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 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.
```

Java Essentials

Building a Strong Foundation

"Wax On Wax Off" - The Karate Kid (1984)
"Jacket On Jacket Off" - The Karate Kid (2010)

Primitive Data Types

Number	Name	Number of Zeros
1,000,000,000	Billion	9 Zeros 💰
1,000,000,000,000	Trillion	12 Zeros 🚀
1,000,000,000,000,000	Quadrillion	15 Zeros 💎
1,000,000,000,000,000,000	Quintillion	18 Zeros 🌠

Primitive Data Types

Data Type	Size (in bits)	Range	Usage
byte	8 bits	-128 to 127	Used for small integers and memory efficiency
short	16 bits	-32768 to 32767	Suitable for a broader range of integer values
char	16 bits	0 to 65535	Stores single character (Unicode characters)
int	32 bits	-2.1 billion to 2.1 billion	Commonly used for integers in general calculations
long	64 bits	-9.2 quintillion to 9.2 quintillion	Used for large integers
float	32 bits	Approx. ±3.4 x 10^-38 to ±3.4 x 10^38	Decimals with lower precision
double	64 bits	Approx. ±1.7 x 10^-308 to ±1.7 x 10^308	Double precision decimals
boolean	N/A	true or false	Represents binary true/false or on/off values.

Character Encoding

ASCII: The Foundation of Character Encoding

- ASCII, or the American Standard Code for Information Interchange, functions as the foundational character encoding system.
- It utilizes numerical values (code points) spanning from 0 (00) to 127 (7F) for character representation.
- ASCII encompasses control characters, digits, letters, and basic punctuation symbols.
- Out of the 128 code points in ASCII, only 95 of them represent printable characters, which significantly limits its range.

Character set [edit]

ASCII (1977/1986)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2x	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Changed or added in 1963 version

Changed in both 1963 version and 1965 draft

https://en.wikipedia.org/wiki/ASCII

ISO-8859: Expanding Character Support

- The ISO-8859 series extended character support beyond ASCII for Latin-based languages.
- ISO-8859-1 (Latin-1) is widely used and includes characters for Western European languages.
- ISO-8859 was a significant step in accommodating diverse character sets.
- It enabled computers to handle text in multiple languages, making it important for global communication.

ISO/IEC 8859-1

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x																
1x																
2x	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
8x																
9x																
Ax	NBSP	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	SHY	®	¯
Bx	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Dx	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
Ex	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
Fx	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Undefined

Symbols and punctuation

Undefined in the first release of ECMA-94 (1985).[16] In the original draft Œ was at 0xD7 and œ was at 0xF7.

https://en.wikipedia.org/wiki/ISO/IEC_8859-1

Unicode and the Basic Multilingual Plane (BMP)

- Unicode is a modern, comprehensive character encoding standard.
- It unifies characters from various languages, scripts, and symbols into a single system.
- The Basic Multilingual Plane (BMP) is the first and most commonly used part of Unicode.

  https://en.wikipedia.org/wiki/Plane_(Unicode)

- BMP includes over 65,000 characters, covering many world languages.

Telugu[1][2]

Official Unicode Consortium code chart (PDF)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
U+0C0x	◌ఀ	◌ఁ	◌ం	◌ః	◌ఄ	అ	ఆ	ఇ	ఈ	ఉ	ఊ	ఋ	ఌ		ఎ	ఏ
U+0C1x	ఐ		ఒ	ఓ	ఔ	క	ఖ	గ	ఘ	ఙ	చ	ఛ	జ	ఝ	ఞ	ట
U+0C2x	ఠ	డ	ఢ	ణ	త	థ	ద	ధ	న		ప	ఫ	బ	భ	మ	య
U+0C3x	ర	ఱ	ల	ళ	ఴ	వ	శ	ష	స	హ			◌఼	ఽ	◌ా	◌ి
U+0C4x	◌ీ	◌ు	◌ూ	◌ృ	◌ౄ		◌ె	◌ే	◌ై		◌ొ	◌ో	◌ౌ	◌్		
U+0C5x						◌ౕ	◌ౖ		ౘ	ౙ	ౚ			ౝ		
U+0C6x	ౠ	ౡ	◌ౢ	◌ౣ			౦	౧	౨	౩	౪	౫	౬	౭	౮	౯
U+0C7x								౷	౸	౹	౺	౻	౼	౽	౾	౿

Notes

1. ^ As of Unicode version 15.1
2. ^ Grey areas indicate non-assigned code points

https://en.wikipedia.org/wiki/Telugu_(Unicode_block)

UTF-8 vs UTF-16

	UTF-8	UTF-16
Encoding Approach	Variable length encoding	Variable length encoding - 2 bytes for BMP, 4 bytes for characters beyond BMP
Byte Order	No specific byte order (Endian)	Can be Little-Endian or Big-Endian
Storage Efficiency	Efficient for English and most Latin characters	Less space-efficient for English and Latin scripts, more efficient for other scripts
Usage	Commonly used on the web, including HTML, JSON etc.	Frequently used in software and systems that require a fixed-width encoding. Ex., NTFS, store file names in UTF-16.

- Java 18 makes UTF-8 the default charset, bringing an end to most issues related to the default charset in versions before Java 18.
- UTF-8 is widely used on the world wide web. Also, most Java programs use UTF-8 to process JSON and XML. Additionally, Java APIs like java.nio.Files use UTF-8 by default.

Reference Types

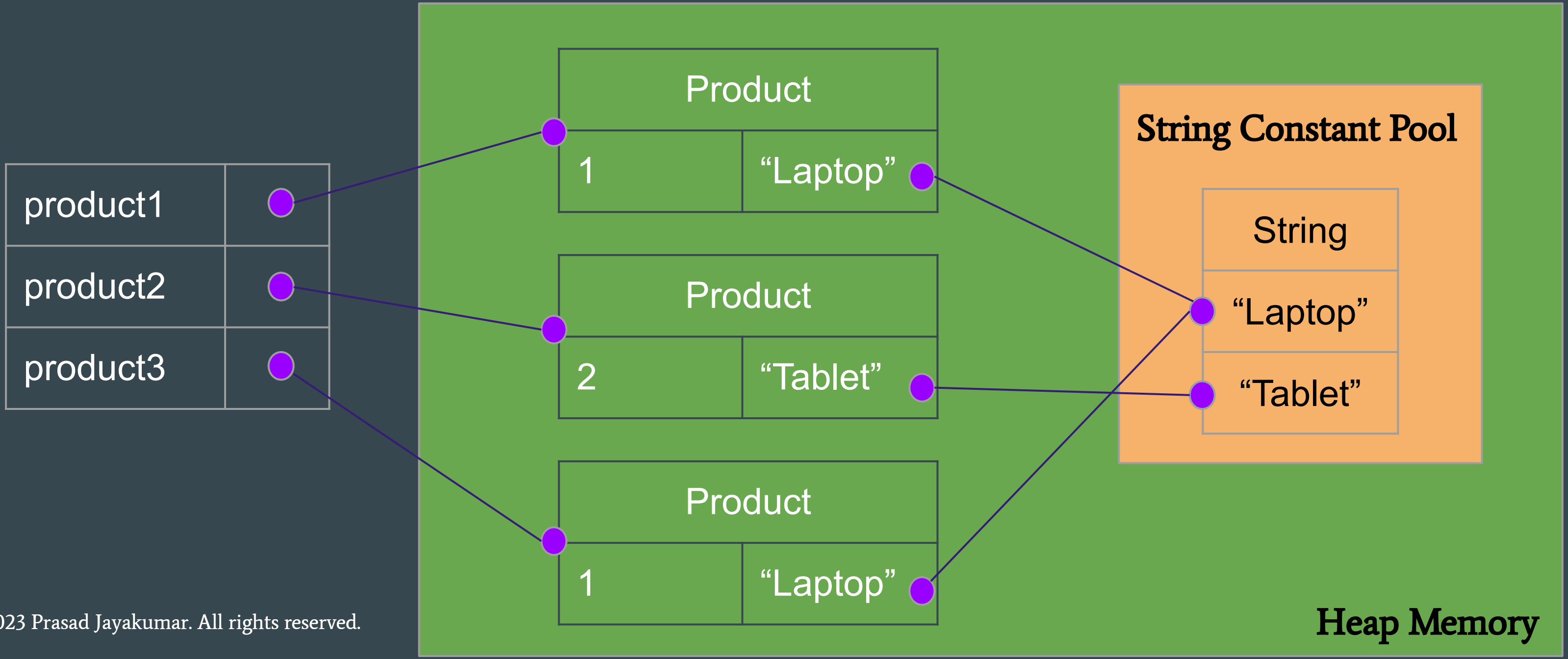
Reference Types

Reference types in Java are a fundamental concept that allows developers to work with objects, effectively managing memory and data.

- Reference vs. Data: Reference types store memory addresses, not data itself.
- Assignment: Assigning one reference to another connects them to the same object in memory.
- Passing References: When an object is passed to a method, the method can modify its contents but not the reference.

Reference Types - Contd.

```
Product product1 = new Product(1, "Laptop");
Product product2 = new Product(2, "Tablet");
Product product3 = new Product(1, "Laptop");
```



Conversion of Reference Types

Widening Conversions (Upcasting)

- Widening implicitly converts a subclass to a parent class (superclass).
- Widening conversions do not throw runtime exceptions.
- No explicit cast is necessary:

```
Object product1 = new Product(1, "Laptop");
Object product2 = new Product(2, "Tablet");
Object product3 = new Product(1, "Laptop");
```

Conversion of Reference Types

Narrowing Conversions (Downcasting)

- Narrowing converts a more general type into a more specific type.
- Narrowing is a conversion of a superclass to a subclass.
- An explicit cast is required. To cast an object to another object, place the type of object to which you are casting in parentheses immediately before the object you are casting.
- Illegitimate narrowing results in a ClassCastException.
- Narrowing may result in a loss of data/precision.

```
Object product1 = new Product(1, "Laptop");
Product product2 = (Product) product1;
```

String

String Comparison

```
String laptopA = "Laptop";
String laptopB = "laptop";

System.out.println(laptopA == laptopB);
System.out.println(laptopA.toUpperCase() == laptopB.toUpperCase());
System.out.println(laptopA.equalsIgnoreCase(laptopB));
System.out.println(laptopA.toUpperCase().intern() == laptopB.toUpperCase().intern());
```

Comparison	Explanation	Result
laptopA == laptopB	Checks if the two strings reference the same **memory location**. In this case, it will print false because laptopA and laptopB are **two different string literals** with different casing.	false
laptopA.toUpperCase() == laptopB.toUpperCase()	Converts both strings to uppercase and checks if they reference the same memory location. This returns false because toUpperCase() **creates new strings**.	false
laptopA.equalsIgnoreCase(laptopB)	Ignores the case and **checks if the content of the strings is equal**. This returns true because the content is the same.	true
laptopA.toUpperCase().intern() == laptopB.toUpperCase().intern()	Converts both strings to uppercase and uses intern() to return a **canonical representation in the string pool**. This returns true because they share the same reference in the string pool.	true

The recommended method for comparing strings for equality is to use the equals() or equalsIgnoreCase() method

Object

Class Object

- The Object class is at the root of the Java class hierarchy.
- Every class in Java implicitly inherits from the Object class, which means that every Java object is an instance of the Object class. This makes it a fundamental and important class in Java.

Method	Description
toString()	Returns a string representation of the object.
equals(Object obj)	Indicates whether some other object is "equal to" this one.
hashCode()	Returns a hash code value for the object.
getClass()	Returns the runtime class of this Object.

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Object.html

Java Collections Framework

Collection Type Characteristics

https://visualgo.net/en/list

Concrete Type	Interface	Ordered	Sorted	Allows Duplicates
ArrayList	List	Yes	No	Yes
LinkedList	List	Yes	No	Yes
HashMap	Map	No	No	No (for keys)
LinkedHashMap	Map	Insertion, Last access	No	No (for keys)
TreeMap	Map	Balanced	Yes	No (for keys)
HashSet	Set	No	No	No
LinkedHashSet	Set	Insertion	No	No
TreeSet	Set	Sorted	Yes	No

ArrayList

- A dynamic array that stores multiple objects of any class or data type.
- ArrayLists have no fixed size limit, making them flexible for dynamic storage.
- ArrayLists provide fast access to elements by index with O(1) time complexity for random access.
- Inserting or removing elements in the middle of an ArrayList can be less efficient due to potential element shifting, resulting in O(n) time complexity.
- ArrayLists are not thread-safe by default and may require external synchronization for concurrent access.

ArrayList [Source File]

- When an ArrayList reaches its maximum capacity, it automatically increases its capacity using a formula

  ((current capacity * 3/2) + 1).

- Existing elements are copied to the new ArrayList.
- The new element is added to the new ArrayList.
- The reference is reassigned to the new ArrayList, allowing the old one to be garbage collected.

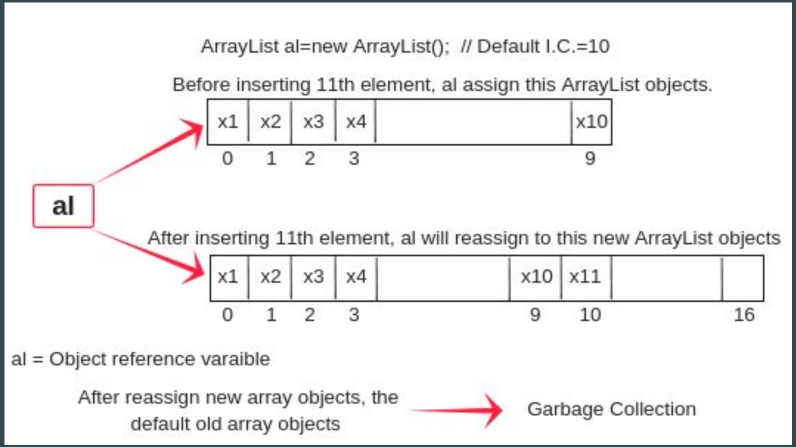


Image Source: https://www.scientecheasy.com/2020/09/arraylist-in-java.html/

LinkedList Source File

- LinkedList uses a doubly linked list to store elements.
- It doesn't have an initial capacity; it starts with a size of zero.
- The size of a LinkedList grows when elements are added and shrinks when elements are removed.

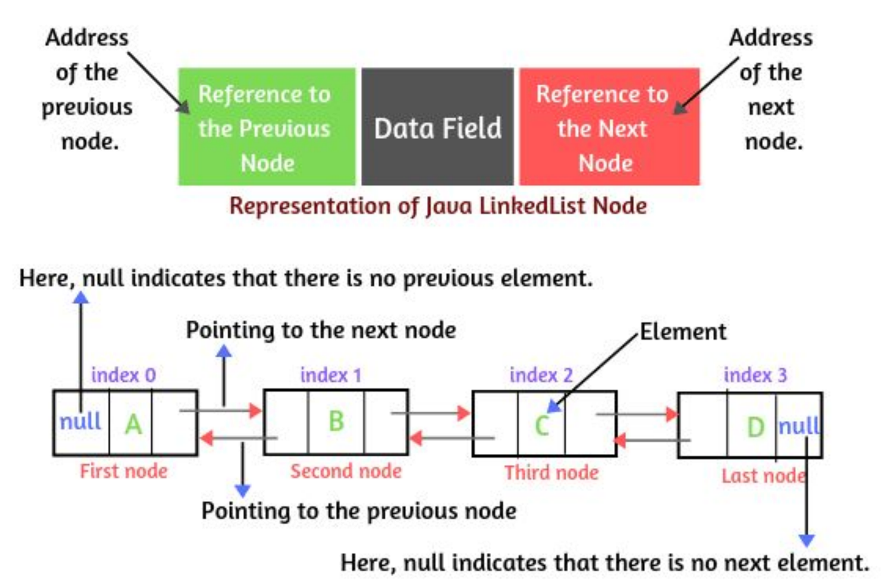


Image Source: https://www.scientecheasy.com/2020/09/java-linkedlist.html/

Map

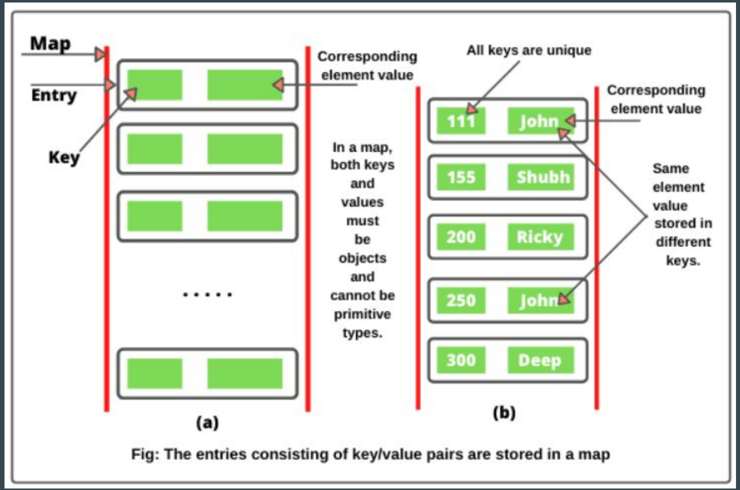


Image Source: https://www.scientecheasy.com/2020/10/map-in-java.html/

HashMap

Source File

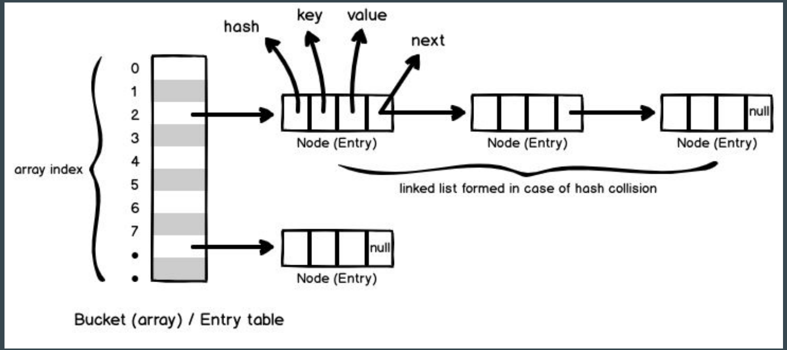


Image Source: https://www.javaquery.com/2019/11/how-hashmap-works-internally-in-java.html

Set

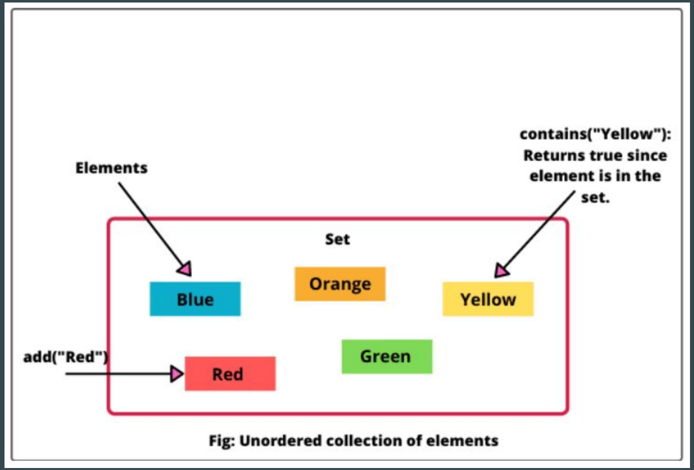


Image Source: https://www.scientecheasy.com/2020/10/java-set.html/

Choosing the Right 'List' Data Structure

- Use ArrayList, if you need fast random access and have a good estimate of the list's size. Set the initial capacity based on your estimate.
  - The default initial capacity is 10, and the loading factor is 0.75, which means the array will be resized when it's 75% full.
  - Adjust the initial capacity based on your specific use case, but it's usually not critical to fine-tune this value unless you're working with very large data sets.
- Use LinkedList, if you need to frequently add or remove elements from both ends of the list, or if you don't have a good estimate of the list's size.

Choosing the Right 'Map' Data Structure

- Use HashMap when you need fast lookups, and the order of entries is not important.
- Use LinkedHashMap when you want to maintain the insertion order of entries or implement LRU caching.
- Use TreeMap when you require sorted access to entries, support complex sorting criteria, or need to perform range queries.

Choosing the Right ‘Set’ Data Structure

- Use HashSet for fast lookups and when the order of elements is not important.
- Use LinkedHashSet when you want to maintain the insertion order of elements.
- Use TreeSet when you require ordered access to elements, support complex sorting criteria, or need to perform range queries.

Time Complexities of Java Collections

O(1) - Constant Time

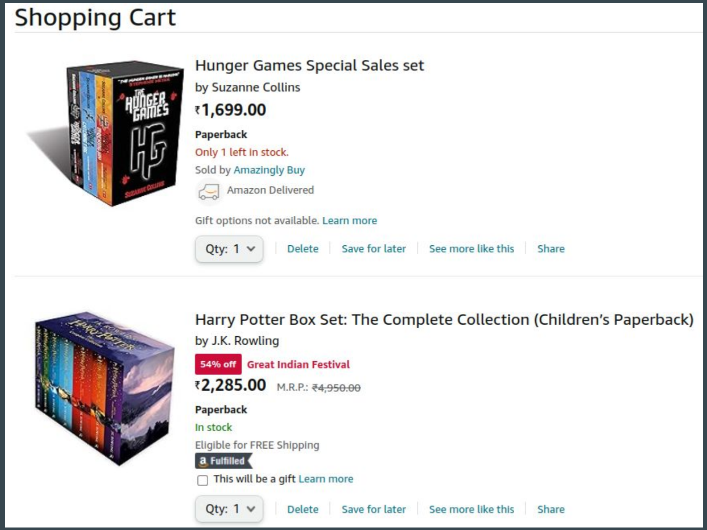
O(log n) - Logarithmic Time - It's more efficient than linear time for large datasets.

O(n) - Linear Time - As the input size grows, the execution time grows at the same rate.

Concrete Type	get	add / put	remove	contains
ArrayList	o(1)	o(n)	o(n)	o(n)
LinkedList (from either end)	o(1)	o(1)	o(1)	o(n)
LinkedList (from / at index)	o(n)	o(n)	o(n)	o(n)
HashMap	o(1)	o(1)	o(1)	o(1)
TreeMap	o(log n)	o(log n)	o(log n)	o(log n)
HashSet	na	o(1)	o(1)	o(1)
TreeSet	na	o(log n)	o(log n)	o(log n)

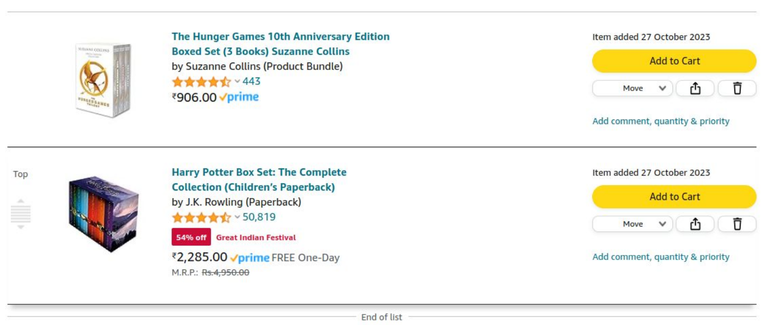
Use Case - Shopping Cart

- Basic Functions
  - Add an item to the cart
  - Display the contents of the cart
  - Modify the quantity of an item
  - Remove an item from the cart
- Common Characteristics
  - Newly added items are positioned at the top of the list.
  - The order of items is preserved.
  - Updating or deleting actions are executed based on the product's unique identifier (productId).
  - The cart's initial size is unknown and expands incrementally.



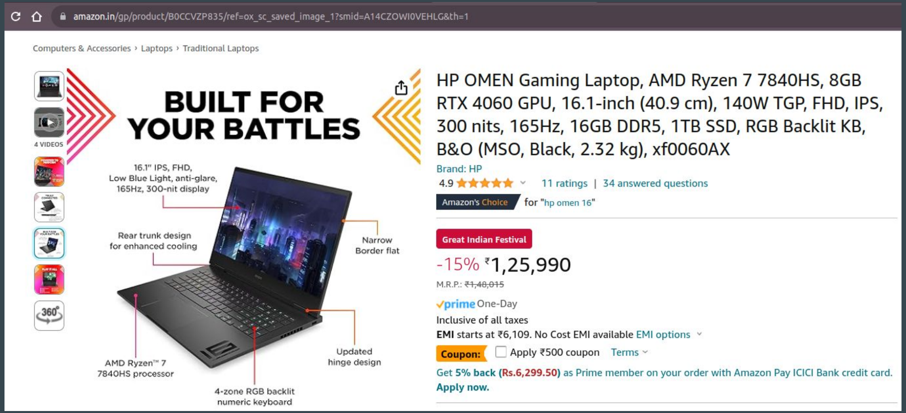
Use Case - Wish List

- Basic Functions
  - Add an item to the wish-list
  - Display the contents of the wish-list
  - Rearrange the wish-list item order
  - Remove an item from the wish-list
- Common Characteristics
  - Newly added items are positioned at the top of the list.
  - The order of items is preserved.
  - Deleting actions are executed based on the product's unique identifier (productId).



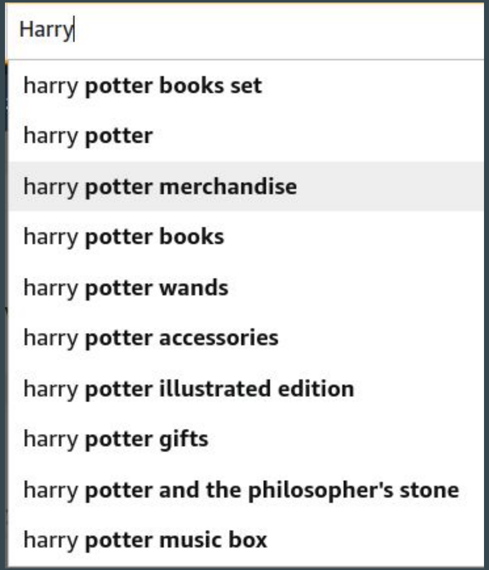
Use Case - Product Details Page

- Basic Functions
  - Create a product catalog to store and manage products.
  - Retrieve product details using a unique product-id.
- Common Characteristics
  - The catalog doesn't require a specific order for products.
  - Optimize data structure for quick and efficient product retrieval.
  - Ensure that each product has a distinct and unique identifier.



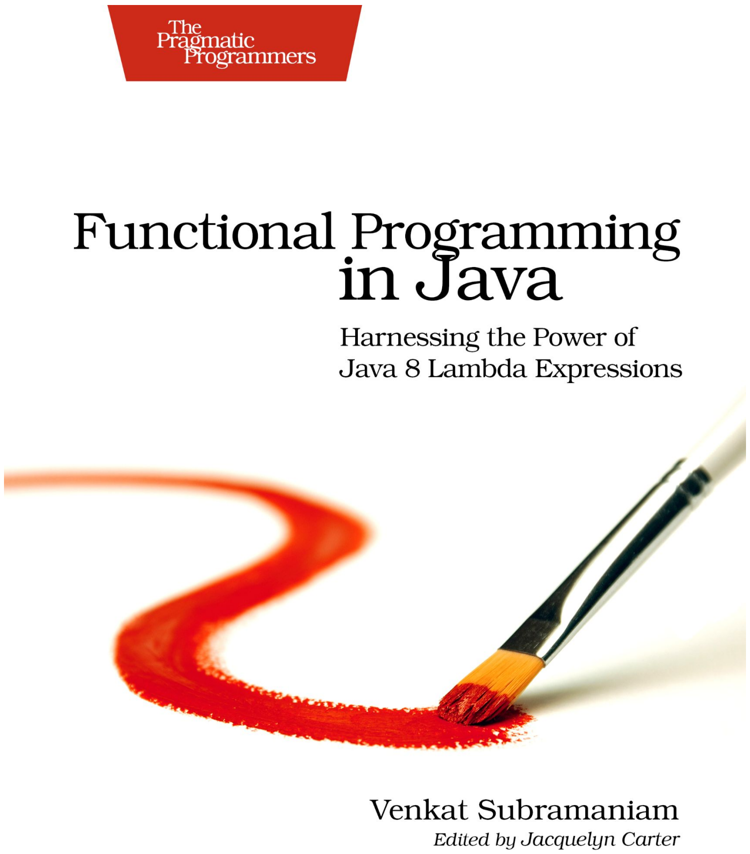
Use Case - Auto Suggestions

- Basic Functions
  - Provide real-time keyword suggestions as the user types, enhancing their search or input experience
  - Suggest relevant content like product names
- Common Characteristics
  - Keep the case insensitive
  - Limit the result to top 10 relevant values



Functional Programming

Java 8 Lambda



Imperative vs Declarative Style

Imperative Style	Declarative Style
You explicitly specify **how to achieve a goal** through step-by-step instructions	You describe **what you want to achieve,** and the system takes care of how to do it
It often involves mutable variables, loops, and explicit control flows	It often involves higher-level abstractions, immutable data, and functional constructs like lambdas and streams

`int[] numbers = {1, 2, 3, 4, 5};`
`int sum = 0;`
`for (int i = 0; i < numbers.length; i++) {`
`    sum += numbers[i];`
`}`
`System.out.println("Sum: " + sum);`
	`List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);`
	`int sum = numbers.stream()`
	`                .reduce(0, (x, y) -> x + y);`
	`System.out.println("Sum: " + sum);`

Why Functional Style?

- The code is more expressive.
- The functional-style is concise and intuitive.
- We avoided explicit mutation or reassignment of variables.
- The functional version can easily be parallelized.

The functional style is not counter to object-oriented programming (OOP). The real paradigm shift is from the imperative to the declarative style of programming. We can intermix functional and OO styles of programming quite effectively.

Collections - Iterations

External Iterator - Imperative Style	`for (int i = 0; i < cities.size(); i++) {`
	`    System.out.println(cities.get(i));`
	`}`

External Iterator - Imperative Style	`for (String name : cities) {`
	`    System.out.println(name);`
	`}`

Internal Iterator - Functional Style	`cities.forEach((city) -> System.out.println(city));`

Collections - Transform

Imperative Style	`final List<String> ucCities = new ArrayList<String>();`
	`for (String city : cities) {`
	`    ucCities.add(city.toUpperCase());`
	`}`

Non-Functional Style	`final List<String> ucCities = new ArrayList<String>();` `cities.forEach(city -> ucCities.add(city.toUpperCase()));`

Functional Style	`final List<String> ucCities = cities.stream()`
	`        .map(city -> city.toUpperCase())`
	`        .toList();`

Collections - Filter

<table>
<tr><td>Imperative Style</td><td>

```java
final List<String> selectedCities = new ArrayList<String>();
for (String city : cities) {
    if (city.startsWith("C")) {
        selectedCities.add(city);
    }
}
```

</td></tr>
<tr><td>Functional Style</td><td>

```java
cities.stream()
 .filter(city -> city.startsWith("C"))
 .collect(Collectors.toList());
```

</td></tr>
</table>

Collections - Function Composition

```java
List<Product> products = Arrays.asList(
        new Product(3, "Mobile", new BigDecimal("100.65"), true),
        new Product(1, "BookA", new BigDecimal("10.23"), true),
        new Product(2, "BookB", new BigDecimal("20.54"), false),
        new Product(4, "Food", new BigDecimal("5.30"), true));

// Find high price under $100 and in-stock
System.out.println(products.stream()
        .filter(FnComposition::isInStock)
        .filter(isPriceLessThan100)
        .findFirst()
        .orElse(null));
```

```
isInStock >> Mobile
isPriceLessThan >> 100.65
isInStock >> BookA
isPriceLessThan >> 10.23
Product [id=1, name=BookA,
```

```java
// Find high price under $100 and in-stock
System.out.println(products.stream()
        .filter(isPriceLessThan100)
        .filter(FnComposition::isInStock)
        .findFirst()
        .orElse(null));
```

```
isPriceLessThan >> 100.65
isPriceLessThan >> 10.23
isInStock >> BookA
Product [id=1, name=BookA,
```

Closure

Enclosing Scope

```java
public static List<String> enclosingScope(final List<String> cities, final String filterBy) {
    return cities.stream()
            .filter(name -> name.startsWith(filterBy))
            .collect(Collectors.toList());
}
```

From within a lambda expression we can only access local variables that are final or effectively final in the enclosing scope.

Enclosing Scope with Error

```java
public static List<String> scopeError(final List<String> cities) {
    String filterBy = "C";
    List<String> selectedCities = cities.stream()
            .filter(name -> name.startsWith(filterBy))
            .collect(Collectors.toList());

    filterBy = "D";
    return selectedCities;
}
```

Local variable filterBy defined in an enclosing scope must be final or effectively final

Press 'F2' for focus

Functional Programming with Java 8 by Venkat Subramaniam

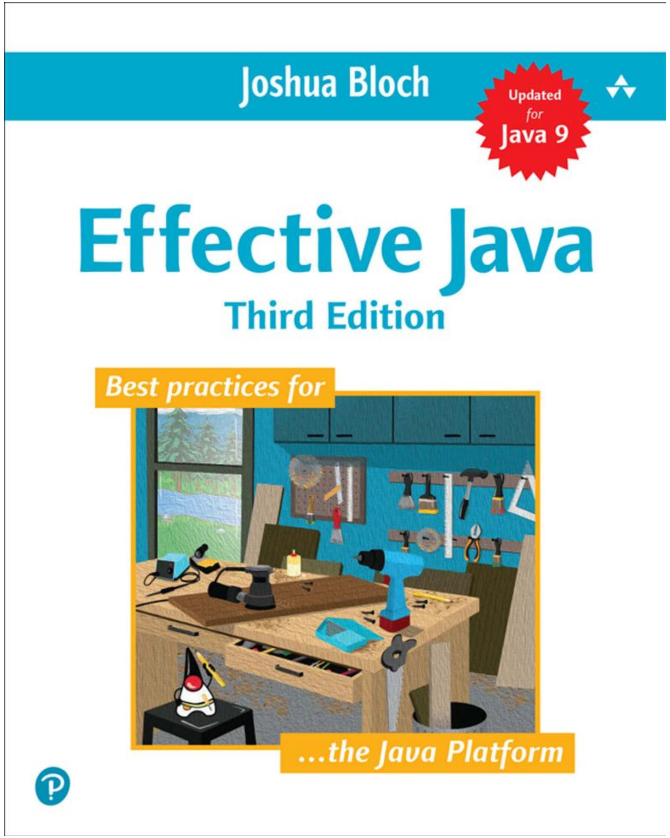
https://www.youtube.com/watch?v=15X0qFtBqiQ

Get a Taste of Lambdas and Get Addicted to Streams [OPTIONAL]

https://www.youtube.com/watch?v=1OpAgZvYXLQ

Effective Java

Customary and Effective Usage



[Source Code]

Creating and Destroying Objects

Creating and Destroying Objects

ID	Guideline	Level
Item 1	Consider static factory methods instead of constructors	Basic
Item 2	Consider a builder when faced with many constructor parameters	Basic
Item 3	Enforce the singleton property with a private constructor or an enum type	Basic
Item 4	Enforce noninstantiability with a private constructor	Basic
Item 5	Prefer dependency injection to hardwiring resources	Advance

Creating and Destroying Objects - Contd.

ID	Guideline	Level
Item 6	Avoid creating unnecessary objects	Basic
Item 7	Eliminate obsolete object references	Basic
Item 8	Avoid finalizers and cleaners	Advance
Item 9	Prefer try-with-resources to try-finally	Advance

Methods Common to All Objects

Methods Common to All Objects

ID	Guideline	Level
Item 10	Obey the general contract when overriding equals	Basic
Item 11	Always override hashCode when you override equals	Basic
Item 12	Always override toString	Basic
Item 13	Override clone judiciously	Advance
Item 14	Consider implementing Comparable	Basic

Classes and Interfaces

Classes and Interfaces

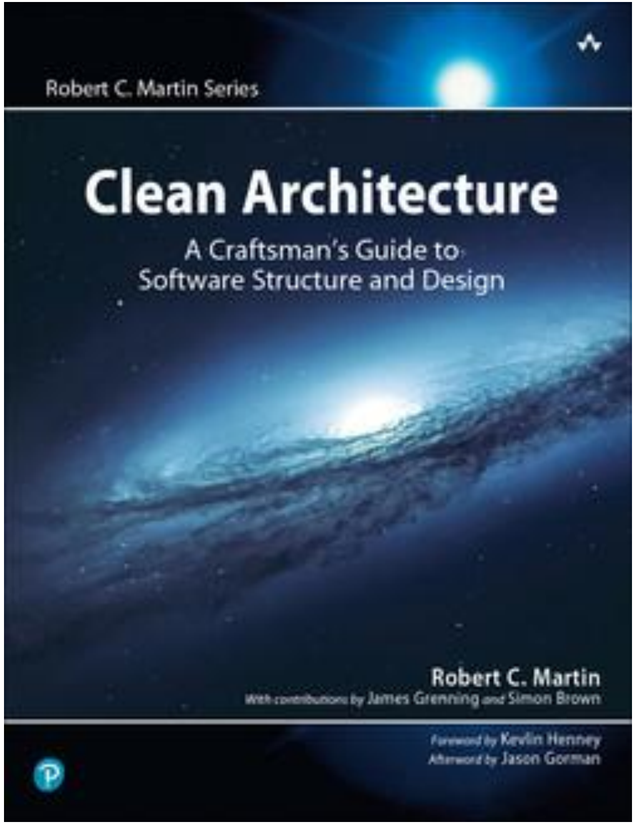
ID	Guideline	Level
Item 15	Minimize the accessibility of classes and members	Basic
Item 16	In public classes, use accessor methods, not public fields	Basic
Item 17	Minimize mutability	Basic
Item 18	Favor composition over inheritance	Advance
Item 19	Design and document for inheritance or else prohibit it	Basic

Classes and Interfaces

ID	Guideline	Level
Item 20	Prefer interfaces to abstract classes	Basic
Item 21	Design interfaces for posterity	Basic
Item 22	Use interfaces only to define types	Basic
Item 23	Prefer class hierarchies to tagged classes	Basic
Item 24	Favor static member classes over nonstatic	Basic
Item 25	Limit source files to a single top-level class	Basic

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.

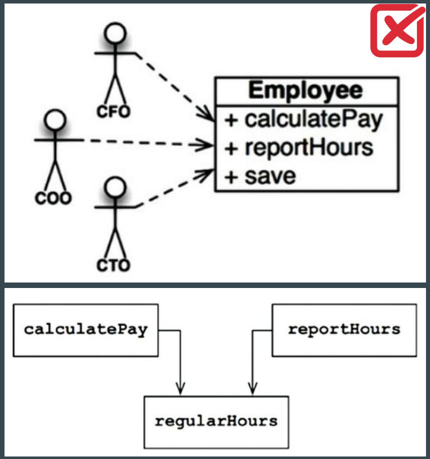


Image from the Book "Clean Architecture"

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

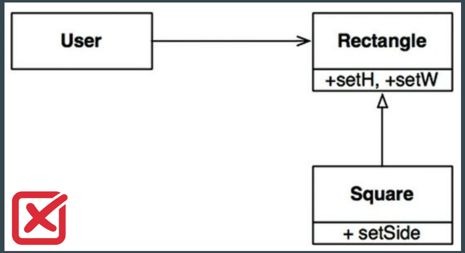
- The Java Collections Framework, specifically the List interface and its various implementations (e.g., ArrayList, LinkedList, Vector, etc.), exemplify LSP.

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.

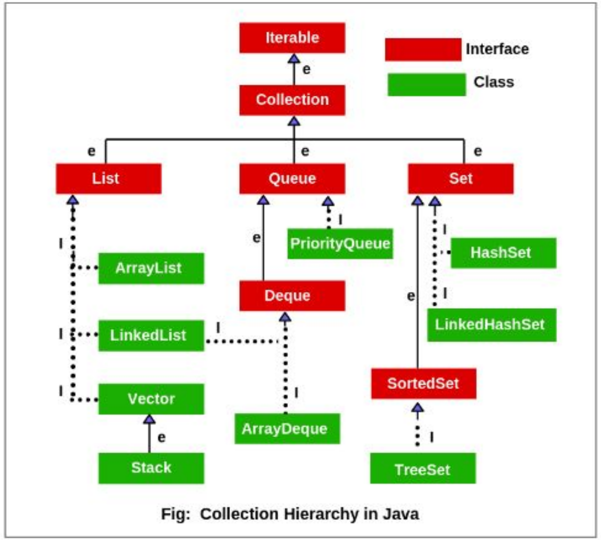


Fig: Collection Hierarchy in Java

e→ extends, I→ implements

Image Source

The Dependency Inversion Principle (DIP)

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

Rationale

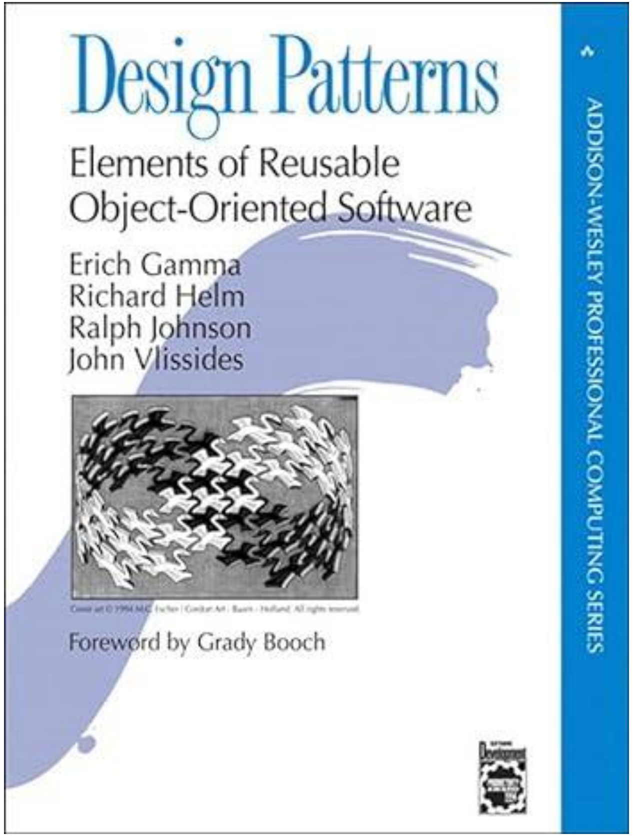
- DIP addresses the issues of tight coupling and rigidity in software design.
- High-level modules can work with different low-level modules that adhere to the same abstractions, and low-level modules can be replaced or modified without affecting the high-level modules.

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.