Java 101

By Prasad Jayakumar

My Perspective

On Learning

Image generated by https://www.canva.com/

The Before-How Wisdom

In the fast-paced tech world, 'How' - the intricate implementation - is time-consuming. But our time is precious. Embrace the 5Ws before diving into 'How'.

Download MAD 2023
Interactive MAD 2023

Why (Purpose): Understand why you're learning Java – whether it's for apps, backend, or skill growth. Purpose fuels motivation.

What (Content): Define your Java focus – core concepts, frameworks, etc. Clear goals ensure strong foundations.

Where (Application): Consider your application context – web, mobile, games, enterprise. Tailor learning to real-world scenarios.

When (Strategic Implementation): Determine the strategic moments to put your Java expertise to work within your chosen context. Ensure it aligns with project timelines and industry trends for maximum impact.

Who (Collaboration): Build a network for support – mentors, peers, colleagues. Collaboration enhances learning and practical use.

The Art of Code Review

Code Quality

Image generated by https://gencraft.com/

The First Round: A-OK or Not-So-OK

In the realm of coding, your code is either A-OK or Not-So-OK, all based on its functional aspects.

- If your code rocks, we label it "OK" (or "pass" for the formal audience).
- If it falls short, it gets the "Not OK" (or "fail") badge.

Image generated by https://gencraft.com/

Spectrum of Code Quality

We can spice things up with all sorts of cool adjectives. For instance:

Code Smell: Clean vs Dirty

- Clean Code: Code that follows best practices, is well-structured, readable, and easy to maintain.
- Dirty Code: Code that exhibits code smells, is poorly structured, violates coding standards, and is challenging to read and maintain.

Reliability: Trustworthy vs. Unpredictable

- Trustworthy: Reliable code is like a dependable safety net, providing assurance that critical operations will always function as expected.
- Unpredictable: Unpredictable code is akin to a fickle safety net, offering no guarantees and leaving you uncertain about whether it will catch you when needed.

Image generated by https://gencraft.com/

Spectrum of Code Quality

Performance: Swift vs. Sluggish

- **Swift Performance:** Swift code exhibits optimal performance characteristics, executing operations swiftly and efficiently. It leverages optimized algorithms, data structures, and resource management to achieve high-speed execution.
- **Sluggish Performance:** Sluggish code, in contrast, experiences suboptimal performance, characterized by slow execution and resource inefficiencies. It often suffers from poorly optimized code, resulting in delays and user frustration.

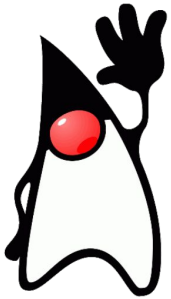
Maintainability: Flexible vs. Rigid

- **Flexible code** prioritizes modular design, loose coupling, and extensibility, allowing it to gracefully adapt to evolving requirements
- In contrast, **rigid code** tends to be monolithic, tightly coupled, and resistant to change, making it challenging to maintain and extend.

Image generated by https://gencraft.com/

Prerequisites

Environment Setup



Tools

- SDKMan - Download
- OpenJDK 11, 17 & 21 - LTS
- AWS Corretto - Homepage
- Eclipse - Download

SDKMAN!

- sdk version
- sdk env init
- sdk list java
- sdk install java 17.0.8-amzn
- sdk home java 17.0.8-amzn
- sdk default java 17.0.8-amzn
- sdk use java 17.0.8-amzn

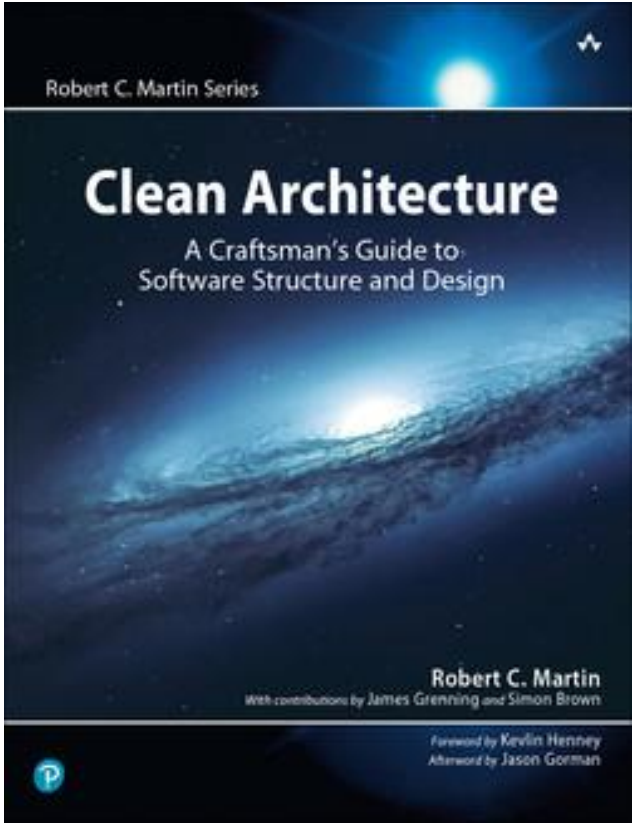
```
================================================================================
Available Java Versions for Linux 64bit
================================================================================
 Vendor        | Use | Version      | Dist    | Status     | Identifier
--------------------------------------------------------------------------------
 Corretto      |     | 21           | amzn    |            | 21-amzn
               |     | 20.0.2       | amzn    |            | 20.0.2-amzn
               |     | 20.0.1       | amzn    |            | 20.0.1-amzn
               |     | 17.0.8       | amzn    |            | 17.0.8-amzn
               |     | 17.0.7       | amzn    |            | 17.0.7-amzn
               |     | 11.0.20      | amzn    |            | 11.0.20-amzn
               |     | 11.0.19      | amzn    |            | 11.0.19-amzn
               |     | 8.0.382      | amzn    |            | 8.0.382-amzn
               |     | 8.0.372      | amzn    |            | 8.0.372-amzn
```

more...

```
prasad@four-dots:~/java-101$ sdk home java 17.0.8-amzn
/home/prasad/.sdkman/candidates/java/17.0.8-amzn
prasad@four-dots:~/java-101$ java --version
openjdk 17.0.8 2023-07-18 LTS
OpenJDK Runtime Environment Corretto-17.0.8.7.1 (build 17.0.8+7-LTS)
OpenJDK 64-Bit Server VM Corretto-17.0.8.7.1 (build 17.0.8+7-LTS, mixed mode, sharing)
prasad@four-dots:~/java-101$ sdk default java 17.0.8-amzn
setting java 17.0.8-amzn as the default version for all shells.
```

S.O.L.I.D.

Design Principles



The Single Responsibility Principle (SRP)

~~"A module should have one, and only one, reason to change."~~

"A module should be responsible to one, and only one, actor"

Rationale

- By ensuring that a module has only one responsibility, we make it easier to understand, maintain, and extend.
- Changes to one responsibility of a class should not affect the others.

Examples

- A **Logger** class should be responsible for logging messages and not for formatting them or handling network communication.
- A **UserRepository** class should be responsible for data access and not for business logic or authentication.

SRP Example in Details

Employee class violates the SRP because those three methods are responsible to three very different actors.

- The calculatePay() method is specified by the accounting department, which reports to the CFO.
- The reportHours() method is specified and used by the human resources department, which reports to the COO.
- The save() method is specified by the database administrators (DBAs), who report to the CTO.

By putting the source code for these three methods into a single Employee class, the developers have coupled each of these actors to the others.

Now suppose that the CFO's team decides that the way non-overtime hours are calculated needs to be tweaked. In contrast, the COO's team in HR does not want that particular tweak because they use non-overtime hours for a different purpose. We have a problem.

The way to avoid this problem is to separate code that supports different actors.

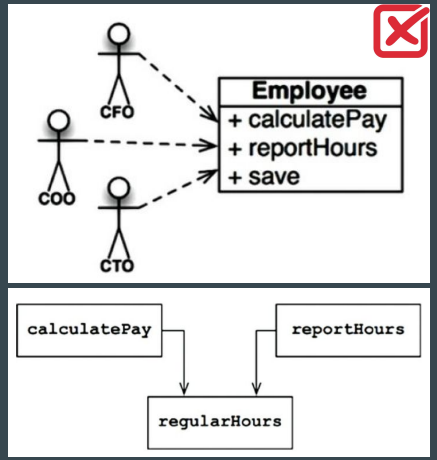


Image from the Book "Clean Architecture"

The Open-Closed Principle (OCP)

“Software entities (classes, modules, functions) should be open for extension but closed for modification.”

Rationale

- By being "open for extension," you can add new features by creating new code that builds upon existing, stable code, rather than modifying that code directly.

Examples

- The logging framework provides a common interface for logging, and it allows you to configure different log output formats, destinations (e.g., console, files), and log levels without changing your application's core logic. This flexibility is a key aspect of adhering to OCP in logging.

The Liskov “Substitution” Principle (LSP)

“Behavioral subtyping -

If for each object o1 of type S there is an object o2 of type T

such that for all programs P defined in terms of T,

the behavior of P is unchanged when o1 is substituted for o2

then S is a subtype of T”

Rationale

- LSP guarantees that derived classes honor the contracts and behaviors specified by the base class, making it possible to extend software systems with new derived classes without needing to modify existing code.

Examples

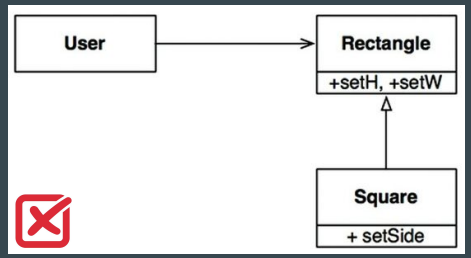
- The Java Collections Framework, specifically the List interface and its various implementations (e.g., ArrayList, LinkedList, Vector, etc.), exemplify LSP.

LSP Example in Details

The infamous square/rectangle problem. In this example, Square is not a proper subtype of Rectangle because

- the height and width of the Rectangle are independently mutable;
- in contrast, the height and width of the Square must change together.

Since the User believes it is communicating with a Rectangle, it could easily get confused




```java
package org.fourdots.solid.lsp.bad;

public class RectangleSquareDemo {
    public static void main(String[] args) {
        Rectangle rectangle = new Square();
        rectangle.setWidth(5);
        rectangle.setHeight(4);

        // Let's calculate the area. Expecting 20, but area is 16.
        int area = rectangle.getWidth() * rectangle.getHeight();
        System.out.println("Area: " + area);
    }
}
```

The Interface Segregation Principle (ISP)

“Clients (those who use interfaces) are not forced to depend on methods they do not use”

Rationale

- Break large, monolithic interfaces into smaller, more specialized ones, each catering to a specific group of clients. This way, classes can choose to implement only the interfaces that contain the methods they require and avoiding unnecessary dependencies.

Examples

- The intent of the Collection interface and its subinterfaces (List, Set, Queue, etc.) is to provide a common way to work with collections of objects. However, these subinterfaces are segregated based on their specific behaviors and use cases.

ISP Example in Details

The Java Collections Framework generally follows the Interface Segregation Principle (ISP).

Separation of Interfaces: The framework separates interfaces based on specific use cases and behaviors. For example:

- java.util.Collection defines the fundamental methods for working with collections, such as add, remove, and contains.
- java.util.List (a subinterface of Collection) adds methods for indexed access, like get and set.
- java.util.Set (another subinterface of Collection) defines methods for collections that do not allow duplicate elements.
- java.util.Queue (yet another subinterface) provides methods for working with queues, like offer, poll, and peek.

Implementation Choices: The framework allows developers to choose the most appropriate implementation of an interface based on their specific use case. For example, you can use ArrayList or LinkedList based on your requirements for list-like behavior, and both classes adhere to the methods defined by the List interface.

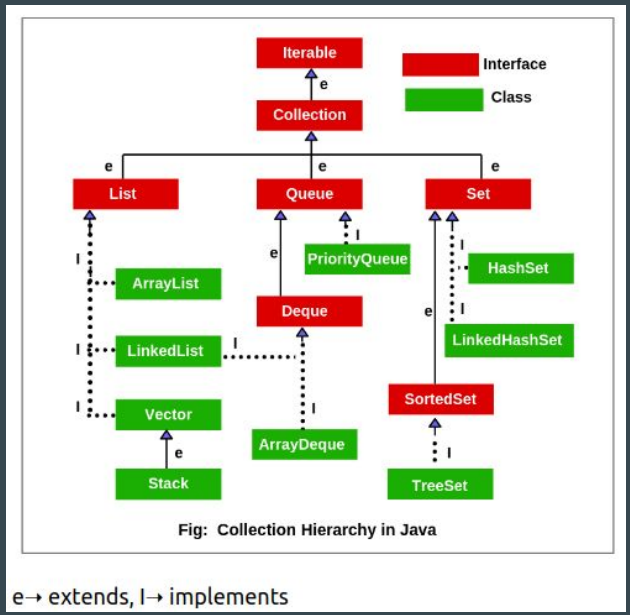


Image Source

The Dependency Inversion Principle (DIP)

“High-level modules should not depend on low-level modules. Both should depend on abstractions.”

Rationale

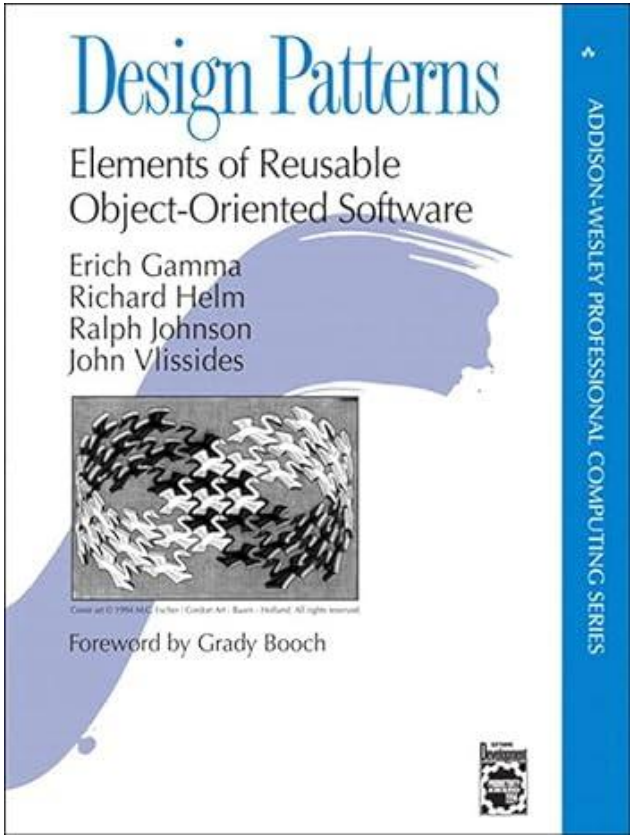
- DIP addresses the issues of tight coupling and rigidity in software design.
- High-level modules can work with different low-level modules that adhere to the same abstractions, and low-level modules can be replaced or modified without affecting the high-level modules.

Examples

- Repository Pattern: Isolates the data layer from the rest of the application, providing an abstraction for how data is accessed and manipulated.

Design Patterns

Gang-of-Four Design Patterns



Design Pattern Space

Purpose, reflects what a pattern does

- **Creational** patterns concern the process of object creation
- **Structural** patterns deal with the composition of classes or objects
- **Behavioral** patterns characterize the ways in which classes or objects interact and distribute responsibility.

Scope, specifies whether the pattern applies primarily to classes or to objects

- **Class** patterns deal with relationships between classes and their subclasses. They are static - fixed at compile-time.
- **Object** patterns deal with object relationships, which can be changed at run-time and are more dynamic.

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

From Book "Design Patterns by Gang of Four"

Creational Patterns

Design Pattern	Aspect(s) That Can Vary
Singleton	the sole instance of a class
Prototype	class of object that is instantiated
Builder	how a composite object gets created
Abstract Factory	families of product objects
Factory Method	subclass of object that is instantiated

From Book "Design Patterns by Gang of Four"

Singleton Pattern

Ensure a class has only one instance and provides a global point of access to that instance.

Applicability: Use when exactly one object needs to coordinate actions across the system, such as a configuration manager, logging service, or thread pool.

Consequences: Provides a single point of control, but can limit extensibility and testability if not used carefully.

Known Uses: Logging services, database connection pools, thread pools, and caching mechanisms often use the Singleton pattern.

Prototype Pattern

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Applicability: Use when the cost of creating an object is more expensive or complex than copying an existing object, and when you want to avoid subclassing to create new objects.

Consequences: Allows dynamic creation of new objects with minimal overhead, but can be challenging to implement with objects that have complex dependencies.

Known Uses: Object cloning in Java, where you can create new objects by copying existing ones, is a common use of the Prototype pattern.

Builder Pattern

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Applicability: Use when an object needs to be constructed with many optional components or configurations, and when you want to improve the readability of object creation code.

Consequences: Allows for the creation of complex objects with a clear separation of concerns, but can result in a more verbose code compared to other creational patterns.

Known Uses: Often used for building complex data structures, such as HTML parsers, document generators, and configuration builders.

Abstract Factory Pattern

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Applicability: Use when a system must be independent of how its objects are created, composed, and represented, and when a system is configured with multiple families of objects.

Consequences: Ensures the compatibility of objects created within a factory, but can be complex to implement, especially for large families of objects.

Known Uses: Graphical user interface libraries and database access libraries often use Abstract Factory pattern.

Factory Method Pattern

Define an interface for creating an object but let subclasses alter the type of objects that will be created.

Applicability: Use when a class cannot anticipate the class of objects it must create or when a class wants to delegate the responsibility of object creation to its subclasses.

Consequences: Promotes loose coupling between the creator and product classes, but can result in a proliferation of factory classes.

Known Uses: GUI frameworks, libraries for database access, and document processing tools commonly use Factory Method pattern.