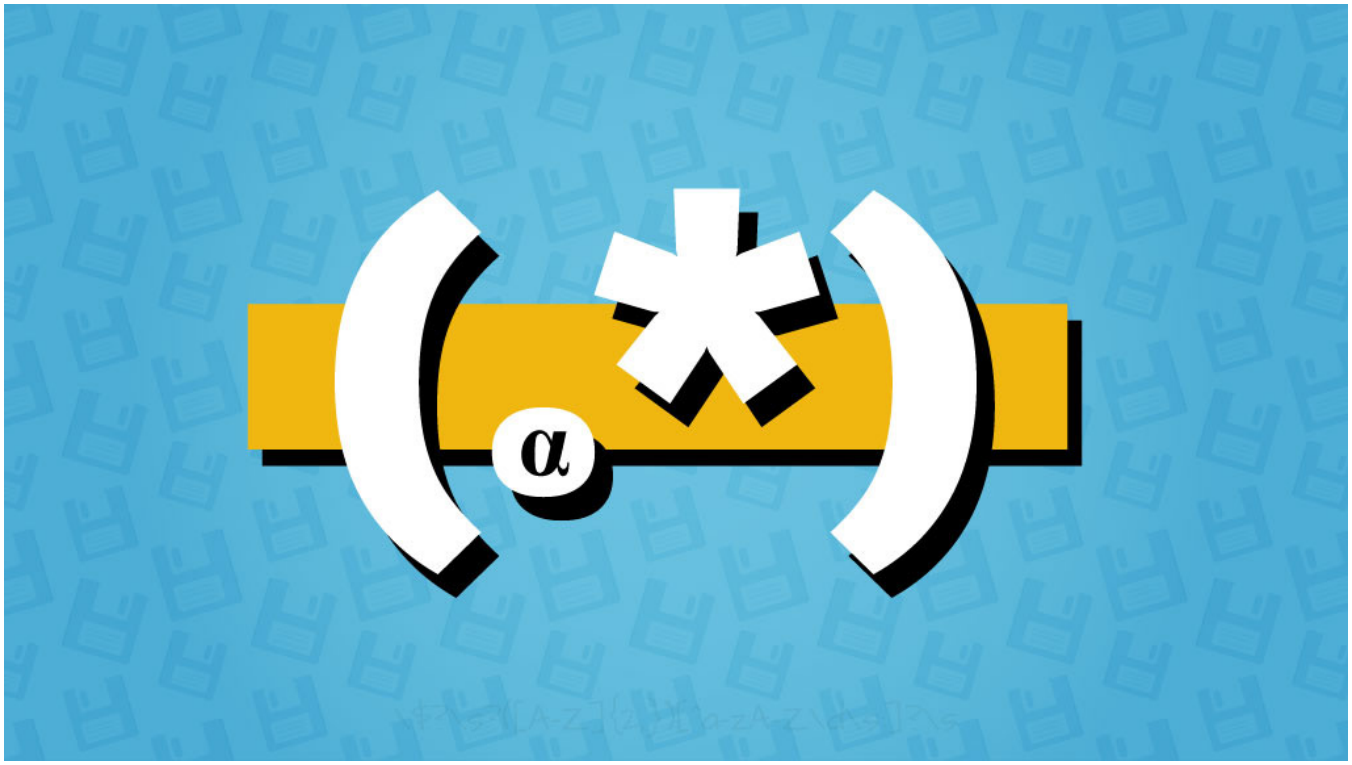
A function or procedure can be considered recursive if it has the following characteristics:

01. A base case, often a conditional statement, by which the flow of the logic of the function can declare the procedure complete;
02. A call to itself when the condition of the base statement is not met;

These two statements are considered the theoretical basis of recursive functions. A third condition is also present for any decent recursive function, however: the simplification of the problem at hand. Without a simplification—progress towards producing a meaningful return value—a recursive function will never meet its base case.

Recommended Reading: *The Little Schemer*
Recommended Course: *Recursion, Backtracking, and Dynamic Programming in Python*

Regular Expressions



Regular expressions are used to identify subsets of text from larger textual data. They help software validate emails, flag offensive content, and can help tidy up databases in a cinch. They can also cause disastrous results when used improperly.

Regular expressions evolved from the early days of Computer Science when the formal definitions of what constituted a regular language were being defined. Concepts like **context-free grammar**, **finite state machines**, and **deterministic finite automatons (DFA)** all played a role in the history of regular expressions.

It has been said that, in solving a problem with regular expressions, one has immediately created a second problem. This statement illustrates the complexity by which many regular expressions can often embody (often by accident.)

Regular expressions (a.k.a. *regex*) are available in many formats in modern programming languages. Most modern implementations reflect the Perl-style regular expressions and can trace their power to the Perl Compatible Regular Expressions (PCRE) library written in C.

Recommended Book: *Regular Expressions Cookbook – O'Reilly*
Recommended Course: *The Complete Regular Expressions (Regex) Course for Beginners*

Software Testing

Testing is an essential aspect of both the development and maintenance of software. There are many types of testing, each addressing specific concerns of a software's operability. There is even a programming paradigm known as Test-Based Development by which developers write tests for the software before they write the actual program. Common types of software testing include the following:

- Unit Testing
- Integration Testing
- Functional Testing
- End-to-End Testing
- Acceptance Testing
- Performance Testing

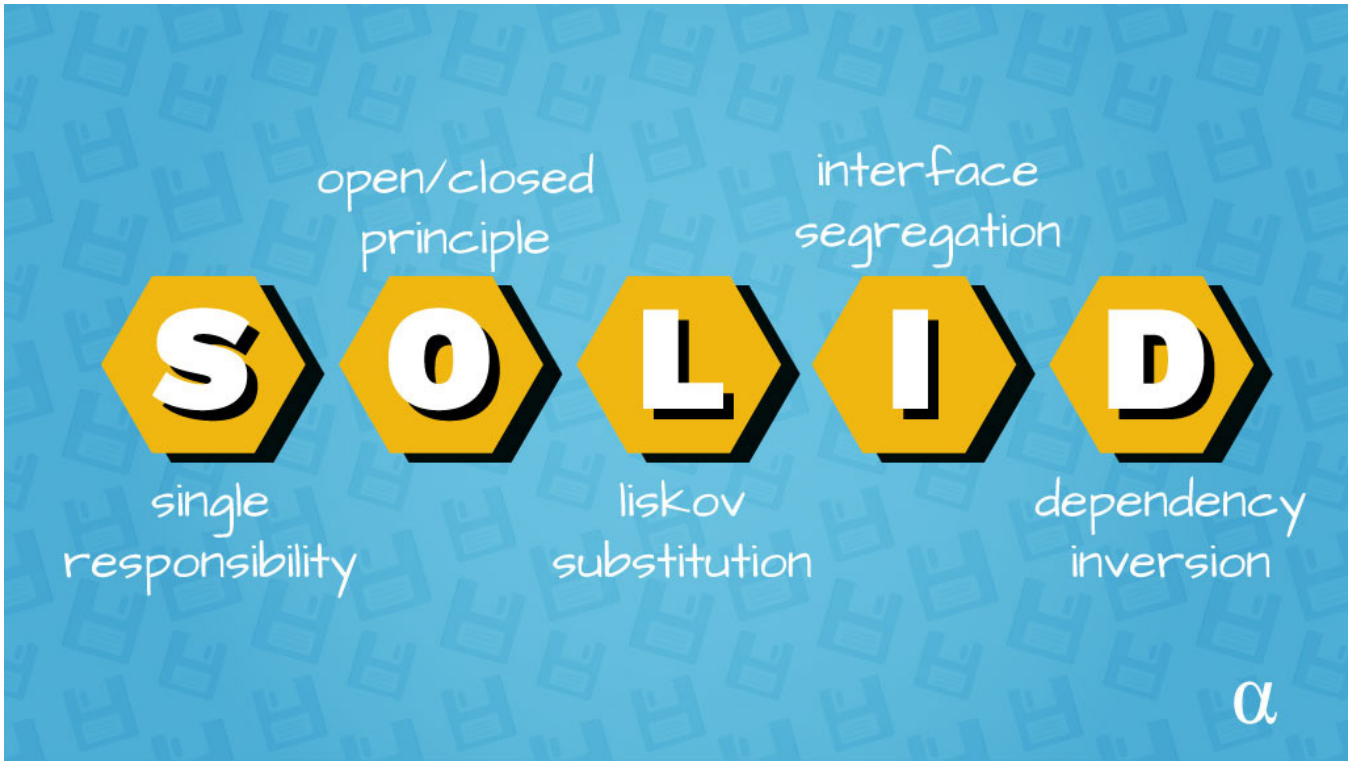
Each of these forms of testing is suited for specific stages during the software development process. For example, Unit Testing is used to test individual functions and methods whereas acceptance testing is more conceptual and can be used to verify a piece of software meets the requirements outlined in contracts with clients.

Modern programming languages almost always come with language-specific testing frameworks. For example, Python's `unittest` framework is part of its standard library. This module provides

functions and classes developers can use to design and automate the process of unit testing their Python code. Other languages like Java require third-party libraries like `JUnit` to provide the same functionality.

Recommended Reading: *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*
Recommended Course: *Unit Testing and Test Driven Development in Python*

S.O.L.I.D. Principles



The *S.O.L.I.D. principles of software development* outline a series of best practices in the field of software engineering. These include guidance for creating efficient, reusable, and interoperable code that can be easily maintained. Below is a brief summary of each primary principle of the S.O.L.I.D. framework:

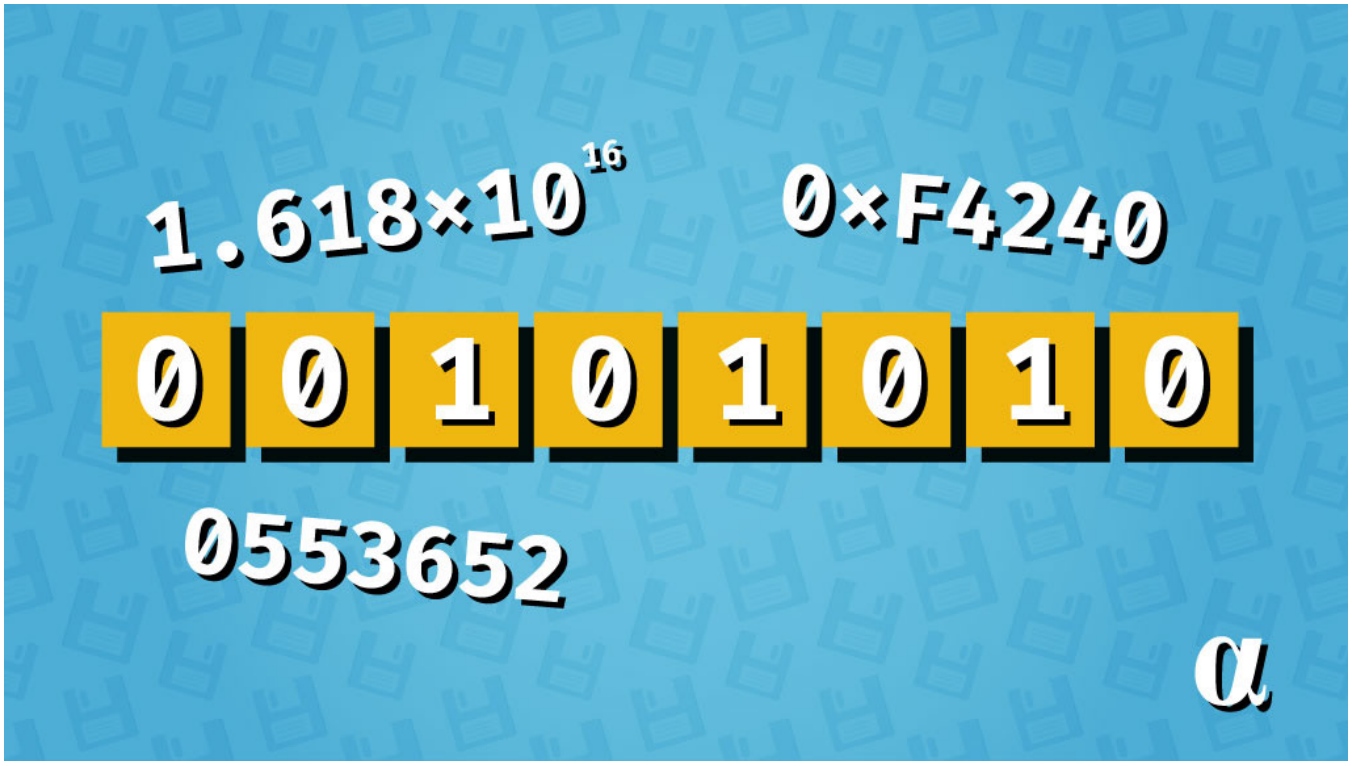
- **S** – Single Responsibility
- **O** – Open/Closed
- **L** – Liskov Substitution
- **I** – Interface Segregation
- **D** – Dependency Inversion

The principles have evolved alongside the field of software development over the years. Their introduction, adoption, and successive evolution all address common problems in software design that can cause issues during the software development life cycle.

The S.O.L.I.D. principles help to design modular packages, decouple essential functionalities, design for high degrees of interoperability, and leave things tidy for the next developer.

Recommended Reading: *SOLID: Guidelines for Better Software Development*
Recommended Course: *SOLID Principles: Introducing Software Architecture & Design*

Numerical Representation



Computer Science is a field in which simple concepts are used in clever ways to emulate, simulate, and create much more complex ideas. The use of 0's and 1's is leveraged to represent the complex interplay between numbers. Square roots, exponentiation, scientific notation, and even imaginary numbers can be represented using only 1's and 0's.

These types of numbers often require complex systems of organizing, parsing, and storing binary representations—using only 1's and 0's. It isn't necessary that every programmer understand how floating-point notation works under the hood. However, the basic understanding that floating-point numbers require more memory and are subject to odd rounding errors is very useful.

Computers represent data in many ways but it can all be boiled down to 1's and 0's eventually. While data is ultimately represented as 1's and 0's, there are several common intermediate methods of representing numerical data. These are derived by using different mathematical base notations. Below are the three most common forms:

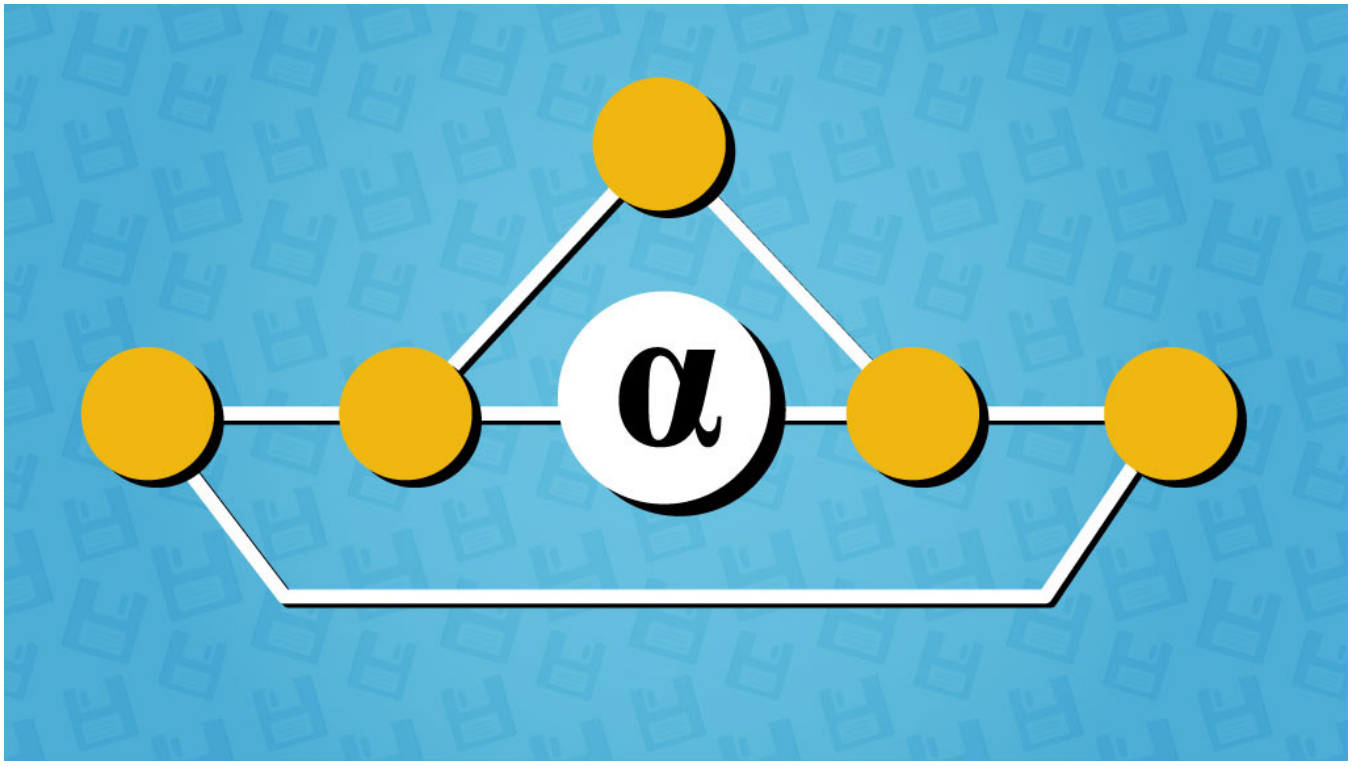
- Binary (base 2)
- Octal (base 8)
- Hexadecimal (base 16)

Binary data use base-2 notation, uses the digits 0 and 1, and is most commonly associated with

the 8-bit notation (a.k.a. *byte*) ranging from 0-128. Octal data uses base-8 notation, uses numbers 0-7, and is much less common today—used mostly for file permissions. The hexadecimal notation uses base-16 notation, introduces letters a-f into the notation, and is common among modern computing systems.

Recommended Reading: *Numerical Representation in Computers*
Recommended Course: *Build a Modern Computer From First Principles*

Version Control



Version control is a systematic means of tracking changes to codebases over time. It's often mistaken by beginner programmers as a simple cloud storage service—which it can technically be used for. Version control, a.k.a. source control is much more complex than a simple backup or cloud storage option.

Version control helps avoid conflicting changes made by many developers to a single codebase, track changes over time and much more. The Git protocol is the most popular form of version control and the technology behind GitHub. Below are the three most popular VC protocols:

- Git
- Mercurial
- Subversion

Mercurial offers a more distributed system that's lighter weight and more portable and Subversion offers a more centralized storage approach. Each may present an ideal solution for different projects. A basic understanding of their differences can help better design project management.

Recommended Book: *Pro Git*
Recommended Course: *Git Going Fast: One Hour Git Crash Course*

Tabs vs. Spaces

There is an age-old debate as to whether programmers should use tabs or spaces for indentation in code. Many programmers argue that two spaces are the proper number for indentation and another many programmers argue that four characters are canonical.

It's important to note that 'tabs' in this sense means the #9 ASCII byte as opposed to the actual TAB key on one's keyboard. Spaces have generally won out this argument in they offer the greatest degree of cross-platform compatibility—though the debate is far from settled.

Most modern IDE's allow one to set a custom value for the Tab key specifying the number of spaces or, God forbid, a Tab character to be used. Knowing when not to spark this debate is an essential skill for any programmer to exercise. One *study of modern programming languages* concluded that spaces were, in fact, the preferred means of representing indentations.

About the Author

Zack West is the founder of alpharithms.com, a website focused on developing resources for programmers, traders, and life-long learners to keep both their skill sets and minds sharpened. He is a regular contributor there as well as several other websites where he writes about programming, health, personal development, and 3D programming.

He is self-employed as a small business owner, a life-long student, and currently formalizing his knowledge of programming and software development through the pursuit of a second Bachelor's Degree—a B.Sc. in Computer Science.

West has 6+ years programming on a daily basis with a focus on stock analysis, web-scraping and data mining, natural language processing., and website development. He has spent the majority of his programming focus in Python but is comfortable in Java, C, C#, and JavaScript as well—though he can often be heard cursing the latter.

Zack can be reached for comment, questions, or general conversation at any of the following points of contact:

Email: *zwest@alpharithms.com*
Twitter: *@alphazwest*
Medium: *@alphazwest*