Java Patterns

Part I: Java Language

Object-Oriented Analysis and Design

- Classes, interfaces, and abstract classes
- Inheritance vs. composition
- SOLID principles
- GRASP patterns
- UML class diagrams
- Domain modeling

Functional Programming and Streams

- Lambda expressions and functional interfaces
- Method references
- Stream creation, intermediate, and terminal ops
- Collectors and custom reductions
- Parallel streams and performance
- Function composition and higher-order functions

Data-Oriented Programming

- Records as transparent data carriers
- Sealed classes and interfaces
- Pattern matching for instanceof and switch
- Deconstructing records
- Algebraic data types in Java

- Combining OOP and data-oriented styles

Part II: Design Patterns

Creational Patterns

- Singleton (enum, thread-safe, DI alternatives)
- Factory Method and Abstract Factory
- Builder (classic, fluent, Lombok, records)
- Prototype (cloning, deep vs. shallow copy)
- Object Pool (connections, threads, HikariCP)

Structural Patterns

- Adapter (class, object, legacy integration)
- Decorator (dynamic behavior, I/O streams)
- Proxy (virtual, dynamic, Spring AOP)
- Facade (API simplification)
- Composite (tree structures)
- Bridge (abstraction from implementation)
- Flyweight (string interning, integer caching)

Behavioral Patterns

- Strategy (algorithm families, lambdas)
- Observer (event-driven, reactive)
- Command (undo/redo, queues)
- Iterator (collections, streams)
- Template Method (abstract hierarchies, hooks)
- Chain of Responsibility (filters, logging)
- State (state machines, enum-based)

- Visitor (double dispatch, AST traversal)
- Mediator (coordination, message brokers)
- Memento (state restoration, event sourcing)
- Interpreter (expression evaluation)

Functional Patterns

- Function composition
- Higher-order functions
- Currying and partial application
- Monads (Optional, Stream, CompletableFuture)
- Immutability patterns
- Pure functions and side effects
- Lazy evaluation
- Functional error handling

Concurrent Patterns

- Thread-safe Singleton
- Thread pools and Executors
- Producer-Consumer
- Read-Write locks
- Future and CompletableFuture
- Fork/Join framework
- Virtual threads (Java 21+)
- Structured concurrency

Part III: Core APIs

Lists and Arrays

- Array fundamentals and manipulation
- ArrayList internals and growth strategy
- LinkedList and when to use it
- List.of() and immutable lists
- Sorting, searching, and iteration patterns
- Array vs. List trade-offs

HashMap and HashSet

- HashMap internals (buckets, tree bins)
- hashCode/equals contract
- Load factor and capacity tuning
- HashSet as HashMap wrapper
- LinkedHashMap for insertion order
- TreeMap and TreeSet for sorted access
- ConcurrentHashMap for thread safety

Collections

- Queue and Deque implementations
- PriorityQueue and ordering
- Collections utility methods
- Unmodifiable and synchronized wrappers
- Concurrent collections overview
- Choosing the right collection

Generics and Boxing

- Type parameters and bounds

- Wildcards (? extends, ? super, PECS)
- Type erasure and its consequences
- Autoboxing and unboxing costs
- Primitive specializations
- Generic methods and type inference

Java Time API

- LocalDate, LocalTime, LocalDateTime
- ZonedDateTime and timezone handling
- Instant and Duration
- Period and temporal adjusters
- Formatting and parsing (DateTimeFormatter)
- Interop with legacy Date/Calendar

Internationalization

- Locale and resource bundles
- MessageFormat and pluralization
- NumberFormat and currency
- Date/time formatting per locale
- Collation and sorting
- Unicode and character encoding

String Processing

- String immutability and interning
- StringBuilder vs. StringBuffer
- Regular expressions (Pattern, Matcher)
- String formatting and text blocks
- Character encoding (UTF-8, charsets)
- String performance patterns

Part IV: Concurrency

Threading

- Thread lifecycle and states
- synchronized and volatile
- wait/notify and condition variables
- Executor framework and thread pools
- Callable, Future, CompletableFuture
- Virtual threads (Project Loom)
- Structured concurrency

Lock-Free Concurrency

- Atomic classes (AtomicInteger, AtomicReference)
- Compare-and-swap (CAS) operations
- Lock-free queues and stacks
- StampedLock and optimistic reads
- Adders and accumulators
- Memory ordering and happens-before

Part V: I/O and Error Handling

Optional and Error Handling

- Optional creation and chaining
- map, flatMap, filter, orElse variants
- Anti-patterns to avoid
- Checked vs. unchecked exceptions
- Try-with-resources
- Custom exception hierarchies

- Functional error handling (Either, Try, Result)

I/O, Files, and NIO

- Classic streams and readers/writers
- Buffering strategies
- NIO.2 Files and Paths
- Walking file trees and WatchService
- Channels, ByteBuffers, and Selectors
- Asynchronous file I/O
- Memory-mapped files

Part VI: Advanced Topics

Reflection and Annotations

- Class introspection and dynamic invocation
- Dynamic proxy creation
- Custom annotations and retention policies
- Runtime and compile-time annotation processing
- MethodHandles as fast alternative
- Frameworks using annotations (JPA, Spring)

Logging and Debugging

- SLF4J facade and binding architecture
- Logback and Log4j2 configuration
- Log levels and best practices
- Structured logging (JSON)
- MDC and correlation IDs
- Performance considerations

Testing

- JUnit 5 (assertions, parameterized, extensions)
- Mockito (mock, spy, stubbing, verification)
- AssertJ fluent assertions
- Testcontainers for databases and services
- Spring Boot testing (@SpringBootTest, MockMvc)
- Property-based testing (jqwik)

Security

- Authentication vs. authorization
- Spring Security (filters, OAuth2, JWT)
- Java Cryptography Architecture
- Input validation and sanitization
- SQL injection and XSS prevention
- CSRF protection

Foreign Function and Memory

- MemorySegment and MemoryLayout
- Arena-based memory management
- Linker and function descriptors
- Calling C libraries from Java
- Structured access to native memory
- Migration from JNI and Unsafe

Math and Vector API

- Vector species and shapes
- Lane-wise operations
- Masks and reductions
- Math class utilities
- StrictMath and floating-point precision

- Performance benchmarks

Networking and HTTP

- Socket and ServerSocket basics
- HttpClient (Java 11+)
- HTTP/2 and WebSocket support
- Synchronous and asynchronous requests
- Request/response builders
- Connection management and timeouts

SQL and JDBC

- Connection management and DataSource
- PreparedStatement and batch operations
- ResultSet processing patterns
- Transaction handling
- Connection pooling (HikariCP)
- SQL injection prevention

Part VII: JVM Internals

Bytecode and JVM

- Class file structure
- Bytecode instruction set
- Stack-based execution model
- Class loading and delegation
- JIT compilation (C1, C2, tiered)
- Inlining and escape analysis
- Bytecode manipulation (Byte Buddy, ASM)

Memory Management

- JVM memory areas (heap, stack, metaspace)
- Object header and layout
- Allocation and TLAB
- Strong, weak, soft, phantom references
- Memory leaks and detection
- Off-heap memory (DirectByteBuffer)
- Object pooling and flyweight

Garbage Collectors

- Generational hypothesis
- Serial and Parallel GC
- G1 GC (regions, mixed collections)
- ZGC (sub-millisecond pauses)
- Shenandoah GC
- GC tuning (-Xms, -Xmx, flags)
- GC log analysis and monitoring

Concurrency Internals

- Java Memory Model (JMM)
- Happens-before relationships
- volatile, synchronized, and final semantics
- Lock internals (biased, thin, fat locks)
- AQS (AbstractQueuedSynchronizer)
- ForkJoinPool internals
- Virtual thread implementation

Performance Optimization

- Profiling (JFR, async-profiler, VisualVM)
- JMH microbenchmarking

- CPU optimization (algorithms, data locality)
- Memory optimization (pooling, off-heap)
- I/O optimization (buffering, async)
- String and collection performance
- Performance anti-patterns
- Performance testing (JMeter, Gatling)

JVM Tools and Options

- jcmd (unified diagnostic command)
- jstack (thread dumps)
- jmap (heap dumps, histogram)
- jstat (GC and class loading stats)
- Java Flight Recorder (jfr)
- jconsole and JMX
- Essential JVM flags (-XX:+PrintGCDetails, etc.)
- Container-aware JVM settings

Functional Programming

Java's embrace of functional programming, beginning with Java 8, fundamentally changed how we think about solving problems. While Java will never be a pure functional language like Haskell, adopting functional principles leads to code that is more predictable, testable, and maintainable. This chapter explores functional thinking and its practical application through the Stream API.

2.1 Principles of Functional Programming

Functional programming is built on three fundamental principles:

1. **Immutability**: Data structures that cannot be modified after creation. Instead of changing an object, you create a new one.

2. **Pure Functions**: Functions whose output depends only on their input parameters, with no side effects.

3. **First-Class Functions**: Functions that can be treated as values—passed as arguments, returned from other functions, and stored in variables.

2.2 Immutability

Thread Safety Without Locks Immutable objects can be shared freely between threads without synchronization. There are no race conditions because there's nothing to race for—the object cannot change.

Counter Example

Look at the difference between mutable and immutable counters. The mutable version requires synchronized methods to prevent race conditions when multiple threads access it. The immutable version sidesteps these threading issues entirely by returning a new instance on each increment.

```
// Mutable: requires synchronization
public class MutableCounter {
  private int count = 0;

  public synchronized void increment() {
    count++;
  }

  public synchronized int getCount() {
    return count;
  }
}

// Immutable: no synchronization needed
public record ImmutableCounter(int count) {
  public ImmutableCounter increment() {
    return new ImmutableCounter(count + 1);
  }
}
```

Process Steps Example

When processing an order through multiple stages, the mutable approach leaves you wondering what each method might have changed. You cannot know without reading the implementation. The immutable approach makes transformations explicit—each step produces a new object, showing exactly what changed.

```java
// Mutable: need to worry about changes
public void processOrder(Order order) {
  calculateTotal(order);
  // Did calculateTotal modify the order? Who knows!
  applyDiscount(order);
  // What about this one? Better check the implementation...
  saveOrder(order);
}

// Immutable: clear data flow
public void processOrder(Order order) {
  Order withTotal = calculateTotal(order);
  Order withDiscount = applyDiscount(withTotal);
  Order finalOrder = saveOrder(withDiscount);
}
```

Simpler Debugging: When tracking down a bug, immutability means you have fewer suspects. An object's state at any point is determined by how it was created, not by the entire execution history.

Better Caching: Immutable objects can be safely cached and reused. Their hash codes never change, making them safe to use as map keys.

Creating Immutable Classes

The simplest way to create immutable classes in modern Java is with records, introduced in Java 14. Records automatically provide private final fields, accessor methods, equals, hashCode, and toString implementations. They enforce immutability by design, making them ideal for value objects.

```java
// A simple immutable person record
public record Person(String name, int age) {
  // Records are automatically immutable:
  // - All fields are private and final
  // - No setters are generated
  // - toString, equals, hashCode are provided
}

// Usage: "modifications" create new instances
Person person = new Person("Alice", 30);
Person older = new Person(person.name(), person.age() + 1);
```

Bank Account Example

A bank account seems like a natural candidate for mutable state, but immutability works beautifully here too. Each transaction returns a new account instance containing the new balance. This approach creates an audit trail and eliminates threading concerns.

```java
public final class BankAccount {
  private final String accountId;
  private final BigDecimal balance;

  public BankAccount(String id, BigDecimal balance) {
    this.accountId = id;
    this.balance = balance;
  }

  public String getAccountId() { return accountId; }
  public BigDecimal getBalance() { return balance; }

  public BankAccount deposit(BigDecimal amount) {
    return new BankAccount(accountId, balance.add(amount));
  }

  public BankAccount withdraw(BigDecimal amount) {
    if (balance.compareTo(amount) < 0) {
      throw new IllegalArgumentException("Insufficient");
    }
    return new BankAccount(
        accountId, balance.subtract(amount));
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof BankAccount that)) return false;
    return accountId.equals(that.accountId) &&
        balance.compareTo(that.balance) == 0;
  }

  @Override
  public int hashCode() {
    return Objects.hash(accountId, balance);
  }
}
```

Rules for creating immutable classes:

1. **Make the class `final`**: Prevents subclasses from adding mutable state
2. **Make all fields `private` and `final`**: Prevents direct access and reassignment
3. **No setters**: Don't provide methods that modify fields
4. **Defensive copying**: For mutable types, copy them in constructors and getters
5. **"Mutation" methods return new instances**: Don't modify, create

Immutable Collections

Java 9 introduced factory methods that create truly immutable collections with minimal syntax. These collections throw UnsupportedOperationException on any attempt to modify them. The factory methods include List.of(), Set.of(), and Map.of() for creating small immutable collections inline.

```java
// Factory methods (Java 9+): Truly immutable
List<String> immutableList = List.of("a", "b", "c");
Set<Integer> immutableSet = Set.of(1, 2, 3);
Map<String, Integer> immutableMap = Map.of(
  "one", 1,
  "two", 2,
  "three", 3
);

// throw UnsupportedOperationException on modification
// immutableList.add("d"); // throws exception

// For larger collections, use copyOf
List<String> original = new ArrayList<>();
original.add("a");
original.add("b");

List<String> immutableCopy = List.copyOf(original);

// Or use ofEntries for maps with many entries
Map<String, Integer> largeMap = Map.ofEntries(
  Map.entry("one", 1),
  Map.entry("two", 2),
  Map.entry("three", 3),
  Map.entry("four", 4)
);
```

Unmodifiable View vs Immutable Copy

There is a crucial distinction between unmodifiable views and truly immutable copies. An unmodifiable view wraps the original collection and reflects any changes made to it. A true immutable copy is independent of the original, providing stronger guarantees about data consistency.

```java
List<String> list = new ArrayList<>();
list.add("a");

// Unmodifiable view: changes to original affect the view
List<String> unmodifiableView =
    Collections.unmodifiableList(list);
list.add("b"); // Original can still be modified
System.out.println(unmodifiableView); // [a, b] - changed!

// True immutable copy: independent of original
List<String> immutableCopy = List.copyOf(list);
list.add("c"); // Original modified
System.out.println(immutableCopy); // [a, b] - unchanged

// Collectors.toUnmodifiableList(): immutable list
List<String> immutableFromStream = list.stream()
  .map(String::toUpperCase)
  .collect(Collectors.toUnmodifiableList());

// stream.toList(): unmodifiable list (Java 16+)
List<String> alsoImmutable = list.stream()
  .map(String::toUpperCase)
  .toList();
```

The Risk of Collections in Records

Records appear immutable because their fields are final, but this is deceiving when those fields reference mutable objects. A mutable list inside a record can still be modified through its reference, breaking the immutability guarantee that records seem to provide.

```java
// UNSAFE: Exposes mutable internal state
public record Team(String name, List<String> members) {
  // Even though this is a record, members can be modified!
}

// The danger:
List<String> developers = new ArrayList<>();
developers.add("Alice");
developers.add("Bob");

Team team = new Team("Backend", developers);
System.out.println(team.members()); // [Alice, Bob]

// Modify the original list
developers.add("Charlie");
System.out.println(team.members()); // [Alice, Bob, Charlie]

// Or modify through the getter
team.members().add("David");
System.out.println(team.members()); // [..., David] added!
```

Solution: Defensive Copy

The fix uses the compact canonical constructor to intercept the incoming list and replace it with an immutable copy. This ensures that external code cannot modify the record’s internal state by holding a reference to the original mutable collection.

```java
public record Team(String name, List<String> members) {
  // Canonical constructor with defensive copy
  public Team {
    // Create an immutable copy
    members = List.copyOf(members);
  }
}

// Usage:
List<String> developers = new ArrayList<>();
developers.add("Alice");
developers.add("Bob");

Team team = new Team("Backend", developers);
developers.add("Charlie"); // Doesn't affect team
System.out.println(team.members()); // [Alice, Bob]

// team.members().add("David"); // UnsupportedOperationEx
```

When to Use Defensive Copying

Legacy mutable types like java.util.Date are particularly dangerous because callers can modify your object’s internal state. Defensive copying protects against this by creating independent copies in constructors and getters. Modern alternatives like java.time.Instant are inherently immutable.

```java
// BAD: Exposes mutable date
public class Event {
  private Date startTime; // Date is mutable!

  public Event(Date startTime) {
    this.startTime = startTime; // Stores reference
  }

  public Date getStartTime() {
    return startTime; // Returns reference
  }
}

Date date = new Date();
Event event = new Event(date);
date.setTime(0); // Modifies the event's date!
event.getStartTime().setTime(0); // Also modifies it!

// GOOD: Defensive copies
public class Event {
  private final Date startTime;

  public Event(Date startTime) {
    this.startTime = new Date(startTime.getTime());
  }

  public Date getStartTime() {
    return new Date(startTime.getTime());
  }
}

// BETTER: Use immutable types
public record Event(Instant startTime) {
  // Instant is immutable, no defensive copying needed
}
```

Best practices for collections in immutable classes:

1. **Prefer immutable collection types**: Use `List.copyOf()`, `Set.copyOf()`, etc.
2. **Defensive copy on input**: Copy collections passed to constructors
3. **Return immutable collections**: Don't return mutable collections from getters

4. **Use immutable types**: When possible, use types that are already immutable (`Instant` instead of `Date` , `BigDecimal` instead of `double` , etc.)

2.3 Pure Functions

A function is pure when it satisfies two conditions:

1. **Deterministic**: The output depends only on the input parameters. Given the same inputs, it always returns the same output.
2. **No side effects**: It doesn't modify external state, perform I/O, or have any observable effect other than computing the return value.

Writing Pure Functions

Pure functions take inputs and produce outputs with nothing else happening. They do not read or modify external state, perform I/O operations, or cause any observable side effects. This predictability makes them easier to understand, test, and compose together.

```java
// PURE: Output depends only on inputs
public static int add(int a, int b) {
  return a + b;
}

public static String formatFullName(String first,
                                    String last) {
  return last + ", " + first;
}

public static List<String> filterLongNames(List<String> names,
                                           int minLength) {
  return names.stream()
    .filter(name -> name.length() >= minLength)
    .toList();
}
```

Impure Functions

Impurity comes in many forms. Functions that read or write shared state, perform I/O, log messages, or generate random values all have side effects. These side effects make functions harder to test and reason about because their behavior depends on external context.

```java
// IMPURE: Depends on external state
private static int counter = 0;

public static int getNextId() {
  return ++counter; // Depends on counter state
}

// IMPURE: Modifies external state (side effect)
private static List<String> log = new ArrayList<>();

public static int multiplyWithLogging(int a, int b) {
  int result = a * b;
  log.add("Multiplied " + a + " and " + b); // Side effect
  return result;
}

// IMPURE: Modifies input parameter (side effect)
public static void addToList(List<String> list, String item) {
  list.add(item); // Modifies the parameter
}

// IMPURE: Performs I/O (side effect)
public static String readFile(String path) throws IOException {
  return Files.readString(Path.of(path)); // I/O operation
}

// IMPURE: Non-deterministic
public static int getRandom() {
  return new Random().nextInt(); // Different each call
}
```

Pure Functions Should Have Return Values

If a function returns void, it must be executing side effects to accomplish anything useful. This includes modifying state, performing I/O, or mutating parameters. A pure function with no return value would compute something and discard it, serving no purpose.

```java
// This can't be pure (void return type)
public void updateUser(User user) {
  // Must either:
  // 1. Modify the user parameter (side effect)
  // 2. Modify external state (side effect)
  // 3. Perform I/O (side effect)
  // Otherwise, it does nothing!
}

// Pure version: returns new user
public User updateUserEmail(User user, String newEmail) {
  return new User(
    user.id(),
    newEmail,
    user.name(),
    user.age()
  );
}
```

Benefits of Pure Functions

1. Easy to Test

Testing pure functions requires no setup, no mocking frameworks, and no teardown. You simply provide inputs and verify the outputs. There are no external dependencies to simulate, no state to reset, and no ordering constraints between tests to worry about.

```java
// Pure function: no setup, no mocks, just input and output
public static BigDecimal calculateDiscount(BigDecimal price,
                                           double rate) {
  return price.multiply(BigDecimal.valueOf(rate));
}

@Test
void testCalculateDiscount() {
  BigDecimal price = new BigDecimal("100.00");
  BigDecimal discount = calculateDiscount(price, 0.15);
  assertEquals(new BigDecimal("15.00"), discount);
  // No mocks, no setup, no teardown
}
```

2. Safe to Parallelize

Pure functions have no shared state, which means they can run concurrently without any coordination between threads. Multiple calls can execute simultaneously without locks, synchronization primitives, or careful ordering. This makes them ideal candidates for parallel stream operations.

```java
// Pure function: safe to call from multiple threads
public static List<Integer> transform(List<Integer> nums) {
  return nums.parallelStream() // Safe parallel stream
    .map(n -> n * n)
    .filter(n -> n > 100)
    .toList();
}

// Impure function: NOT safe to parallelize
private static int sum = 0;

public static void addToSum(int value) {
  sum += value; // Race condition!
}
```

3. Can Be Memoized

Because pure functions always return the same output for the same input, you can cache their results. This technique, called memoization, trades memory for computation time. Expensive calculations only run once per unique input, with subsequent calls returning cached values.

```
// Pure function: results can be cached
public class PureFunctions {
  private static final Map<Integer, Integer> fibCache =
      new ConcurrentHashMap<>();

  public static int fibonacci(int n) {
    return fibCache.computeIfAbsent(n, k -> {
      if (k <= 1) return k;
      return fibonacci(k - 1) + fibonacci(k - 2);
    });
  }
}
```

Separating Pure and Impure

In real applications, you cannot avoid side effects entirely. The key insight is to push I/O operations to the boundaries of your system and keep the core logic pure. This creates a testable core wrapped by a thin impure shell.

```java
// Pure core: all business logic
public class OrderProcessor {
  public static record ProcessingResult(
      Order processedOrder,
      List<String> notifications,
      List<AuditEvent> auditEvents
  ) {}

  // Pure function: returns what should happen
  public static ProcessingResult processOrder(
      Order order,
      Customer customer,
      List<Product> products,
      DiscountRules rules) {

    // All business logic here, pure calculations
    BigDecimal subtotal =
        calculateSubtotal(order, products);
    BigDecimal discount =
        rules.calculateDiscount(customer, subtotal);
    BigDecimal total = subtotal.subtract(discount);

    Order processed =
        order.withTotal(total).withDiscount(discount);

    var notifications = List.of(
        "Order confirmed: " + order.id(),
        "Total: $" + total
    );

    var auditEvents = List.of(
        new AuditEvent("ORDER_CREATED", order.id()),
        new AuditEvent("DISCOUNT_APPLIED", discount + "")
    );

    return new ProcessingResult(
        processed, notifications, auditEvents);
  }
}

// Impure shell: I/O operations
public class OrderService {
  private final OrderRepository orderRepo;
  private final CustomerRepository customerRepo;
  private final ProductRepository productRepo;
  private final EmailService emailService;
  private final AuditLog auditLog;
```

```java
    public void submitOrder(String orderId) {
      // Impure: fetch all data
      Order order = orderRepo.findById(orderId);
      Customer cust = customerRepo.findById(order.customerId());
      var prods = productRepo.findByIds(order.getProductIds());
      DiscountRules rules = customerRepo.getDiscountRules();

      // Pure: process
      var result = OrderProcessor.processOrder(
          order, cust, prods, rules);

      // Impure: execute effects
      orderRepo.save(result.processedOrder());

      for (String notif : result.notifications()) {
        emailService.send(cust.email(), notif);
      }

      for (AuditEvent event : result.auditEvents()) {
        auditLog.record(event);
      }
    }
  }
```

2.4 First Class Functions

A language has first-class functions when functions can be:

1. **Stored in variables**
2. **Passed as arguments to other functions**
3. **Returned as values from other functions**
4. **Stored in data structures** (lists, maps, etc.)

In Java, we achieve this through **functional interfaces** and **lambda expressions**.

```java
// 1. Storing functions in variables
Function<String, Integer> stringLength = s -> s.length();
Predicate<Integer> isEven = n -> n % 2 == 0;
Consumer<Object> printer = System.out::println;

// 2. Passing functions as arguments
public <T> List<T> filter(
    List<T> list, Predicate<T> predicate) {
  return list.stream().filter(predicate).toList();
}

List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);
List<Integer> evens = filter(numbers, isEven);

// 3. Returning functions from methods
public Predicate<Integer> greaterThan(int threshold) {
  return n -> n > threshold;
}

Predicate<Integer> greaterThan10 = greaterThan(10);
boolean result = greaterThan10.test(15); // true

// 4. Storing functions in data structures
Map<String, Function<Integer, Integer>> ops = Map.of(
  "double", n -> n * 2,
  "square", n -> n * n,
  "negate", n -> -n
);

Function<Integer, Integer> op = ops.get("square");
int squared = op.apply(5); // 25
```

Lambda Syntax

Lambda expressions come in several forms depending on the number of parameters and complexity of the body. Single-parameter lambdas can omit parentheses. Multi-statement bodies require braces and an explicit return. Type annotations are optional when the compiler can infer them.

```java
// No parameters
Runnable r = () -> System.out.println("Hello");

// One parameter (parentheses optional)
Function<String, Integer> length = s -> s.length();
Function<String, Integer> length2 = (s) -> s.length();

// Multiple parameters
BinaryOperator<Integer> add = (a, b) -> a + b;

// Explicit types (optional, usually inferred)
BinaryOperator<Integer> multiply =
    (Integer a, Integer b) -> a * b;

// Block body for multiple statements
Function<List<String>, String> concat = list -> {
  StringBuilder sb = new StringBuilder();
  for (String s : list) {
    sb.append(s);
  }
  return sb.toString();
};

// Block body with explicit return
IntUnaryOperator square = n -> {
  int result = n * n;
  return result;
};
```

Functional Interfaces (SAM)

A functional interface declares exactly one abstract method, making it a target for lambda expressions. The @FunctionalInterface annotation documents this intent and triggers a compile error if you accidentally add a second abstract method. Default and static methods are permitted.

```java
// This is a functional interface
@FunctionalInterface
public interface Transformer<T> {
  T transform(T input);

  // Default and static methods are allowed
  default Transformer<T> andThen(Transformer<T> next) {
    return input -> next.transform(this.transform(input));
  }

  static <T> Transformer<T> identity() {
    return input -> input;
  }
}

// Lambda implements the functional interface
Transformer<String> uppercase = s -> s.toUpperCase();
Transformer<String> trim = String::trim;

// Compose transformers
Transformer<String> cleanAndCapitalize =
    trim.andThen(uppercase);
String result = cleanAndCapitalize.transform("  hello  ");
```

How Lambdas Work

Lambdas are not anonymous inner classes despite their similar syntax. They use the invokedynamic bytecode instruction, introduced in Java 7 for dynamic language support. This approach avoids creating a separate class file for each lambda, resulting in better performance and smaller memory footprint.

```java
// This lambda
Function<String, Integer> length = s -> s.length();

// Is NOT compiled to an anonymous inner class
// Instead, the compiler:
// 1. Creates a synthetic method with the lambda body
// 2. Uses invokedynamic to lazily create interface
// 3. That instance delegates to the synthetic method
```

Advantages:

- **No class file per lambda**: Better performance and less memory
- **JVM can optimize**: More flexibility for the runtime
- **Stateless lambdas can be cached**: Same lambda reused

Closures and Variable Capture

Lambdas can access variables from their enclosing scope, creating what is known as a closure. However, Java requires these captured variables to be effectively final. This restriction prevents subtle concurrency bugs that would arise from modifying shared state.

```
public void example() {
    String prefix = "Hello, ";  // Effectively final
    int multiplier = 10;        // Effectively final
    int count = 0;              // NOT effectively final

    // This works: captures effectively final variables
    Function<String, String> greeter = name -> prefix + name;
    Function<Integer, Integer> mult = n -> n * multiplier;

    // This won't compile: count is not effectively final
    // Supplier<Integer> counter = () -> count;

    count++;  // Modifying count makes previous line fail
}
```

Instance Fields Are Different

The effectively-final restriction applies only to local variables, not to instance or class fields. Instance fields can be modified inside lambdas because they are accessed through the this reference, which is implicitly final. However, this reintroduces potential concurrency issues.

```java
public class Counter {
  private int count = 0;  // Instance field

  public Runnable createIncrementer() {
    // This is allowed: can modify instance fields
    return () -> count++;
  }

  public Runnable createLogger() {
    // Can also read instance fields
    return () -> System.out.println("Count: " + count);
  }
}
```

Method References

Method references provide a concise syntax when a lambda would simply delegate to an existing method. They often make code more readable by using the method name directly. There are four kinds: static methods, instance methods on specific objects, arbitrary instance methods, and constructors.

```java
// 1. Static method reference
Function<String, Integer> parser = Integer::parseInt;
// Equivalent to: s -> Integer.parseInt(s)

BinaryOperator<Integer> max = Math::max;
// Equivalent to: (a, b) -> Math.max(a, b)

// 2. Instance method reference on a specific object
String prefix = "Hello, ";
UnaryOperator<String> greeter = prefix::concat;
// Equivalent to: s -> prefix.concat(s)

List<String> list = new ArrayList<>();
Consumer<String> adder = list::add;
// Equivalent to: s -> list.add(s)

// 3. Instance method on arbitrary object of a type
Function<String, String> upper = String::toUpperCase;
// Equivalent to: s -> s.toUpperCase()

Function<String, Integer> length = String::length;
// Equivalent to: s -> s.length()

BiPredicate<String, String> equals = String::equals;
// Equivalent to: (s1, s2) -> s1.equals(s2)

// 4. Constructor reference
Supplier<List<String>> listFactory = ArrayList::new;
// Equivalent to: () -> new ArrayList<>()

IntFunction<List<String>> listWithSize = ArrayList::new;
// Equivalent to: size -> new ArrayList<>(size)

Function<String, User> userFactory = User::new;
// Equivalent to: name -> new User(name)
```

When to Use Method References

Method references shine when they make the code's intent clearer. They work best for simple delegations to existing methods. However, when additional logic or parameter manipulation is needed, a lambda expression remains the better choice for maintaining readability and clarity.

```
// Good: clear and concise
list.stream()
  .map(String::toUpperCase)
  .forEach(System.out::println);

// Method reference might be less clear here
list.stream().map(User::getName)    // Clear: getting name
// vs
list.stream().map(u -> processName(u.getName())) // Need λ
```

2.5 Optional

Optional provides three factory methods for three possible states:

```
// With a value
var present = Optional.of("hello");

// Might be empty (handles null safely)
var maybe = Optional.ofNullable(possiblyNullValue);

// Empty
var empty = Optional.empty();
```

Functional Operations

Optional becomes powerful when you chain its functional methods. Rather than checking for presence with conditional statements, you can transform, filter, and combine Optional values using map, flatMap, and filter. This leads to more declarative code with fewer null checks.

```
// map: Transform the value if present
var name = Optional.of("alice");
var length = name.map(String::length);
var upper = name.map(String::toUpperCase);

// flatMap: Avoid Optional<Optional<T>>
var user = findUser(userId);
var email = user.flatMap(User::getEmail);

// filter: Keep only if predicate matches
var longName = name.filter(s -> s.length() > 5);

// ifPresent: Execute action if present
name.ifPresent(n -> System.out.println("Hello, " + n));

// ifPresentOrElse: Execute one of two actions
name.ifPresentOrElse(
  n -> System.out.println("Found: " + n),
  () -> System.out.println("Not found")
);

// orElse: Provide default
String result = name.orElse("default");

// orElseGet: Provide default via supplier (lazy)
String result2 = name.orElseGet(() -> computeDefault());

// orElseThrow: Throw if empty
String result3 = name.orElseThrow();
String result4 = name.orElseThrow(
        () -> new IllegalStateException("Required"));
```

Optional Uses

Optional works best as a return type that signals possible absence. Do not use Optional as method parameters because callers must still handle the possibility of null being passed. Method overloading or separate methods provide cleaner APIs for optional inputs.

```java
// GOOD
public Optional<User> findUser(String id) { /* ... */ }

// BAD
public void setUser(Optional<User> user) { /* ... */ }

// BETTER: use overloading
public void setUser(User user) { /* ... */ }
public void clearUser() { /* ... */ }
```

2.6 Java Streams

A stream is **not** a data structure—it's a sequence of elements from a source that supports aggregate operations:

- **No storage**: Streams don't store elements; they convey elements from a source
- **Functional**: Operations produce results but don't modify the source
- **Lazy**: Intermediate operations are not executed until a terminal operation is invoked
- **Possibly unbounded**: Streams can represent infinite sequences
- **Consumable**: Elements are visited only once

```java
// Stream vs Collection
List<String> list = List.of("a", "b", "c");  // Stores elements
var stream = list.stream();  // Views elements, doesn't store

// Stream operations don't modify the source
var names = new ArrayList<>(List.of("alice", "bob"));
names.stream()
  .map(String::toUpperCase)
  .forEach(System.out::println);  // Prints "ALICE", "BOB"

System.out.println(names);  // Still ["alice", "bob"]
```

Stream Creation

```java
// From collections
List<String> list = List.of("a", "b", "c");
Stream<String> stream = list.stream();
Stream<String> parallel = list.parallelStream();

// From arrays
String[] array = {"a", "b", "c"};
Stream<String> streamFromArray = Arrays.stream(array);
var streamFromRange = Arrays.stream(array, 1, 3); // 1, 2

// From values
Stream<String> streamOfValues = Stream.of("a", "b", "c");
Stream<Integer> singleElement = Stream.of(42);

// From builder
Stream<String> built = Stream.<String>builder()
  .add("a")
  .add("b")
  .add("c")
  .build();

// Empty stream
Stream<String> empty = Stream.empty();

// Infinite streams
Stream<Integer> naturals = Stream.iterate(0, n -> n + 1);
Stream<Integer> evens = Stream.iterate(0, n -> n + 2);
Stream<Double> randoms = Stream.generate(Math::random);

// From files (auto-closeable)
try (var lines = Files.lines(Path.of("file.txt"))) {
  lines.forEach(System.out::println);
}

// Primitive streams (better performance)
IntStream intRange = IntStream.range(0, 10);      // 0-9
IntStream closed = IntStream.rangeClosed(0, 10); // 0-10
LongStream longStream = LongStream.of(1L, 2L, 3L);
DoubleStream doubles = DoubleStream.of(1.0, 2.0, 3.0);
```

Intermediate Operations (Lazy)

Intermediate operations return a new stream and are lazy—they are not executed until a terminal operation is called. This laziness enables optimizations like short-circuiting, where the pipeline stops processing as soon as the result can be determined without examining all elements.

filter and map

The filter operation selects elements matching a predicate, keeping only those for which the predicate returns true. The map operation transforms each element by applying a function, producing a stream of transformed values. These two operations form the foundation of most pipelines.

```
var names = List.of("Alice", "Bob", "Charlie", "David");

// filter: Select elements matching a predicate
names.stream().filter(s -> s.length() > 4);  // Alice...

// map: Transform each element
names.stream().map(String::length);      // 5, 3, 7, 5
names.stream().map(String::toUpperCase); // ALICE, BOB...
```

flatMap

The flatMap operation transforms each element into a stream, then concatenates all those streams into a single flat stream. This is essential for working with nested collections, converting a stream of lists into a single stream containing all elements.

```
var nested = List.of(List.of("a", "b"), List.of("c", "d"));
nested.stream().flatMap(List::stream);  // a, b, c, d

// flatMap strings to characters
Stream.of("hello", "world")
  .flatMap(w -> w.chars().mapToObj(c -> (char) c));
```

distinct and sorted

The distinct operation removes duplicates based on the equals method. The sorted operation orders elements either by their natural ordering or using a custom comparator. Both are stateful operations that must process multiple elements before producing output, unlike filter or map.

```
Stream.of("a", "b", "a", "c", "b").distinct();  // a, b, c

names.stream().sorted();  // Alice, Bob... (natural order)
names.stream()
  .sorted(Comparator.comparingInt(String::length));
names.stream().sorted(Comparator.comparingInt(String::length)
  .thenComparing(String::compareTo));  // Secondary sort
```

limit, skip, takeWhile, dropWhile

These operations extract portions of a stream based on position or condition. The limit and skip operations work by element count, while takeWhile and dropWhile use predicates to determine where to cut. TakeWhile stops at the first non-matching element; dropWhile starts there.

```
names.stream().limit(3);  // First 3: Alice, Bob, Charlie
names.stream().skip(2);   // Skip first 2: Charlie, David

// Java 9+: conditional slicing
Stream.of(1, 2, 3, 4, 5, 6, 7)
  .takeWhile(n -> n < 5);  // 1, 2, 3, 4
Stream.of(1, 2, 3, 4, 5, 6, 7)
  .dropWhile(n -> n < 5);  // 5, 6, 7
```

peek

The peek operation performs a side-effect action on each element, primarily intended for debugging. It lets you observe values as they flow through the pipeline without modifying the stream. Avoid using peek for production logic since side effects make streams harder to reason about.

```java
names.stream()
  .peek(s -> System.out.println("Before: " + s))
  .map(String::toUpperCase)
  .peek(s -> System.out.println("After: " + s))
  .toList();
```

Terminal Operations (Eager)

Terminal operations trigger stream processing and produce a result or side effect. Unlike intermediate operations, they are eager and execute the entire pipeline. A stream can only have one terminal operation, and after it executes, the stream cannot be reused.

collect and toList

The collect operation uses a Collector to accumulate stream elements into a container. Java 16 introduced toList as a convenient shortcut that returns an unmodifiable list. For mutable lists or other collection types, use Collectors.toList, Collectors.toSet, or other specialized collectors.

```java
List<String> names = List.of("Alice", "Bob", "Charlie");

// collect to list (mutable)
List<String> list = names.stream()
  .map(String::toUpperCase)
  .collect(Collectors.toList());

// toList (Java 16+, immutable)
List<String> immutable = names.stream()
  .map(String::toUpperCase)
  .toList();

// collect to set (removes duplicates)
Set<String> set = names.stream().collect(Collectors.toSet());
```

forEach and forEachOrdered

The forEach operation executes an action for each element in the stream. For parallel streams, the order of execution is undefined. Use forEachOrdered when you need to maintain encounter order, though this may reduce the benefits of parallel processing significantly.

```
names.stream().forEach(System.out::println);

names.parallelStream()
  .map(String::toUpperCase)
  .forEachOrdered(System.out::println);  // Keeps order
```

reduce

The reduce operation combines all stream elements into a single value by repeatedly applying a binary operator. It can optionally take an identity value as the starting point. Without an identity, it returns an Optional since the stream might be empty.

```
// Without identity (returns Optional)
Optional<String> joined = names.stream()
  .reduce((a, b) -> a + ", " + b);  // "Alice, Bob..."

// With identity (returns value directly)
String prefixed = names.stream()
  .reduce("Names:", (a, n) -> a + " " + n);

// Sum with reduce
int sum = IntStream.of(1, 2, 3, 4, 5)
  .reduce(0, Integer::sum);
int sum2 = IntStream.of(1, 2, 3, 4, 5).sum();  // Shortcut
```

count and Match Operations

```java
long cnt = names.stream()
  .filter(s -> s.length() > 4).count();  // 2

boolean hasShort = names.stream()
  .anyMatch(s -> s.length() < 4);
boolean allLong = names.stream()
  .allMatch(s -> s.length() > 2);
boolean noneVeryLong = names.stream()
  .noneMatch(s -> s.length() > 10);
```

findFirst and findAny

```java
Optional<String> first = names.stream()
  .filter(s -> s.startsWith("C"))
  .findFirst();  // Optional["Charlie"]

Optional<String> any = names.parallelStream()
  .filter(s -> s.startsWith("B"))
  .findAny();  // Any match (faster in parallel)
```

min, max, and toArray

```java
Optional<String> shortest = names.stream()
  .min(Comparator.comparingInt(String::length)); // "Bob"

Optional<String> longest = names.stream()
  .max(Comparator.comparingInt(String::length)); // "Charlie"

String[] array = names.stream().toArray(String[]::new);
```

Practical Example: Data Transformation

```java
public record Product(
    String id, String name, BigDecimal price,
    String category, int stock) {}

var bd = (Function<String, BigDecimal>) BigDecimal::new;
List<Product> products = List.of(
  new Product("P1", "Laptop", bd.apply("999.99"), "Elec", 5),
  new Product("P2", "Mouse", bd.apply("19.99"), "Elec", 50),
  new Product("P3", "Desk", bd.apply("299.99"), "Furn", 10),
  new Product("P4", "Chair", bd.apply("149.99"), "Furn", 0),
  new Product("P5", "Monitor", bd.apply("399.99"), "Elec", 8)
);

// In-stock electronics under $500, sorted by price
List<Product> affordable = products.stream()
  .filter(p -> p.category().equals("Elec"))
  .filter(p -> p.stock() > 0)
  .filter(p -> p.price().compareTo(new BigDecimal("500")) < 0)
  .sorted(Comparator.comparing(Product::price))
  .toList();
// Result: [Mouse $19.99, Monitor $399.99]

// Total value of all inventory
BigDecimal totalValue = products.stream()
  .map(p -> p.price().multiply(BigDecimal.valueOf(p.stock())))
  .reduce(BigDecimal.ZERO, BigDecimal::add);

// Most expensive product in each category
var mostExpensive = products.stream()
  .collect(Collectors.groupingBy(
    Product::category,
    Collectors.maxBy(Comparator.comparing(Product::price))
  ));

// Names of out-of-stock items
List<String> outOfStock = products.stream()
  .filter(p -> p.stock() == 0)
  .map(Product::name)
  .toList();  // ["Chair"]
```

Collectors

Essential Collection and Joining

```
var names = List.of("Alice", "Bob", "Charlie", "David");

List<String> toList = names.stream().toList();
Set<String> toSet = new HashSet<>(names);
var joined = names.stream()
  .collect(Collectors.joining(", "));  // "Alice, Bob..."
var formatted = names.stream()
  .collect(Collectors.joining(", ", "[", "]")); // "[...]"
```

groupingBy

The groupingBy collector partitions stream elements into groups based on a classification function. The result is a Map where keys are the classification values and values are lists of elements in each group. Downstream collectors can further process each group.

```
// Group by length
Map<Integer, List<String>> byLength = names.stream()
  .collect(Collectors.groupingBy(String::length));
// {3=[Bob], 5=[Alice, David], 7=[Charlie]}

// Count per group
var countByLength = names.stream()
  .collect(Collectors.groupingBy(
    String::length, Collectors.counting()));
// {3=1, 5=2, 7=1}

// Join per group
var joinedByLength = names.stream()
  .collect(Collectors.groupingBy(
    String::length, Collectors.joining(", ")));
// {3="Bob", 5="Alice, David", 7="Charlie"}
```

partitioningBy

The partitioningBy collector splits elements into exactly two groups based on a predicate. Unlike groupingBy, it always produces a Map with two entries: true and false. This is useful for binary classifications like separating valid from invalid items or passing from failing tests.

```
Map<Boolean, List<String>> partitioned = names.stream()
  .collect(Collectors.partitioningBy(s -> s.length() > 4));
// {false=[Bob], true=[Alice, Charlie, David]}
```

toMap

The toMap collector transforms stream elements into key-value pairs in a Map. You provide functions to extract keys and values from each element. A merge function handles duplicate keys. Without a merge function, duplicate keys throw an IllegalStateException, so plan accordingly.

```
// Simple key-value mapping
Map<String, Integer> nameToLength = names.stream()
  .collect(Collectors.toMap(n -> n, String::length));

// Handle duplicate keys with merge function
Map<Integer, String> lengthToName = names.stream()
  .collect(Collectors.toMap(
    String::length,
    n -> n,
    (existing, replacement) -> existing + ", " + replacement
  ));
// {3="Bob", 5="Alice, David", 7="Charlie"}
```

summarizingInt and collectingAndThen

```
IntSummaryStatistics stats = names.stream()
  .collect(Collectors.summarizingInt(String::length));
// stats.getCount(), getSum(), getMin(), getMax()

String result = List.of(1, 2, 3).stream()
  .collect(Collectors.collectingAndThen(
    Collectors.toList(),
    list -> "Count: " + list.size() + ", Items: " + list
  ));
```

teeing (Java 12+)

The teeing collector applies two collectors simultaneously to the same stream, then combines their results using a merger function. This avoids iterating twice when you need two different aggregations. It was introduced in Java 12 to simplify common multi-aggregation patterns.

```
record Stats(double average, long count) {}

public static Stats computeStats(List<Integer> numbers) {
  return numbers.stream()
      .collect(Collectors.teeing(
        Collectors.averagingInt(Integer::intValue),
        Collectors.counting(),
        Stats::new
      ));
}

var stats = computeStats(List.of(10, 20, 30, 40, 50));
// Stats[average=30.0, count=5]
```

Custom Collectors

When built-in collectors do not fit your needs, Collector.of lets you define custom collection logic. You provide functions for creating the accumulator, adding elements, combining accumulators for parallel execution, and optionally finishing the result into a different type.

```java
public static <T> Collector<T,?,List<List<T>>> toChunks(
    int size) {
  return Collector.of(
    ArrayList::new,
    (list, item) -> {
      if (list.isEmpty() ||
          list.getLast().size() == size) {
        list.add(new ArrayList<>());
      }
      list.get(list.size() - 1).add(item);
    },
    (list1, list2) -> {
      list1.addAll(list2);
      return list1;
    }
  );
}

List<List<Integer>> chunks = IntStream.range(0, 10)
    .boxed()
    .collect(toChunks(3));
// [[0, 1, 2], [3, 4, 5], [6, 7, 8], [9]]
```

Real-World Example: Processing Orders

```
public record Order(
  String orderId,
  String customerId,
  LocalDate orderDate,
  BigDecimal amount,
  OrderStatus status
) {}

public enum OrderStatus {
  PENDING, SHIPPED, DELIVERED, CANCELLED
}

List<Order> orders = getOrders();

// Revenue for delivered orders this month
YearMonth currentMonth = YearMonth.now();
BigDecimal revenue = orders.stream()
  .filter(o -> o.status() == OrderStatus.DELIVERED)
  .filter(o -> YearMonth.from(o.orderDate())
      .equals(currentMonth))
  .map(Order::amount)
  .reduce(BigDecimal.ZERO, BigDecimal::add);

// Top 5 customers by total order amount
Map<String, BigDecimal> customerTotals = orders.stream()
  .collect(Collectors.groupingBy(
    Order::customerId,
    Collectors.reducing(
      BigDecimal.ZERO,
      Order::amount,
      BigDecimal::add
    )
  ));

List<String> topCustomers = customerTotals.entrySet().stream()
  .sorted(Map.Entry.<String, BigDecimal>comparingByValue()
    .reversed())
  .limit(5)
  .map(Map.Entry::getKey)
  .toList();

// Group orders by status and calculate totals
Map<OrderStatus, BigDecimal> byStatus = orders.stream()
  .collect(Collectors.groupingBy(
    Order::status,
    Collectors.reducing(
      BigDecimal.ZERO,
      Order::amount,
      BigDecimal::add
    )
  ));

// Orders pending for > 7 days
LocalDate sevenDaysAgo = LocalDate.now().minusDays(7);
List<Order> oldPending = orders.stream()
  .filter(o -> o.status() == OrderStatus.PENDING)
  .filter(o -> o.orderDate().isBefore(sevenDaysAgo))
  .sorted(Comparator.comparing(Order::orderDate))
  .toList();

// Average order value by month
Map<YearMonth, Double> avgByMonth = orders.stream()
  .collect(Collectors.groupingBy(
    o -> YearMonth.from(o.orderDate()),
    Collectors.averagingDouble(o -> o.amount().doubleValue())
  ));
```

Advanced Stream Patterns

Interleaving Two Streams

Merge two streams by alternating their elements to create an interleaved sequence. This pattern is useful for combining data from two sources in round-robin fashion. When streams have different lengths, the remaining elements from the longer stream appear at the end.

```
public static <T> Stream<T> interleave(
    Stream<T> a, Stream<T> b) {
  Iterator<T> iterA = a.iterator();
  Iterator<T> iterB = b.iterator();

  return Stream.generate(() -> {
    List<T> batch = new ArrayList<>(2);
    if (iterA.hasNext()) batch.add(iterA.next());
    if (iterB.hasNext()) batch.add(iterB.next());
    return batch;
  })
  .takeWhile(batch -> !batch.isEmpty())
  .flatMap(List::stream);
}

Stream<String> s1 = Stream.of("A", "B", "C");
Stream<String> s2 = Stream.of("1", "2", "3", "4");

var result = interleave(s1, s2).toList();
// [A, 1, B, 2, C, 3, 4]
```

Cartesian Product

Generate all combinations from two collections using flatMap to pair each element of the first collection with every element of the second. The result contains every possible pair, with the total count being the product of the two collection sizes.

```
record Pair<A, B>(A first, B second) {}

public static <T> Stream<Pair<T,T>> product(
    List<T> a, List<T> b) {
  return a.stream()
      .flatMap(x -> b.stream().map(y -> new Pair<>(x, y)));
}

List<String> colors = List.of("Red", "Green", "Blue");
List<String> sizes = List.of("S", "M", "L");

var combos = product(colors, sizes).toList();
// 9 combinations: (Red,S), (Red,M), (Red,L)...
```

Sliding Window

Compute statistics over fixed-size overlapping segments of data. Sliding windows are essential for time-series analysis, computing running averages, and detecting patterns in sequential data. Each window overlaps with its neighbors, providing smooth transitions between consecutive computed values in the output.

```java
public static List<Double> movingAverage(
    double[] data, int win) {
  return IntStream.rangeClosed(0, data.length - win)
      .mapToObj(i -> Arrays.copyOfRange(data, i, i + win))
      .map(w -> Arrays.stream(w).average().orElse(0.0))
      .toList();
}

double[] data = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0};
var averages = movingAverage(data, 3); // [2.0, 3.0, 4.0, 5.0]
```

Hierarchical Grouping

Group data by multiple criteria with nested collectors to create hierarchical structures. The outer groupingBy establishes the first level of categorization, while downstream collectors create additional levels. This pattern is powerful for building multi-dimensional aggregations and summary reports from flat data.

```java
record Sale(String region, String prod, double amount) {}

List<Sale> sales = List.of(
  new Sale("North", "Widget", 100.0),
  new Sale("North", "Widget", 150.0),
  new Sale("North", "Gadget", 200.0),
  new Sale("South", "Widget", 120.0),
  new Sale("South", "Gadget", 180.0)
);

// Group by region, then product, then sum amounts
Map<String, Map<String, Double>> result = sales.stream()
    .collect(Collectors.groupingBy(
      Sale::region,
      Collectors.groupingBy(
        Sale::prod,
        Collectors.summingDouble(Sale::amount)
      )
    ));
// {North={Widget=250.0, Gadget=200.0}, South={...}}
```

Streaming Algorithms

Memory-Efficient File Processing

```java
public static long countLinesMatching(String path, String pat)
    throws IOException {
  try (var lines = Files.lines(Paths.get(path))) {
    return lines
        .filter(line -> line.contains(pat))
        .count();
  }
}

// Word frequency from large file
public static Map<String, Long> wordFrequency(String path)
    throws IOException {
  try (var lines = Files.lines(Paths.get(path))) {
    return lines
        .filter(line -> !line.isEmpty())
        .flatMap(line -> Arrays.stream(line.split("\\s+")))
        .map(String::toLowerCase)
        .collect(Collectors.groupingBy(
          w -> w,
          Collectors.counting()
        ));
  }
}
```

Top-K Selection with Min-Heap

This algorithm achieves O(N log K) complexity instead of O(N log N) for full sorting. It maintains a min-heap of size K, only keeping the largest elements seen. Each new element potentially replaces the heap minimum, making it efficient for large datasets.

```java
public static <T extends Comparable<T>> List<T> topK(
    Stream<T> stream, int k) {
  PriorityQueue<T> minHeap = new PriorityQueue<>(k);

  stream.forEach(item -> {
    if (minHeap.size() < k) {
      minHeap.offer(item);
    } else if (item.compareTo(minHeap.peek()) > 0) {
      minHeap.poll();
      minHeap.offer(item);
    }
  });

  return new ArrayList<>(minHeap);
}

var nums = Stream.of(5, 2, 9, 1, 7, 6, 3, 8, 4);
List<Integer> top3 = topK(nums, 3);
Collections.sort(top3, Collections.reverseOrder());
// [9, 8, 7]
```

Lazy Deduplication

```java
public static <T> Stream<T> deduplicate(Stream<T> stream) {
  Set<T> seen = new HashSet<>();
  return stream.filter(seen::add);
}

var words = Stream.of("apple", "banana", "apple", "cherry");
var unique = deduplicate(words).toList();  // [apple, banana..]
```

Merging Pre-Sorted Streams

```
public static <T extends Comparable<T>> Stream<T> mergeSorted(
    Stream<T> streamA, Stream<T> streamB) {

  Iterator<T> a = streamA.iterator();
  Iterator<T> b = streamB.iterator();

  return Stream.generate(new Supplier<T>() {
    T nextA = a.hasNext() ? a.next() : null;
    T nextB = b.hasNext() ? b.next() : null;

    @Override
    public T get() {
      if (nextA == null && nextB == null) {
        return null;
      } else if (nextA == null) {
        T result = nextB;
        nextB = b.hasNext() ? b.next() : null;
        return result;
      } else if (nextB == null) {
        T result = nextA;
        nextA = a.hasNext() ? a.next() : null;
        return result;
      } else if (nextA.compareTo(nextB) <= 0) {
        T result = nextA;
        nextA = a.hasNext() ? a.next() : null;
        return result;
      } else {
        T result = nextB;
        nextB = b.hasNext() ? b.next() : null;
        return result;
      }
    }
  }).takeWhile(Objects::nonNull);
}

Stream<Integer> s1 = Stream.of(1, 3, 5, 7);
Stream<Integer> s2 = Stream.of(2, 4, 6, 8, 9);
var merged = mergeSorted(s1, s2).toList();  // [1,2,3,4,5...]
```

Streaming CSV to Domain Objects

```java
record Person(String name, int age, String city) {}

public static Stream<Person> parseCsv(String path)
    throws IOException {
  return Files.lines(Paths.get(path))
      .skip(1)  // Skip header
      .map(line -> {
        String[] parts = line.split(",");
        return new Person(
          parts[0].trim(),
          Integer.parseInt(parts[1].trim()),
          parts[2].trim()
        );
      });
}
```

Batch Processing

```
public static <T> void processInBatches(
    Stream<T> stream,
    int batchSize,
    Consumer<List<T>> processBatch) {

  List<T> batch = new ArrayList<>(batchSize);

  stream.forEach(item -> {
    batch.add(item);
    if (batch.size() == batchSize) {
      processBatch.accept(new ArrayList<>(batch));
      batch.clear();
    }
  });

  // Process remaining items
  if (!batch.isEmpty()) {
    processBatch.accept(batch);
  }
}

Stream<Integer> numbers = IntStream.range(0, 25).boxed();
processInBatches(numbers, 10, batch -> {
  System.out.println("Processing batch: " + batch);
});
```

Parallel Streams

Parallel streams automatically divide work among multiple threads using the common ForkJoinPool. They split the data source, process segments concurrently, and combine results. This can dramatically improve performance for CPU-intensive operations on large datasets when operations are independent and stateless.

```java
var numbers = IntStream.range(0, 1_000_000).boxed().toList();

// Sequential: uses single thread
long seqCount = numbers.stream()
  .filter(n -> isPrime(n))
  .count();

// Parallel: uses ForkJoinPool with multiple threads
long parCount = numbers.parallelStream()
  .filter(n -> isPrime(n))
  .count();

// Convert sequential to parallel
long alsoPar = numbers.stream()
  .parallel()
  .filter(n -> isPrime(n))
  .count();

// Convert parallel back to sequential
long backToSeq = numbers.parallelStream()
  .sequential()
  .filter(n -> isPrime(n))
  .count();
```

When to use parallel streams:

- Large datasets (typically 10,000+ elements)
- CPU-intensive operations per element
- Independent operations (no shared mutable state)
- Operations where parallelization overhead < performance gain

When NOT to use parallel streams:

- Small datasets (overhead exceeds benefit)
- I/O-bound operations (use async/reactive instead)
- Operations with side effects or shared mutable state
- When order matters and you need `forEachOrdered`

Avoiding Common Mistakes

```
// BAD: Shared mutable state in parallel stream
List<String> results = new ArrayList<>();  // Not thread-safe!
names.parallelStream()
  .forEach(results::add);  // Race condition!

// GOOD: Use collect instead
List<String> safeResults = names.parallelStream()
  .collect(Collectors.toList());  // Thread-safe

// BAD: I/O in parallel stream
files.parallelStream()
  .forEach(file -> {
    try {
      Files.delete(file);  // I/O doesn't benefit from parallel
    } catch (IOException e) { }
  });

// GOOD: CPU-intensive computation
List<Integer> primes = IntStream.range(2, 100_000)
  .parallel()
  .filter(this::isPrime)
  .boxed()
  .toList();
```

Custom ForkJoinPool

Control parallelism with a custom ForkJoinPool when you need to limit or expand the default thread count. This is useful for isolating parallel stream work from other tasks, preventing resource contention, or tuning thread count based on the nature of your workload.

```java
public static <T, R> List<R> parallelWithCustomPool(
    List<T> data,
    Function<T, R> mapper,
    int parallelism) throws Exception {

  ForkJoinPool customPool = new ForkJoinPool(parallelism);
  try {
    return customPool.submit(() ->
      data.parallelStream()
        .map(mapper)
        .toList()
    ).get();
  } finally {
    customPool.shutdown();
  }
}

var nums = IntStream.range(0, 100).boxed().toList();
var result = parallelWithCustomPool(nums, x -> x * x, 4);
```

Stream Best Practices

1. Prefer method references when possible

```java
// Less readable
names.stream().map(s -> s.toUpperCase()).toList();

// More readable
names.stream().map(String::toUpperCase).toList();
```

2. Keep pipelines short and readable

```
// Better: proper formatting
List<String> result = orders.stream()
  .filter(o -> o.status() == DELIVERED)
  .filter(o -> o.amount().compareTo(HUNDRED) > 0)
  .map(Order::customerId)
  .distinct()
  .sorted()
  .limit(10)
  .toList();

// Or extract predicates
Predicate<Order> isDelivered = o -> o.status() == DELIVERED;
Predicate<Order> isHighValue =
    o -> o.amount().compareTo(HUNDRED) > 0;

List<String> result = orders.stream()
  .filter(isDelivered.and(isHighValue))
  .map(Order::customerId)
  .distinct()
  .sorted()
  .limit(10)
  .toList();
```

3. Be aware of stream consumption

```
Stream<String> stream = names.stream();
stream.forEach(System.out::println);  // Consumes stream
// stream.count();  // IllegalStateException!

// Create new stream for each operation
names.stream().forEach(System.out::println);
long count = names.stream().count();
```

4. Use primitive streams for better performance

```
// Boxing overhead
int sum = numbers.stream()
  .map(n -> n * 2)
  .reduce(0, Integer::sum);

// No boxing
int sum = numbers.stream()
  .mapToInt(n -> n * 2)
  .sum();
```

5. Consider laziness

```
// Lazy: filter and limit work together efficiently
List<String> first3Long = names.stream()
  .filter(s -> s.length() > 5)  // Lazy
  .limit(3)                      // Lazy
  .toList();                     // Terminal: executes pipeline
```

Common Patterns and Idioms

Finding unique elements

```
List<String> unique = list.stream().distinct().toList();
```

Flattening nested collections

```
var nested = List.of(List.of("a", "b"), List.of("c", "d"));
var flattened = nested.stream().flatMap(List::stream).toList();
```

Counting occurrences

```
var wordCounts = words.stream()
  .collect(Collectors.groupingBy(
    Function.identity(), Collectors.counting()));
```

Finding min/max with custom comparator

```
Optional<Product> cheapest = products.stream()
  .min(Comparator.comparing(Product::price));

Optional<Product> mostExpensive = products.stream()
  .max(Comparator.comparing(Product::price));
```

Performance: Streams vs For-Loops

When Streams Are Faster

1. Parallel CPU-intensive operations

```
// Sequential for-loop: Single-threaded
List<Integer> primesLoop = new ArrayList<>();
for (Integer n : numbers) {
  if (isPrime(n)) primesLoop.add(n);
}
// Result: ~2000ms (single core)

// Parallel stream: Multi-threaded
List<Integer> primesStream = numbers.parallelStream()
  .filter(this::isPrime)
  .toList();
// Result: ~500ms (4 cores) - 4x faster!
```

2. Short-circuiting with lazy evaluation

```
// Stream stops as soon as 100 matches found
List<String> resultStream = names.stream()
  .filter(name -> name.length() > 10)
  .filter(name -> name.startsWith("A"))
  .map(String::toUpperCase)
  .limit(100)
  .toList();
```

3. Primitive streams avoid boxing

```
// Pure primitive operations - fastest
int sum = IntStream.range(0, 1_000_000).sum();  // ~2ms
```

When For-Loops Are Faster

1. Small collections

```java
// For small collections (<100), loop has less overhead
List<String> resultLoop = new ArrayList<>();
for (String name : smallList) {
  resultLoop.add(name.toUpperCase());
}
```

2. Complex state and index access

```java
// Direct index access is more natural in loops
for (int i = 0; i < list.size(); i++) {
  String current = list.get(i);
  String previous = i > 0 ? list.get(i - 1) : null;
  String next = i < list.size() - 1 ? list.get(i + 1) : null;
  if (needsProcessing(current, previous, next)) {
    process(current);
  }
}
```

3. Multiple passes over same data

```java
// Single pass, multiple operations
long sum = 0, count = 0, max = Long.MIN_VALUE;
for (long value : values) {
  sum += value;
  count++;
  if (value > max) max = value;
}
```

Performance Guidelines

Size	Operation Type	Recommendation
Small (<100)	Simple (map, filter)	For-loop (simpler)
Small (<100)	Complex pipeline	Stream (readability)
Medium (100-10K)	Simple	Either (similar)
Medium (100-10K)	CPU-intensive	Parallel stream

Size	Operation Type	Recommendation
Large (>10K)	Any	Sequential or parallel
Large (>10K)	CPU-intensive	Parallel stream
Any	I/O-bound	Sequential (never parallel)

Optimization Tips

1. Use primitive streams

```
int sum = list.stream().mapToInt(Person::getAge).sum();
```

2. Use primitive comparators

```
list.stream()
  .sorted(Comparator.comparingInt(Person::getAge)).toList();
```

3. Filter before expensive operations

```
list.stream()
  .filter(s -> cheapCheck(s))      // Quick filter first
  .map(expensiveTransformation)    // Only transforms filtered
  .toList();
```

4. Avoid intermediate collections

```
// Inefficient: Creates intermediate list
List<String> temp = list.stream().filter(p1).toList();
List<String> result = temp.stream().filter(p2).toList();

// Efficient: Single pipeline
var result = list.stream().filter(p1).filter(p2).toList();
```

Bottom Line:

- For most code: Use streams for clarity unless loops are obviously better
- For hot paths: Profile first, then optimize
- For production: Balance readability with measured performance needs

Design Patterns - Creational

4.1 Singleton Pattern - Beyond the Basics

4.1.1 The Double-Checked Locking Problem

Broken Double-Checked Locking

Most developers have seen this pattern, but the JVM's memory model permits instruction reordering. Without `volatile`, a thread can see a non-null reference to an object whose constructor hasn't finished, resulting in corrupt state.

```java
public class Singleton {
  private static Singleton instance;

  public static Singleton getInstance() {
    if (instance == null) {  // Check 1
      synchronized (Singleton.class) {
        if (instance == null) {  // Check 2
          instance = new Singleton();
        }
      }
    }
    return instance;
  }
}
```

Why this is broken (pre-Java 5):

The problem lies in the JVM's memory model and instruction reordering. The line `instance = new Singleton()` is actually three operations at the bytecode level:

1. Allocate memory for `Singleton`

2. Initialize the `Singleton` object (call constructor)
3. Assign the memory address to `instance`

The JVM can reorder steps 2 and 3:

```
1. Allocate memory
3. Assign address to instance (instance is now non-null!)
2. Initialize object (constructor runs)
```

Thread A might set `instance` to a non-null value before the constructor completes. Thread B then sees `instance != null` (Check 1 passes), skips synchronization, and gets a partially constructed object.

The fix (Java 5+):

Volatile Double-Checked Locking

The `volatile` keyword establishes a happens-before relationship between writes and subsequent reads, preventing dangerous reordering. When a thread writes to a volatile variable, all prior writes become visible to any thread that later reads it.

```java
public class Singleton {
  private static volatile Singleton instance;  // volatile critical

  public static Singleton getInstance() {
    if (instance == null) {
      synchronized (Singleton.class) {
        if (instance == null) {
          instance = new Singleton();
        }
      }
    }
    return instance;
  }
}
```

4.1.2 Enum Singleton - The Elegant Solution

Enums provide the most robust singleton implementation because the JVM guarantees exactly one instance exists, handling thread safety, serialization, and reflection automatically. The enum constant initializes when the class loads, with the JVM's initialization lock ensuring thread safety.

```java
public enum Singleton {
  INSTANCE;

  private final Connection connection;

  Singleton() {
    this.connection = createConnection();
  }

  public void doSomething() {
    // business logic
  }
}

// Usage
Singleton.INSTANCE.doSomething();
```

The Compiler-Generated Enum Class

When you declare an enum, the compiler transforms it into a final class extending `java.lang.Enum` with a static initializer block. This block runs exactly once during class loading, protected by the JVM's initialization lock, making enums inherently thread-safe.

```
// What the compiler actually creates
public final class Singleton extends Enum<Singleton> {
  public static final Singleton INSTANCE;

  static {
    INSTANCE = new Singleton("INSTANCE", 0);
  }

  private Singleton(String name, int ordinal) {
    super(name, ordinal);
  }
}
```

The `static` block runs exactly once when the class is loaded, providing thread-safe lazy initialization.

When NOT to use enum singletons:

- When you need lazy initialization (enum is eagerly initialized)
- When you need to extend a class (enums can't extend other classes)
- When you need to mock in tests (enums are final)

4.1.3 Initialization-on-Demand Holder (Bill Pugh Singleton)

This pattern exploits the JVM's class loading guarantees to achieve lazy initialization without synchronization. The inner `Holder` class isn't loaded until `getInstance()` is first called, at which point the JVM initializes the singleton thread-safely.

```
public class Singleton {
  private Singleton() {}

  private static class Holder {
    private static final Singleton INSTANCE = new Singleton();
  }

  public static Singleton getInstance() {
    return Holder.INSTANCE;
  }
}
```

4.1.4 Dependency Injection - The Better Alternative

Spring's dependency injection container manages singleton scope by default, eliminating manual singleton implementations. The container creates one instance per bean definition and injects it wherever needed, handling thread safety and lifecycle while improving testability.

```java
@Configuration
public class AppConfig {
  @Bean
  @Scope("singleton")  // default, can be omitted
  public DatabaseService databaseService() {
    return new DatabaseService();
  }
}
```

Spring Singleton Registry Internals

Spring's `DefaultSingletonBeanRegistry` maintains a `ConcurrentHashMap` of singleton instances for bean lifecycle management. The registry uses double-checked locking internally to ensure thread-safe lazy instantiation while minimizing synchronization overhead during bean retrieval.

```java
// Simplified Spring internals
public class DefaultSingletonBeanRegistry {
  private final Map<String,Object> m = new ConcurrentHashMap<>();

  public Object getSingleton(String beanName) {
    Object obj = this.m.get(beanName);
    if (obj == null) {
      synchronized (this.m) {
        obj = this.m.get(beanName);
        if (obj == null) {
          obj = createBean(beanName);
          this.m.put(beanName, obj);
        }
      }
    }
    return obj;
  }
}
```

Advantages over traditional singletons:

- Testability (can inject mocks)
- No global state
- Lifecycle management
- Proxy creation for AOP

When to still use traditional singletons:

- Library code without DI framework
- Performance-critical paths (avoiding DI lookup overhead)
- Simple utility classes

4.2 Factory Pattern - Lambda and Method Reference Era

4.2.1 Traditional Factory Method

The traditional switch-based factory is easy to understand and works well for stable, small sets of types. However, every new payment processor requires modifying the factory class, violating the open-closed principle and creating coupling issues.

```
public interface PaymentProcessor {
  void process(Payment payment);
}

public class PaymentProcessorFactory {
  public static PaymentProcessor create(PaymentType type) {
    return switch (type) {
      case CREDIT_CARD -> new CreditCardProcessor();
      case PAYPAL -> new PayPalProcessor();
      case CRYPTO -> new CryptoProcessor();
    };
  }
}
```

This works, but it's rigid and requires modifying the factory for new types.

4.2.2 Modern Factory with Supplier

Capturing constructor references as `Supplier` instances transforms a rigid switch statement into a flexible, data-driven registry. New implementations can be registered at runtime without modifying factory code, enabling plugin architectures and configuration-driven assembly.

```
public class PaymentProcessorFactory {
  static Map<PaymentType,Supplier<PaymentProcessor>> F = Map.of(
    PaymentType.CREDIT_CARD, CreditCardProcessor::new,
    PaymentType.PAYPAL, PayPalProcessor::new,
    PaymentType.CRYPTO, CryptoProcessor::new
  );

  public static PaymentProcessor create(PaymentType type) {
    Supplier<PaymentProcessor> factory = F.get(type);
    if (factory == null) {
      throw new IllegalArgumentException("Unknown type: " + type);
    }
    return factory.get();
  }

  // Allow registration of new factories at runtime
  static void register(PaymentType t, Supplier<PaymentProcessor> f){
    if (F instanceof HashMap) {  // if mutable
      F.put(t, f);
    }
  }
}
```

Lambda Metafactory Equivalent Class

When you write `CreditCardProcessor::new`, the compiler uses the `invokedynamic` bytecode instruction instead of generating an anonymous class. At runtime, the lambda metafactory creates a lightweight functional interface implementation that delegates to the constructor efficiently.

```
// Lambda metafactory generates something like this
Supplier<PaymentProcessor> s = new Supplier<PaymentProcessor>() {
  @Override
  public PaymentProcessor get() {
    return new CreditCardProcessor();
  }
};
```

But it's more efficient - the JVM uses `invokedynamic` with lambda metafactory:

```
invokedynamic #42, 0  // InvokeDynamic #0:get:
()Ljava/util/function/Supplier;
```

The `invokedynamic` instruction creates a call site that's optimized by the JIT compiler.

4.2.3 Service Loader Pattern (Java SPI)

Java's Service Provider Interface allows implementations to be discovered automatically at runtime based on classpath entries. List implementation class names in `META-INF/services/` files, and `ServiceLoader` finds and instantiates them using reflection.

```
// META-INF/services/com.example.PaymentProcessor
com.example.CreditCardProcessor
com.example.PayPalProcessor

// Usage
var loader = ServiceLoader.load(PaymentProcessor.class);
for (PaymentProcessor processor : loader) {
  if (processor.supports(paymentType)) {
    return processor;
  }
}
```

How ServiceLoader works internally:

1. Reads `META-INF/services/` files from classpath

2. Uses reflection to instantiate classes: `Class.forName(className).getDeclaredConstructor().newInstance()`
3. Caches instances in a `LinkedHashMap`
4. Lazy loading - classes are instantiated only when iterated

Performance consideration:

- Reflection overhead on first load
- Not suitable for hot paths
- Used heavily by JDBC drivers, logging frameworks

4.2.4 Spring's FactoryBean

When a bean requires complex initialization logic—conditional configuration, external resource setup, or multi-step construction—a `FactoryBean` encapsulates that complexity. Spring calls `getObject()` to obtain the actual bean while the factory handles configuration.

```
public class PoolFactoryBean implements FactoryBean<DataSource> {
  private String url;
  private int maxPoolSize;

  @Override
  public DataSource getObject() throws Exception {
    HikariConfig config = new HikariConfig();
    config.setJdbcUrl(url);
    config.setMaximumPoolSize(maxPoolSize);
    return new HikariDataSource(config);
  }

  @Override
  public Class<?> getObjectType() {
    return DataSource.class;
  }

  @Override
  public boolean isSingleton() {
    return true;
  }
}
```

Spring's internal handling:

When you inject a `FactoryBean`, Spring:

1. Detects the `FactoryBean` interface
2. Calls `getObject()` to get the actual bean
3. Caches the result if `isSingleton()` returns `true`
4. The bean name `&connectionPoolFactoryBean` gives you the factory itself

Accessing Spring FactoryBeans

Spring distinguishes between the factory and its product through a naming convention: prefix the bean name with `&` to get the factory, omit it to get the product. This allows inspecting factory configuration or accessing factory-specific methods.

```
// Get the factory
var fct = context.getBean("&poolFactoryBean", FactoryBean.class);

// Get the product
var ds = context.getBean("poolFactoryBean", DataSource.class);
```

4.3 Builder Pattern - Records, Lombok, and Immutability

4.3.1 Classic Builder (Effective Java Style)

The Effective Java style builder provides a fluent API for constructing complex immutable objects with many parameters. The builder accumulates configuration through method chaining, then `build()` constructs the validated, immutable instance.

```java
public class HttpRequest {
  private final String url;
  private final String method;
  private final Map<String, String> headers;
  private final String body;
  private final Duration timeout;

  private HttpRequest(Builder builder) {
    this.url = requireNonNull(builder.url, "url");
    this.method = builder.method;
    this.headers = Map.copyOf(builder.headers);  // defensive copy
    this.body = builder.body;
    this.timeout = builder.timeout;
  }

  public static class Builder {
    private String url;
    private String method = "GET";
    private Map<String, String> headers = new HashMap<>();
    private String body;
    private Duration timeout = Duration.ofSeconds(30);

    public Builder url(String url) {
      this.url = url;
      return this;
    }

    public Builder method(String method) {
      this.method = method;
      return this;
    }

    public Builder header(String key, String value) {
      this.headers.put(key, value);
      return this;
    }

    public Builder body(String body) {
      this.body = body;
      return this;
    }

    public Builder timeout(Duration timeout) {
      this.timeout = timeout;
      return this;
    }
    public HttpRequest build() {
      return new HttpRequest(this);
    }
  }

  public static Builder builder() {
    return new Builder();
  }
}

// Usage
HttpRequest request = HttpRequest.builder()
  .url("https://api.example.com")
  .method("POST")
  .header("Content-Type", "application/json")
  .body("{\"key\":\"value\"}")
  .timeout(Duration.ofSeconds(60))
  .build();
```

Why this pattern:

- Immutability (all fields `final`)
- Readable API
- Validation in constructor
- Default values in builder
- Defensive copying prevents external modification

4.3.2 Lombok @Builder

Lombok's `@Builder` annotation eliminates builder boilerplate while generating a fully-featured fluent API. The `@Singular` annotation adds special handling for collections, allowing elements to be added one at a time. Combined with `@Value` , this delivers the builder pattern concisely.

```java
@Builder
@Value  // immutable
public class HttpRequest {
  @NonNull String url;
  @Builder.Default String method = "GET";
  @Singular Map<String, String> headers;
  String body;
  @Builder.Default Duration timeout = Duration.ofSeconds(30);
}

// Usage - identical to manual builder
HttpRequest request = HttpRequest.builder()
  .url("https://api.example.com")
  .method("POST")
  .header("Content-Type", "application/json")  // @Singular
  .body("{\"key\":\"value\"}")
  .build();
```

Delomboked Lombok Builder Output

Running `delombok` transforms Lombok annotations into the Java code they generate, revealing what happens at compile time. The generated code shows how `@Singular` creates parallel lists for map keys and values, building an unmodifiable collection.

```java
// Simplified version of what Lombok creates
public class HttpRequest {
  private final String url;
  private final String method;
  private final Map<String, String> headers;
  // ... other fields

  public static class HttpRequestBuilder {
    private String url;
    private String method = "GET";
    private ArrayList<String> headers$key;  // @Singular
    private ArrayList<String> headers$value;

    public HttpRequestBuilder header(String key, String value) {
      if (this.headers$key == null) {
        this.headers$key = new ArrayList<>();
        this.headers$value = new ArrayList<>();
      }
      this.headers$key.add(key);
      this.headers$value.add(value);
      return this;
    }

    public HttpRequest build() {
      Map<String, String> headers;
      if (this.headers$key == null) {
        headers = Collections.emptyMap();
      } else {
        headers = new LinkedHashMap<>();
        for (int i = 0; i < this.headers$key.size(); i++) {
          headers.put(headers$key.get(i), headers$value.get(i));
        }
        headers = Collections.unmodifiableMap(headers);
      }
      return new HttpRequest(url, method, headers, body, timeout);
    }
  }
}
```

Trade-offs:

- ✅ Less boilerplate
- ✅ @Singular for collections
- ✅ @Builder.Default for default values
- ❌ Compile-time dependency on Lombok

- ❌ IDE support required
- ❌ Delombok needed for some tools
- ❌ Less control over builder logic

4.3.3 Records as Builders (Java 14+)

Records give you immutability and auto-generated `equals`, `hashCode`, and `toString`, but lack a built-in builder. Adding a nested builder class provides the fluent construction API, while `withX` methods enable easy copying with modifications.

```java
public record HttpRequest(
  String url,
  String method,
  Map<String, String> headers,
  String body,
  Duration timeout
) {
  // Canonical constructor with validation
  public HttpRequest {
    requireNonNull(url, "url cannot be null");
    method = method == null ? "GET" : method;
    headers = headers == null ? Map.of() : Map.copyOf(headers);
    timeout = timeout == null ? Duration.ofSeconds(30) : timeout;
  }

  // Builder
  public static Builder builder() {
    return new Builder();
  }

  public static class Builder {
    private String url;
    private String method;
    private Map<String, String> headers = new HashMap<>();
    private String body;
    private Duration timeout;

    public Builder url(String url) {
      this.url = url;
      return this;
    }

    public Builder method(String method) {
      this.method = method;
      return this;
    }

    public Builder header(String key, String value) {
      this.headers.put(key, value);
      return this;
    }

    public Builder body(String body) {
      this.body = body;
      return this;
    }
```

```
      public Builder timeout(Duration timeout) {
        this.timeout = timeout;
        return this;
      }

      public HttpRequest build() {
        return new HttpRequest(url, method, headers, body, timeout);
      }
    }

    // Convenient "with" methods for immutable updates
    public HttpRequest withUrl(String url) {
      return new HttpRequest(url, method, headers, body, timeout);
    }

    public HttpRequest withTimeout(Duration timeout) {
      return new HttpRequest(url, method, headers, body, timeout);
    }
  }
```

Record Combined with Lombok Builder

Lombok and records work together beautifully—Lombok generates the builder while the record provides immutability and structural methods. The compact constructor handles validation and default values, running automatically when the builder calls the canonical constructor.

```
@Builder
public record HttpRequest(
  @NonNull String url,
  String method,
  @Singular Map<String, String> headers,
  String body,
  Duration timeout
) {
  // Compact constructor for validation
  public HttpRequest {
    if (method == null) method = "GET";
    if (headers == null) headers = Map.of();
    if (timeout == null) timeout = Duration.ofSeconds(30);
  }
}
```

4.3.4 Staged Builder (Type-Safe Builder)

Staged builders use the type system to enforce a specific construction sequence at compile time—you cannot call methods out of order because return types guide you through the stages. Each stage interface exposes only methods valid at that point.

```java
public class DatabaseConnection {
  private final String host;
  private final int port;
  private final String database;
  private final String username;
  private final String password;

  private DatabaseConnection(Builder builder) {
    this.host = builder.host;
    this.port = builder.port;
    this.database = builder.database;
    this.username = builder.username;
    this.password = builder.password;
  }

  // Staged builder interfaces
  public interface HostStage {
    PortStage host(String host);
  }

  public interface PortStage {
    DatabaseStage port(int port);
  }

  public interface DatabaseStage {
    CredentialsStage database(String database);
  }

  public interface CredentialsStage {
    BuildStage credentials(String username, String password);
  }

  public interface BuildStage {
    BuildStage sslEnabled(boolean enabled);
    DatabaseConnection build();
  }

  public static class Builder implements HostStage, PortStage,
      DatabaseStage, CredentialsStage, BuildStage {
    private String host;
    private int port;
    private String database;
    private String username;
    private String password;
    private boolean sslEnabled = true;

    @Override
    public PortStage host(String host) {
      this.host = host;
      return this;
    }

    @Override
    public DatabaseStage port(int port) {
      this.port = port;
      return this;
    }

    @Override
    public CredentialsStage database(String database) {
      this.database = database;
      return this;
    }

    @Override
    public BuildStage credentials(String user, String pass) {
      this.username = user;
      this.password = pass;
      return this;
    }

    @Override
    public BuildStage sslEnabled(boolean enabled) {
      this.sslEnabled = enabled;
      return this;
    }

    @Override
    public DatabaseConnection build() {
      return new DatabaseConnection(this);
    }
  }

  public static HostStage builder() {
    return new Builder();
  }
}
```

Staged Builder Usage

At the call site, the staged builder reads like any fluent API, but the compiler verifies that you've called each required method in sequence. Skipping a step or calling methods out of order produces a compile error rather than runtime exception.

```
// Usage - compiler enforces order
DatabaseConnection conn = DatabaseConnection.builder()
    .host("localhost")            // Must be first
    .port(5432)                   // Must be second
    .database("mydb")             // Must be third
    .credentials("user", "pass") // Must be fourth
    .sslEnabled(true)             // Optional
    .build();

// This won't compile:
// DatabaseConnection.builder().port(5432) // Error: no host()
```

Use cases:

- Complex objects with required initialization order
- APIs where mistakes are costly (database, cloud resources)
- Public libraries where compile-time safety matters

Trade-off: More code, but impossible to create invalid objects.

4.4 Prototype Pattern - Cloning Done Right

4.4.1 The Problem with `Object.clone()`

The default `clone()` performs a shallow copy—primitive fields are duplicated, but object references are simply copied, not the objects themselves. Both original and clone share mutable nested objects, so modifications through one appear in the other.

```java
public class Document implements Cloneable {
  private String title;
  private List<String> tags;
  private Metadata metadata;

  @Override
  protected Object clone() throws CloneNotSupportedException {
    return super.clone();  // Shallow copy!
  }
}
```

Problem: Shallow copy means `tags` and `metadata` references are shared:

Shared Reference Demonstration

This code demonstrates shallow cloning's danger—adding a tag to what appears to be an independent clone actually modifies the original document's list. The symptom might not appear until much later in execution, making debugging difficult.

```java
var doc1 = new Document("Orig", new ArrayList<>(), new Metadata());
var doc2 = (Document) doc1.clone();

doc2.getTags().add("new-tag");
// doc1 also has "new-tag" - they share the same List!
```

4.4.2 Proper Deep Clone

A proper deep clone must recursively duplicate every mutable object in the graph, not just the top-level object. Collections need to be copied into new instances, and nested objects should have their `clone()` method called recursively.

```java
public class Document implements Cloneable {
  private String title;
  private List<String> tags;
  private Metadata metadata;

  @Override
  protected Document clone() {  // Covariant return type (Java 5+)
    try {
      Document cloned = (Document) super.clone();
      cloned.tags = new ArrayList<>(this.tags);  // Deep copy
      cloned.metadata = this.metadata.clone();   // Recursive clone
      return cloned;
    } catch (CloneNotSupportedException e) {
      throw new AssertionError("Cannot happen - Cloneable");
    }
  }
}
```

Issues with clone():

1. Checked exception `CloneNotSupportedException` is annoying
2. Requires `Cloneable` marker interface (design smell)
3. Doesn't call constructor (bypasses validation)
4. Hard to get right with inheritance

4.4.3 Copy Constructor (Recommended)

A copy constructor takes an existing instance and creates a new object with the same values, using normal constructor machinery. Unlike `clone()`, the copy constructor runs all validation logic and works naturally with `final` fields.

```java
public class Document {
  private final String title;
  private final List<String> tags;
  private final Metadata metadata;

  public Document(String title, List<String> tags, Metadata meta) {
    this.title = requireNonNull(title);
    this.tags = new ArrayList<>(tags);  // Defensive copy
    this.metadata = meta;
  }

  // Copy constructor
  public Document(Document other) {
    this(other.title, other.tags, new Metadata(other.metadata));
  }
}

// Usage
var original = new Document("Title", List.of("tag1"), metadata);
var copy = new Document(original);
```

Advantages:

- No exceptions
- No marker interfaces
- Calls constructor (validation runs)
- Clear intent
- Works with `final` fields

4.4.4 Static Factory for Copying

A static factory method like `copyOf` clearly communicates intent while encapsulating copy logic in one place. Unlike constructors, factory methods can have descriptive names that make code more readable and give flexibility to change implementations.

```java
public class Document {
  // ... fields and constructor

  public static Document copyOf(Document other) {
    return new Document(other);
  }
}

// Usage
Document copy = Document.copyOf(original);
```

4.4.5 Builder-Based Copying

Lombok's `toBuilder = true` generates a method returning a pre-populated builder from an existing instance, perfect for creating variations of immutable objects. You can override specific fields while keeping the rest unchanged, far cleaner than copy-and-modify patterns.

```java
@Builder(toBuilder = true)  // Lombok generates toBuilder()
@Value
public class Document {
  String title;
  @Singular List<String> tags;
  Metadata metadata;
}

// Usage
Document original = Document.builder()
  .title("Original")
  .tag("tag1")
  .metadata(metadata)
  .build();

// Create copy with modifications
Document modified = original.toBuilder()
  .title("Modified")
  .build();
```

Understanding the Generated toBuilder Method

The `toBuilder()` method Lombok generates creates a new builder populated with all the current object's field values. This eliminates tedious boilerplate, and the returned builder is fully mutable so you can chain modifications before calling `build()`.

```java
public Builder toBuilder() {
  return new Builder()
    .title(this.title)
    .tags(this.tags)
    .metadata(this.metadata);
}
```

4.4.6 Serialization-Based Cloning (Deep Copy Any Object)

Serialization-based cloning exploits Java's serialization to create an independent copy of an arbitrarily complex object graph. The technique serializes to bytes and deserializes back, automatically handling all nested references but requiring `Serializable` implementation.

```java
public class SerializationUtil {
  @SuppressWarnings("unchecked")
  public static <T extends Serializable> T deepCopy(T object) {
    try {
      ByteArrayOutputStream baos = new ByteArrayOutputStream();
      try (ObjectOutputStream oos = new ObjectOutputStream(baos)) {
        oos.writeObject(object);
      }

      var bais = new ByteArrayInputStream(baos.toByteArray());
      try (ObjectInputStream ois = new ObjectInputStream(bais)) {
        return (T) ois.readObject();
      }
    } catch (IOException | ClassNotFoundException e) {
      throw new RuntimeException("Deep copy failed", e);
    }
  }
}
```

Trade-offs:

- ✅ Handles arbitrarily complex object graphs
- ✅ No manual copying logic
- ❌ Slow (serialization overhead)
- ❌ Requires `Serializable`
- ❌ Can break with custom `readObject()` / `writeObject()`

Faster Deep Copying with JSON Serialization

JSON-based cloning offers a faster alternative to Java serialization, leveraging Jackson's optimized engine. The process converts objects to JSON and back, typically faster than native serialization. This works with any POJO without requiring the `Serializable` interface.

```
public class JsonUtil {
  private static final ObjectMapper mapper = new ObjectMapper();

  @SuppressWarnings("unchecked")
  public static <T> T deepCopy(T object) {
    try {
      Class<T> clazz = (Class<T>) object.getClass();
      String json = mapper.writeValueAsString(object);
      return mapper.readValue(json, clazz);
    } catch (JsonProcessingException e) {
      throw new RuntimeException("Deep copy failed", e);
    }
  }
}
```

Faster and more flexible, but requires Jackson.

4.5 Object Pool Pattern - Managing Expensive Resources

4.5.1 Apache Commons Pool 2

Apache Commons Pool 2 provides foundational infrastructure for many pooling implementations. You extend `BasePooledObjectFactory` to define pooled object lifecycle: creation, wrapping for tracking, validation, and destruction. The pool handles threading, eviction, and hand-off logic.

```java
class ObjFactory extends BasePooledObjectFactory<ExpensiveObject> {

  @Override
  public ExpensiveObject create() throws Exception {
    // Expensive creation
    return new ExpensiveObject();
  }

  @Override
  public PooledObject<ExpensiveObject> wrap(ExpensiveObject obj) {
    return new DefaultPooledObject<>(obj);
  }

  @Override
  public void destroyObject(PooledObject<ExpensiveObject> p) {
    p.getObject().close();
  }

  @Override
  public boolean validateObject(PooledObject<ExpensiveObject> p) {
    return p.getObject().isValid();
  }
}

// Configuration
var config = new GenericObjectPoolConfig<ExpensiveObject>();
config.setMaxTotal(10);
config.setMaxIdle(5);
config.setMinIdle(2);
config.setTestOnBorrow(true);
config.setTestWhileIdle(true);

// Create pool
GenericObjectPool<ExpensiveObject> pool =
  new GenericObjectPool<>(new ExpensiveObjectFactory(), config);

// Usage
ExpensiveObject obj = pool.borrowObject();
try {
  obj.doWork();
} finally {
  pool.returnObject(obj);
}
```

Internal mechanisms:

Commons Pool uses:

- **LinkedBlockingDeque** for available objects
- **ConcurrentHashMap** for tracking all objects
- **EvictionTimer** for idle object removal
- **AQS (AbstractQueuedSynchronizer)** for blocking on borrow

Tuning Pool Behavior with Configuration Parameters

Pool behavior is controlled through parameters balancing resource usage against availability. Size parameters (`maxTotal` , `maxIdle` , `minIdle`) control object counts, while timeout parameters prevent indefinite waits. Eviction settings determine how aggressively the pool removes stale objects.

```
maxTotal    // Maximum objects in pool (active + idle)
maxIdle     // Maximum idle objects
minIdle     // Minimum idle objects (pool creates proactively)
maxWaitMillis  // How long to wait before throwing exception

// Eviction
timeBetweenEvictionRunsMillis  // How often to run evictor
minEvictableIdleTimeMillis     // Minimum idle time before eviction
numTestsPerEvictionRun         // Objects to test per eviction run

// Validation
testOnBorrow   // Validate before borrowing
testOnReturn   // Validate before returning
testWhileIdle  // Validate during eviction
```

4.5.2 HikariCP - The Fastest Connection Pool

HikariCP is the de facto standard for JDBC connection pooling, chosen as Spring Boot’s default for exceptional performance. Configuration centers on pool sizing to match database limits, timeouts to prevent hung connections, and validation ensuring borrowed connections are usable.

```java
HikariConfig config = new HikariConfig();
config.setJdbcUrl("jdbc:postgresql://localhost:5432/mydb");
config.setUsername("user");
config.setPassword("password");

// Pool sizing
config.setMaximumPoolSize(10);
config.setMinimumIdle(5);

// Timeouts
config.setConnectionTimeout(30000);   // 30 seconds
config.setIdleTimeout(600000);        // 10 minutes
config.setMaxLifetime(1800000);       // 30 minutes

// Validation
config.setConnectionTestQuery("SELECT 1");
config.setValidationTimeout(5000);

HikariDataSource dataSource = new HikariDataSource(config);

// Usage
try (Connection conn = dataSource.getConnection()) {
  // Use connection
}  // Automatically returned to pool
```

The ConcurrentBag Data Structure

HikariCP's speed comes from its custom `ConcurrentBag` designed for borrow/return patterns. Each thread first checks its local list before the shared list, eliminating contention. Compare-and-set operations on connection state avoid locks, making hand-offs nearly lock-free.

```java
// Simplified version
public class ConcurrentBag<T> {
  private final CopyOnWriteArrayList<T> sharedList;
  private final ThreadLocal<List<T>> threadList;

  public T borrow() {
    // Try thread-local list first (no contention!)
    List<T> list = threadList.get();
    for (T item : list) {
      if (item.compareAndSet(STATE_NOT_IN_USE, STATE_IN_USE)) {
        return item;
      }
    }

    // Fall back to shared list
    for (T item : sharedList) {
      if (item.compareAndSet(STATE_NOT_IN_USE, STATE_IN_USE)) {
        return item;
      }
    }

    return null;  // Pool exhausted
  }
}
```

2. **FastList** - Custom ArrayList without range checking (safe in controlled context)

3. **Bytecode generation** - Uses Javassist to create optimized proxy classes

4. **Minimal synchronization** - Uses CAS operations instead of locks where possible

Connection lifecycle in HikariCP:

```
Create → Borrow → Return → Idle → Evict → Close
            ↑        ↓
            └────────┘
          (reuse loop)
```

Validation strategy:

HikariCP doesn't validate on borrow (too expensive). Instead:

- Validates during idle eviction
- Relies on `maxLifetime` to recycle connections
- Handles exceptions during use and removes bad connections

4.5.3 Thread Pools - ExecutorService

The `Executors` factory provides methods for common thread pool configurations suited to different workloads. Fixed pools maintain constant threads for CPU-bound work; cached pools grow dynamically for I/O-bound loads; `ThreadPoolExecutor` offers full control.

```java
// Fixed thread pool
ExecutorService executor = Executors.newFixedThreadPool(10);

// Cached thread pool (grows unbounded!)
ExecutorService executor = Executors.newCachedThreadPool();

// Custom ThreadPoolExecutor
ThreadPoolExecutor executor = new ThreadPoolExecutor(
  5,                              // corePoolSize
  10,                             // maximumPoolSize
  60L, TimeUnit.SECONDS,          // keepAliveTime
  new LinkedBlockingQueue<>(100),   // workQueue
  new ThreadPoolExecutor.CallerRunsPolicy()  // rejectionHandler
);
```

Understanding ThreadPoolExecutor's Task Dispatch Logic

When you submit a task, `ThreadPoolExecutor` follows a decision tree balancing responsiveness against efficiency. First it creates a core thread; if saturated, it queues; only when the queue is full does it create threads up to maximum.

```
// Simplified logic
public void execute(Runnable command) {
  int c = ctl.get();  // Atomic int encoding state + worker count

  // 1. If fewer than core threads, create new worker
  if (workerCountOf(c) < corePoolSize) {
    if (addWorker(command, true)) {
      return;
    }
  }

  // 2. Try to add to queue
  if (isRunning(c) && workQueue.offer(command)) {
    // Successfully queued
    return;
  }

  // 3. Queue full, try to create up to max pool size
  if (!addWorker(command, false)) {
    // Rejected!
    reject(command);
  }
}
```

Parallel Divide-and-Conquer with ForkJoinPool

`ForkJoinPool` is optimized for recursive divide-and-conquer algorithms where tasks spawn subtasks. The pattern splits work until small enough, then forks subtasks asynchronously and joins results. Work-stealing ensures all processors stay busy.

```
ForkJoinPool pool = new ForkJoinPool(
  Runtime.getRuntime().availableProcessors()
);

RecursiveTask<Integer> task = new RecursiveTask<>() {
  @Override
  protected Integer compute() {
    if (shouldSplit()) {
      RecursiveTask<Integer> left = createSubtask();
      RecursiveTask<Integer> right = createSubtask();

      left.fork();  // Async execute
      int rightResult = right.compute();  // Sync execute
      int leftResult = left.join();  // Wait for result

      return leftResult + rightResult;
    } else {
      return computeDirectly();
    }
  }
};

int result = pool.invoke(task);
```

How Work Stealing Balances Load Across Threads

Each worker maintains a deque of tasks, taking from the head using LIFO for cache locality. When empty, it steals from another thread's tail using FIFO, grabbing older tasks. This asymmetric pattern minimizes contention while ensuring load distribution.

Each worker thread has a deque:

```
Thread 1: [Task A, Task B, Task C] ← Takes from head (LIFO)
Thread 2: [Task D, Task E]         ← Takes from head
Thread 3: []                       ← Steals from tail of others
(FIFO)
```

Why LIFO for own thread? Cache locality - recently created subtasks likely share data. Why FIFO for stealing? Larger tasks are typically enqueued first.

4.5.4 Virtual Thread Pool (Java 21+)

Java 21's virtual threads change thread pooling: since virtual threads are cheap and blocking them unmounts from carrier threads, traditional pooling reasons disappear. Create a virtual thread per task and let the JVM schedule them across carrier threads.

```
// Old way - limit threads
ExecutorService executor = Executors.newFixedThreadPool(100);

// New way - create millions of virtual threads
var executor = Executors.newVirtualThreadPerTaskExecutor();

// Or use directly
Thread.startVirtualThread(() -> {
  // This thread is cheap!
});
```

Why pooling changes:

- Virtual threads are lightweight (few KB vs. MB for platform threads)
- Creating a virtual thread is cheap
- Blocking a virtual thread is cheap (unmounts from carrier)

When to still pool with virtual threads:

- Expensive object creation (DB connections still expensive)
- Resource limiting (connection limits, rate limiting)
- Complex lifecycle management

4.6 Pattern Selection Guide

Pattern	Use When	Avoid When
Singleton	Truly global resource, stateless utility	Need testability, multiple instances

Pattern	Use When	Avoid When
Enum Singleton	Default singleton choice	Need lazy init, inheritance, or mocking
Factory	Object creation complexity, runtime type selection	Simple constructors suffice
Builder	4+ parameters, many optional fields	Simple objects, all fields required
Prototype	Copying complex objects	Simple objects, immutable data
Object Pool	Expensive creation, high reuse rate	Cheap objects, low reuse

---

4.7 Modern Java Patterns Summary

Use:

- ✅ Enum singleton for singletons
- ✅ Dependency injection over manual singletons
- ✅ Lambda-based factories with `Supplier<T>`
- ✅ Lombok `@Builder` for internal DTOs
- ✅ Records + manual builder for public APIs
- ✅ Copy constructors over `clone()`
- ✅ HikariCP for connection pooling
- ✅ `Executors.newVirtualThreadPerTaskExecutor()` for Java 21+

Avoid:

- ❌ Double-checked locking without `volatile`
- ❌ `Object.clone()` without deep copy
- ❌ Custom thread pools (use `Executors` or virtual threads)
- ❌ `Cloneable` interface

Key insight: Modern Java (8+) with lambdas, records, and better concurrency primitives makes many classic patterns simpler and more elegant. Don’t cargo-cult old patterns - adapt them to leverage modern features.