OOPS Cheat Sheet

🚀 Java OOP Mastery Cheat Sheet for Senior Developers

From one senior dev to another - this is what you actually need to know for interviews and real projects

---

📦 1. Core OOP Principles (The Classic 4 + 2)

🔒 Encapsulation

What it really means: Hide your mess, expose only what's necessary. Think of it as your service's public API.

Real-world: Like a payment gateway - you don't care HOW Stripe processes payments, you just call `processPayment()`.

```
@Service
public class PaymentService {
    private final StripeClient stripeClient;  // Hidden implementation
    private final Logger logger;              // Internal details

    public PaymentResult processPayment(PaymentRequest request) {
        // Public API - clean interface
        validateRequest(request);
        return executePayment(request);
    }

    private void validateRequest(PaymentRequest request) {
        // Hidden complexity
    }
}
```

🌳 Inheritance

What it really means: Reuse code, but don't overdo it. Composition > Inheritance in most cases.

Real-world: Base repository for common CRUD operations.

```
// Good use of inheritance
public abstract class BaseEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @CreatedDate
    private LocalDateTime createdAt;

    @LastModifiedDate
    private LocalDateTime updatedAt;
}

@Entity
public class User extends BaseEntity {
    private String email;
    // User-specific fields
}
```

🎭 Abstraction

What it really means: Define the "what", not the "how". Your interface is a contract.

Real-world: Different notification channels, same interface.

```
public interface NotificationService {
    void send(String recipient, String message);
}

@Service
@ConditionalOnProperty(name = "notification.type", havingValue = "emai
l")
public class EmailNotificationService implements NotificationService {
    @Override
    public void send(String recipient, String message) {
        // Email-specific implementation
```

```
    }
}

@Service
@ConditionalOnProperty(name = "notification.type", havingValue = "sms")
public class SmsNotificationService implements NotificationService {
    @Override
    public void send(String recipient, String message) {
        // SMS-specific implementation
    }
}
```

🔄 Polymorphism

What it really means: Same interface, different behaviors. Spring uses this everywhere with dependency injection.

Real-world: Different payment processors, same interface.

```
@RestController
public class PaymentController {
    private final Map<String, PaymentProcessor> processors;

    @Autowired
    public PaymentController(List<PaymentProcessor> processorList) {
        // Spring injects all implementations
        this.processors = processorList.stream()
            .collect(Collectors.toMap(
                p -> p.getType(),
                Function.identity()
            ));
    }

    @PostMapping("/pay")
    public PaymentResult pay(@RequestBody PaymentRequest request) {
        // Runtime polymorphism
        return processors.get(request.getType())
            .process(request);
```

```
    }
}
```

🏗️ Class & Object

What it really means: Class is the blueprint (like your Dockerfile), Object is the running container.

```
// Class = Blueprint
@Component
@Scope("prototype")  // New instance each time
public class ShoppingCart {
    private List<Item> items = new ArrayList<>();

    public void addItem(Item item) {
        items.add(item);
    }
}

// Object = Instance
@Service
public class CheckoutService {
    @Autowired
    private Provider<ShoppingCart> cartProvider;

    public void checkout(Long userId) {
        ShoppingCart cart = cartProvider.get(); // New object
        // Use the cart instance
    }
}
```

📨 Message Passing

What it really means: Objects communicate via method calls. In microservices, it's REST/messaging.

```
// Within JVM
@Service
public class OrderService {
```

```
    @Autowired
    private PaymentService paymentService;

    public void createOrder(Order order) {
        // Message passing via method call
        PaymentResult result = paymentService.processPayment(order.getPa
yment());
    }
}

// Between services
@FeignClient(name = "payment-service")
public interface PaymentClient {
    @PostMapping("/payments")
    PaymentResult processPayment(@RequestBody PaymentRequest reque
st);
}
```

🔧 2. Supporting Java OOP Constructs

`this` keyword

When to use: Disambiguate parameters, method chaining, passing current instance.

```
@Entity
public class User {
    private String email;

    public User(String email) {
        this.email = email;  // Disambiguate
    }

    public User withEmail(String email) {
        this.email = email;
        return this;  // Method chaining
    }
```

```
    public void register() {
        eventPublisher.publish(this);  // Pass current instance
    }
}
```

`super` keyword

When to use: Call parent constructor/methods, especially in inheritance chains.

```
@Service
public class PremiumUserService extends UserService {

    public PremiumUserService(UserRepository repo) {
        super(repo);  // Call parent constructor
    }

    @Override
    public void register(User user) {
        super.register(user);  // Extend parent behavior
        addPremiumFeatures(user);
    }
}
```

`final` keyword

When to use: Immutability, prevent inheritance, constants.

```
public final class SecurityUtils {  // Can't be extended

    private static final String SECRET_KEY = "xyz";  // Constant

    public static String encrypt(final String data) {  // Can't reassign paramet
er
        // Implementation
    }
}

@Service
public class UserService {
```

```
    private final UserRepository repository;  // Must be initialized, can't be ch
anged

    public UserService(UserRepository repository) {
        this.repository = repository;
    }
}
```

`static` keyword

When to use: Utility methods, factory methods, shared state (carefully).

```
@Component
public class ApplicationUtils {
    private static final Logger logger = LoggerFactory.getLogger(Application
Utils.class);

    public static String generateId() {
        return UUID.randomUUID().toString();
    }

    // Factory method
    public static User createUser(String email) {
        return User.builder()
            .email(email)
            .id(generateId())
            .build();
    }
}
```

`transient` keyword

When to use: Skip serialization for sensitive/calculated fields.

```
@Entity
public class User implements Serializable {
    private String username;
    private transient String password;  // Don't serialize
    private transient String sessionToken;  // Calculated field
```

```
    @JsonIgnore  // Modern alternative for JSON
    private String internalNotes;
}
```

`volatile` keyword

When to use: Multi-threaded access to shared variables (rare in Spring apps).

```
@Component
public class FeatureToggle {
    private volatile boolean enabled = false;  // Thread-safe read/write

    public void toggle() {
        enabled = !enabled;  // Visible to all threads immediately
    }
}
```

Access Modifiers

```
public class ServiceLayer {
    public void apiMethod() {}       // Accessible everywhere
    protected void inheritMethod() {} // Same package + subclasses
    void packageMethod() {}          // Same package only (default)
    private void internalMethod() {} // This class only
}
```

Constructor Chaining

```
@Entity
public class Product {
    private String name;
    private BigDecimal price;
    private String category;

    public Product() {
        this("Unknown", BigDecimal.ZERO);  // Chain to another constructor
    }
```

```
    public Product(String name, BigDecimal price) {
        this(name, price, "General");
    }

    public Product(String name, BigDecimal price, String category) {
        this.name = name;
        this.price = price;
        this.category = category;
    }
}
```

Object Lifecycle

```
public class ObjectLifecycle {
    // 1. Class loading (static blocks)
    static {
        System.out.println("Class loaded");
    }

    // 2. Instance creation
    private String field = "initialized";

    // 3. Constructor
    public ObjectLifecycle() {
        System.out.println("Constructor called");
    }

    // 4. Usage
    public void doWork() { }

    // 5. Eligible for GC when no references

    // 6. finalize() - deprecated, don't use
}
```

`==` VS `.equals()` VS `hashCode()`

```
@Entity
public class User {
    private Long id;
    private String email;

    // == compares references
    // .equals() compares values (must override)
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;  // Same reference
        if (!(o instanceof User)) return false;
        User user = (User) o;
        return Objects.equals(email, user.email);  // Value comparison
    }

    // hashCode() must be consistent with equals()
    @Override
    public int hashCode() {
        return Objects.hash(email);
    }
}

// Usage
User u1 = new User("test@example.com");
User u2 = new User("test@example.com");
u1 == u2;       // false (different objects)
u1.equals(u2);   // true (same email)
```

`instanceof` and Pattern Matching (Java 14+)

```
public void processEntity(Object entity) {
    // Old way
    if (entity instanceof User) {
        User user = (User) entity;
        processUser(user);
    }

    // Modern way (Java 14+)
```

```
    if (entity instanceof User user) {
        processUser(user);  // No casting needed
    }

    // Pattern matching in switch (Java 17+)
    switch (entity) {
        case User user -> processUser(user);
        case Product product -> processProduct(product);
        case null -> handleNull();
        default -> handleUnknown(entity);
    }
}
```

🔗 3. OOP Relationships

Association (Uses-A)

What it means: Loose coupling, objects can exist independently.

```
@Entity
public class User {
    @ManyToMany
    private List<Project> projects;  // User uses projects
}

@Entity
public class Project {
    @ManyToMany(mappedBy = "projects")
    private List<User> users;  // Projects used by users
}
```

Aggregation (Has-A)

What it means: Whole-part relationship, but parts can exist independently.

```
@Entity
public class Department {
    @OneToMany
```

```
    private List<Employee> employees;  // Department has employees

    // Employees can exist without department
}
```

Composition (Part-Of)

What it means: Strong relationship, parts can't exist without whole.

```
@Entity
public class Order {
    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    private List<OrderLine> lines;  // Lines are part of order

    // OrderLines deleted when Order is deleted
}
```

Dependency

What it means: One class depends on another to function.

```
@Service
public class EmailService {
    private final JavaMailSender mailSender;  // Depends on mail sender

    public EmailService(JavaMailSender mailSender) {
        this.mailSender = mailSender;
    }
}
```

🎯 4. Design-Oriented OOP Concepts

SOLID Principles

S - Single Responsibility

```
// Bad
public class UserService {
```

```
    public void saveUser(User user) { }
    public void sendEmail(String email) { }  // Different responsibility
    public void generateReport() { }         // Different responsibility
}

// Good
@Service
public class UserService {
    public void saveUser(User user) { }
}

@Service
public class EmailService {
    public void sendEmail(String email) { }
}
```

O - Open/Closed

```
// Open for extension, closed for modification
public interface PaymentProcessor {
    PaymentResult process(PaymentRequest request);
}

// Extend by adding new implementations, not modifying existing
@Component
public class StripeProcessor implements PaymentProcessor { }

@Component
public class PayPalProcessor implements PaymentProcessor { }
```

L - Liskov Substitution

```
// Subclass should be replaceable for base class
public class Rectangle {
    protected int width, height;

    public void setWidth(int width) { this.width = width; }
    public void setHeight(int height) { this.height = height; }
```

```
}

// Bad - Square changes behavior
public class Square extends Rectangle {
    @Override
    public void setWidth(int width) {
        this.width = width;
        this.height = width;  // Breaks LSP
    }
}
```

I - Interface Segregation

```
// Bad - fat interface
public interface Worker {
    void work();
    void eat();
    void sleep();
}

// Good - segregated interfaces
public interface Workable {
    void work();
}

public interface LivingBeing {
    void eat();
    void sleep();
}
```

D - Dependency Inversion

```
// Depend on abstractions, not concretions
@Service
public class NotificationService {
    private final NotificationSender sender;  // Interface, not concrete class

    public NotificationService(NotificationSender sender) {
```

```
        this.sender = sender;
    }
}
```

DRY (Don't Repeat Yourself)

```
// Bad
public BigDecimal calculateTax(BigDecimal amount) {
    return amount.multiply(new BigDecimal("0.18"));
}

public BigDecimal calculateServiceTax(BigDecimal amount) {
    return amount.multiply(new BigDecimal("0.18"));
}

// Good
private static final BigDecimal TAX_RATE = new BigDecimal("0.18");

public BigDecimal calculateTax(BigDecimal amount) {
    return amount.multiply(TAX_RATE);
}
```

KISS (Keep It Simple, Stupid)

```
// Over-engineered
public class UserValidator {
    private final ValidatorFactory factory;
    private final ValidatorChain chain;
    // 10 more dependencies
}

// Simple and clear
public class UserValidator {
    public boolean isValid(User user) {
        return user != null
            && user.getEmail() != null
            && user.getEmail().contains("@");
```

```
    }
}
```

YAGNI (You Aren't Gonna Need It)

```
// Don't build for imaginary future requirements
// Bad
public interface UserService {
    User findById(Long id);
    User findByEmail(String email);
    User findByPhone(String phone);       // Not needed yet
    User findBySocialSecurity(String ssn);  // Not needed yet
    List<User> findByZipCode(String zip);   // Not needed yet
}
```

🎨 5. Abstraction Tools in Java

Abstract Class vs Interface

```
// Abstract class - IS-A relationship, shared implementation
public abstract class BaseRepository<T, ID> {
    @Autowired
    protected EntityManager entityManager;

    public abstract T findById(ID id);

    public void save(T entity) {
        entityManager.persist(entity);  // Shared implementation
    }
}

// Interface - CAN-DO relationship, contract only
public interface Cacheable {
    String getCacheKey();
    default long getCacheTTL() {
        return 3600;  // Default implementation (Java 8+)
```

```
    }
}
```

Marker Interface

```
// No methods, just marks a capability
public interface Auditable { }

@Aspect
@Component
public class AuditAspect {
    @AfterReturning("@target(com.example.Auditable)")
    public void audit(JoinPoint joinPoint) {
        // Audit all classes marked with Auditable
    }
}
```

Functional Interface

```
@FunctionalInterface  // Single abstract method
public interface Processor<T, R> {
    R process(T input);

    default R processWithLogging(T input) {
        log.info("Processing: {}", input);
        return process(input);
    }
}

// Lambda usage
Processor<String, Integer> lengthProcessor = str -> str.length();
```

When to Use What?

Feature	Abstract Class	Interface
Multiple inheritance	❌ Not supported (class inheritance is single)	✅ Supported (can implement multiple interfaces)

Feature	Abstract Class	Interface
Instance variables	✅ Allowed	❌ Not allowed (only `public static final`)
Constructor	✅ Can have constructors	❌ No constructors
Access modifiers	All allowed (`public`, `protected`, etc.)	Mostly `public` (methods are `public` by default)
When to use	Shared code + state	Contract-only; API boundaries

🔮 6. Runtime & Reflection Concepts

Reflection Basics

```
@Component
public class BeanInspector {

    public void inspectBean(Object bean) throws Exception {
        Class<?> clazz = bean.getClass();

        // Get all fields
        Field[] fields = clazz.getDeclaredFields();
        for (Field field : fields) {
            field.setAccessible(true);  // Access private fields
            Object value = field.get(bean);
            System.out.println(field.getName() + " = " + value);
        }

        // Call method dynamically
        Method method = clazz.getMethod("toString");
        String result = (String) method.invoke(bean);
    }
}
```

Dynamic Proxies (How Spring AOP works)

```
public class TransactionProxy implements InvocationHandler {
    private final Object target;
    private final TransactionManager txManager;
```

```
    public Object invoke(Object proxy, Method method, Object[] args) throw
s Throwable {
        Transaction tx = txManager.begin();
        try {
            Object result = method.invoke(target, args);
            tx.commit();
            return result;
        } catch (Exception e) {
            tx.rollback();
            throw e;
        }
    }
}

// Spring does this automatically with @Transactional
```

Custom Annotations

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface RateLimit {
    int value() default 10;
    int duration() default 60;
}

@Aspect
@Component
public class RateLimitAspect {
    @Around("@annotation(rateLimit)")
    public Object enforce(ProceedingJoinPoint pjp, RateLimit rateLimit) thro
ws Throwable {
        String key = generateKey(pjp);
        if (rateLimiter.tryAcquire(key, rateLimit.value(), rateLimit.duration())) {
            return pjp.proceed();
        }
        throw new RateLimitExceededException();
```

```
    }
}
```

ClassLoaders

```
// Understanding classloader hierarchy
public class ClassLoaderDemo {
    public void showClassLoaders() {
        // Bootstrap classloader (null) → Extension → Application → Custom

        ClassLoader appClassLoader = this.getClass().getClassLoader();
        System.out.println("App ClassLoader: " + appClassLoader);

        ClassLoader extClassLoader = appClassLoader.getParent();
        System.out.println("Extension ClassLoader: " + extClassLoader);

        ClassLoader bootstrapClassLoader = extClassLoader.getParent();
        System.out.println("Bootstrap ClassLoader: " + bootstrapClassLoade
r); // null
    }
}
```

🏗️ 7. Design Patterns Using OOP

Factory Pattern

Problem: Object creation logic becomes complex

```
@Component
public class NotificationFactory {
    private final Map<String, NotificationService> services;

    public NotificationService create(String type) {
        return services.get(type);
    }
}
```

Singleton Pattern

Problem: Need exactly one instance

```
// Spring handles this with @Component/@Service (singleton by default)
// Manual implementation (avoid if possible)
public enum DatabaseConnection {
    INSTANCE;

    private final DataSource dataSource;

    DatabaseConnection() {
        this.dataSource = createDataSource();
    }
}
```

Strategy Pattern

Problem: Multiple algorithms for same purpose

```
@Component
public class PricingService {
    private final Map<CustomerType, PricingStrategy> strategies;

    public BigDecimal calculatePrice(Order order, CustomerType type) {
        return strategies.get(type).calculate(order);
    }
}

public interface PricingStrategy {
    BigDecimal calculate(Order order);
}
```

Observer Pattern

Problem: Notify multiple objects of state changes

```
// Spring's ApplicationEventPublisher
@Component
```

```
public class OrderService {
    @Autowired
    private ApplicationEventPublisher publisher;

    public void createOrder(Order order) {
        // Business logic
        publisher.publishEvent(new OrderCreatedEvent(order));
    }
}

@EventListener
public void handleOrderCreated(OrderCreatedEvent event) {
    // React to event
}
```

Decorator Pattern

Problem: Add behavior without modifying original

```
public interface DataSource {
    String read();
}

@Component
@Primary
public class CachedDataSource implements DataSource {
    private final DataSource delegate;
    private final Cache cache;

    public String read() {
        return cache.get("key", () -> delegate.read());
    }
}
```

Builder Pattern

Problem: Complex object construction

```
@Builder  // Lombok
public class User {
    private String email;
    private String name;
    private Set<Role> roles;
}

// Usage
User user = User.builder()
    .email("test@example.com")
    .name("Test User")
    .roles(Set.of(Role.USER))
    .build();
```

💾 8. Memory & Performance in Java OOP

Heap vs Stack

```
public class MemoryExample {
    // Stack: Primitives, references, method calls
    public void method() {
        int x = 10;             // Stack
        String s = "Hello";     // Reference on stack, object on heap
        User user = new User();  // Reference on stack, object on heap
    }  // All stack variables cleared here

    // Heap: Objects, arrays, static variables
    private static final Map<String, User> cache = new HashMap<>();  // He
ap
}
```

Garbage Collection

```
// Object becomes eligible for GC when no references exist
public class GCExample {
    public void demonstrateGC() {
```

```
        User user = new User("temp");  // Strong reference
        WeakReference<User> weak = new WeakReference<>(user);

        user = null;  // Now eligible for GC
        System.gc();  // Suggest GC (don't do in production)

        User retrieved = weak.get();  // Might be null
    }
}

// Help GC with proper resource management
@Service
public class ResourceService {
    public void processLargeFile(String path) {
        try (BufferedReader reader = new BufferedReader(new FileReader(pat
h))) {
            // Process file
        }  // Auto-closed, memory freed
    }
}
```

Escape Analysis

```
// JVM optimization - objects that don't escape can be stack-allocated
public class EscapeAnalysis {
    // Object doesn't escape - can be optimized
    public int calculate() {
        Point p = new Point(10, 20);  // May be stack-allocated
        return p.x + p.y;
    }

    // Object escapes - must be heap-allocated
    private Point savedPoint;
    public void save() {
        savedPoint = new Point(10, 20);  // Escapes method
    }
}
```

Object Pooling

```
// Pool expensive objects
@Configuration
public class DataSourceConfig {
    @Bean
    public DataSource dataSource() {
        HikariConfig config = new HikariConfig();
        config.setMaximumPoolSize(10);  // Pool connections
        config.setMinimumIdle(5);
        config.setConnectionTimeout(30000);
        return new HikariDataSource(config);
    }
}

// Manual pooling for expensive objects
@Component
public class ExpensiveObjectPool {
    private final Queue<ExpensiveObject> pool = new ConcurrentLinkedQue
ue<>();

    public ExpensiveObject borrow() {
        ExpensiveObject obj = pool.poll();
        return obj != null ? obj : new ExpensiveObject();
    }

    public void returnObject(ExpensiveObject obj) {
        obj.reset();
        pool.offer(obj);
    }
}
```

Performance Optimization Tips

```
// 1. Use primitives when possible
// Bad
private Integer count;  // Unnecessary boxing
// Good
```

```
private int count;

// 2. Intern strings for comparison
private static final String STATUS_ACTIVE = "ACTIVE".intern();
if (user.getStatus().intern() == STATUS_ACTIVE) {  // Reference comparison
    // Faster than equals()
}

// 3. Use StringBuilder for concatenation
// Bad
String result = "";
for (String s : list) {
    result += s;  // Creates new String each time
}
// Good
StringBuilder sb = new StringBuilder();
for (String s : list) {
    sb.append(s);
}

// 4. Cache expensive computations
@Component
public class ExpensiveService {
    @Cacheable("expensive-calculations")
    public Result calculate(String input) {
        // Expensive computation
    }
}

// 5. Use lazy initialization for expensive objects
@Component
@Lazy
public class HeavyService {
    // Initialized only when first accessed
}
```

🎯 Interview Pro Tips

Common Gotchas

1. **Private methods can't be overridden** - They're not inherited
2. **Static methods can't be overridden** - They're hidden, not overridden
3. **Constructors aren't inherited** - Each class needs its own
4. **Multiple inheritance** - Java doesn't support it for classes (Diamond problem)
5. **Object class methods** - Know toString(), equals(), hashCode(), clone(), finalize()

Real Project Examples

```
// How we actually use OOP in production

// 1. Service layer abstraction
@Service
@Transactional
public class UserServiceImpl implements UserService {
    // Real implementation with transaction boundaries
}

// 2. Repository pattern
public interface UserRepository extends JpaRepository<User, Long> {
    // Spring Data JPA generates implementation
}

// 3. Event-driven architecture
@Component
public class OrderEventHandler {
    @EventListener
    @Async
    public void handle(OrderCreatedEvent event) {
        // Decoupled event handling
    }
}

// 4. Configuration as code
```

```
@Configuration
@Profile("production")
public class ProductionConfig {
    // Environment-specific beans
}
```

Memory-Efficient Patterns

```
// 1. Flyweight pattern for repeated objects
@Component
public class CurrencyFactory {
    private final Map<String, Currency> cache = new ConcurrentHashMap<
>();

    public Currency getCurrency(String code) {
        return cache.computeIfAbsent(code, Currency::new);
    }
}

// 2. String deduplication
private final Map<String, String> stringPool = new ConcurrentHashMap<>
();
public String intern(String str) {
    return stringPool.putIfAbsent(str, str);
}

// 3. Lazy collections
@Entity
public class Department {
    @OneToMany(fetch = FetchType.LAZY)  // Load only when accessed
    private List<Employee> employees;
}
```

📝 Quick Interview Answers

Q: Difference between abstract class and interface?
A: "In my projects, I use interfaces for contracts (like PaymentProcessor) and

abstract classes when I need shared implementation (like BaseRepository with common CRUD). Since Java 8, interfaces can have default methods, but I still prefer abstract classes when I need state or constructor logic."

Q: How does Spring use OOP?
A: "Spring is built on OOP - dependency injection uses polymorphism, AOP uses dynamic proxies, and the entire bean container relies on reflection. Every @Service, @Repository is leveraging inheritance and polymorphism."

Q: When would you break encapsulation?
A: "Rarely, but frameworks do it - like JPA accessing private fields via reflection, or Spring injecting into private fields. In application code, I'd only consider it for testing with @VisibleForTesting."

Q: Real example of polymorphism in your project?
A: "We have multiple payment gateways - Stripe, PayPal, Razorpay. All implement PaymentGateway interface. The controller doesn't know which implementation it's using - Spring injects based on configuration. Classic polymorphism."

Remember: In interviews, connect OOP concepts to real problems you've solved. Don't just recite definitions - show you understand the WHY behind these principles.

✈️Happy Coding!
