1. Encapsulamento

O encapsulamento protege os dados internos de uma classe, permitindo acesso apenas através de métodos controlados.

Java:

```
public class ContaBancaria {
    // Atributos privados (encapsulados)
    private String numeroConta;
    private double saldo;
    private String titular;

    // Construtor
    public ContaBancaria(String numeroConta, String titular) {
        this.numeroConta = numeroConta;
        this.titular = titular;
        this.saldo = 0.0;
    }

    // Métodos públicos para acesso controlado
    public void depositar(double valor) {
        if (valor > 0) {
            saldo += valor;
            System.out.println("Depósito de R$" + valor + " realizado.");
        }
    }

    public boolean sacar(double valor) {
        if (valor > 0 && valor <= saldo) {
            saldo -= valor;
            System.out.println("Saque de R$" + valor + " realizado.");
            return true;
        }
        System.out.println("Saldo insuficiente.");
        return false;
    }

    // Getter para saldo (apenas leitura)
    public double getSaldo() {
        return saldo;
    }

    // Getter para titular
    public String getTitular() {
        return titular;
    }
```

```
    → Setter para titular (com validação)
    public void setTitular(String titular) {
        if (titular != null && !titular.isEmpty()) {
            this.titular = titular;
        }
    }
}

// Uso
public class Main {
    public static void main(String[] args) {
        ContaBancaria conta = new ContaBancaria("12345", "João");
        conta.depositar(1000);
        conta.sacar(500);
        System.out.println("Saldo: R$" + conta.getSaldo());

        // conta.saldo = 1000000; // ERRO: atributo privado
    }
}
```

2. Herança

Permite que uma classe herde atributos e métodos de outra classe.

```java
// Classe base (pai)
public class Veiculo {
    protected String marca;
    protected String modelo;
    protected int ano;

    public Veiculo(String marca, String modelo, int ano) {
        this.marca = marca;
        this.modelo = modelo;
        this.ano = ano;
    }

    public void acelerar() {
        System.out.println("Veículo acelerando...");
    }

    public void frear() {
        System.out.println("Veículo freando...");
    }
}

// Classe derivada (filha) - Herança
public class Carro extends Veiculo {
```

```
    private int portas;
    private boolean arCondicionado;

    public Carro(String marca, String modelo, int ano, int portas, boolean arCondicionado) {
        super(marca, modelo, ano); // Chama construtor da classe pai
        this.portas = portas;
        this.arCondicionado = arCondicionado;
    }

    // Método específico de Carro
    public void abrirPortaMalas() {
        System.out.println("Porta-malas aberto");
    }

    // Sobrescrita de método (override)
    @Override
    public void acelerar() {
        System.out.println("Carro acelerando rapidamente!");
    }
}

// Outra classe derivada
public class Moto extends Veiculo {
    private int cilindradas;

    public Moto(String marca, String modelo, int ano, int cilindradas) {
        super(marca, modelo, ano);
        this.cilindradas = cilindradas;
    }

    public void empinar() {
        System.out.println("Moto empinando!");
    }
}

// Uso
public class Main {
    public static void main(String[] args) {
        Carro carro = new Carro("Toyota", "Corolla", 2023, 4, true);
        Moto moto = new Moto("Honda", "CB500", 2023, 500);

        carro.acelerar(); // "Carro acelerando rapidamente!"
        moto.acelerar();  // "Veículo acelerando..."

        carro.abrirPortaMalas();
        moto.empinar();
    }
}
```

3. Polimorfismo

Permite que objetos de diferentes classes respondam ao mesmo método de formas diferentes.

```java
// Interface para polimorfismo
public interface Animal {
    void emitirSom();
    void mover();
}

// Implementações concretas
public class Cachorro implements Animal {
    @Override
    public void emitirSom() {
        System.out.println("Au Au!");
    }

    @Override
    public void mover() {
        System.out.println("Cachorro correndo");
    }

    public void abanarRabo() {
        System.out.println("Abanando o rabo");
    }
}

public class Gato implements Animal {
    @Override
    public void emitirSom() {
        System.out.println("Miau!");
    }

    @Override
    public void mover() {
        System.out.println("Gato andando silenciosamente");
    }

    public void arranhar() {
        System.out.println("Arranhando");
    }
}

public class Passaro implements Animal {
    @Override
    public void emitirSom() {
        System.out.println("Piu Piu!");
```

```
    }

    @Override
    public void mover() {
        System.out.println("Pássaro voando");
    }

    public void voar() {
        System.out.println("Voando alto");
    }
}

// Uso com polimorfismo
public class Main {
    public static void main(String[] args) {
        // Array polimórfico
        Animal[] animais = {
            new Cachorro(),
            new Gato(),
            new Passaro()
        };

        // Polimorfismo: cada animal se comporta diferente
        for (Animal animal : animais) {
            animal.emitirSom();
            animal.mover();
            System.out.println("---");
        }

        // Downcasting para métodos específicos
        Animal animal = new Cachorro();
        if (animal instanceof Cachorro) {
            Cachorro cachorro = (Cachorro) animal;
            cachorro.abanarRabo();
        }
    }
}
```

4. Abstração

Permite criar classes e métodos abstratos que definem comportamentos sem implementação completa.

```java
// Classe abstrata - não pode ser instanciada
public abstract class Funcionario {
    protected String nome;
    protected double salarioBase;
```

```
    public Funcionario(String nome, double salarioBase) {
        this.nome = nome;
        this.salarioBase = salarioBase;
    }

    // Método abstrato - deve ser implementado pelas subclasses
    public abstract double calcularSalario();

    // Método concreto
    public void trabalhar() {
        System.out.println(nome + " está trabalhando...");
    }

    // Getter
    public String getNome() {
        return nome;
    }
}

// Classes concretas que implementam a abstração
public class Desenvolvedor extends Funcionario {
    private int horasExtras;

    public Desenvolvedor(String nome, double salarioBase, int horasExtras) {
        super(nome, salarioBase);
        this.horasExtras = horasExtras;
    }

    @Override
    public double calcularSalario() {
        return salarioBase + (horasExtras * 50);
    }

    public void programar() {
        System.out.println("Desenvolvendo código...");
    }
}

public class Gerente extends Funcionario {
    private double bonus;

    public Gerente(String nome, double salarioBase, double bonus) {
        super(nome, salarioBase);
        this.bonus = bonus;
    }

    @Override
    public double calcularSalario() {
        return salarioBase + bonus;
```

```
    }

    public void gerenciarEquipe() {
        System.out.println("Gerenciando equipe...");
    }
}

// Interface abstrata
public interface Autenticavel {
    boolean autenticar(String senha);
    void alterarSenha(String novaSenha);
}

// Classe que implementa interface
public class Usuario implements Autenticavel {
    private String senha;

    @Override
    public boolean autenticar(String senha) {
        return this.senha.equals(senha);
    }

    @Override
    public void alterarSenha(String novaSenha) {
        this.senha = novaSenha;
    }
}

// Uso
public class Main {
    public static void main(String[] args) {
        // Funcionario func = new Funcionario(); // ERRO: classe abstrata

        Funcionario dev = new Desenvolvedor("João", 5000, 10);
        Funcionario gerente = new Gerente("Maria", 8000, 2000);

        System.out.println("Salário dev: " + dev.calcularSalario());
        System.out.println("Salário gerente: " + gerente.calcularSalario());

        dev.trabalhar();
        gerente.trabalhar();
    }
}
```

Resumo dos Conceitos:

Conceito	Propósito	Exemplo
Encapsulamento	Proteger dados e controlar acesso	Atributos privados com getters/setters
Herança	Reutilizar código e criar hierarquias	`class Carro extends Veiculo`
Polimorfismo	Comportamentos diferentes para mesmo método	`Animal.emitirSom()`varia por tipo
Abstração	Definir contratos sem implementação completa	Classes/métodos abstratos e interfaces

1. Encapsulamento - Exemplos Avançados

Exemplo 1: Sistema de Login com Validação

```
public class Usuario {
    private String username;
    private String password;
    private int tentativasLogin;
    private boolean bloqueado;

    public Usuario(String username, String password) {
        this.username = username;
        setPassword(password); // Usa setter no construtor
        this.tentativasLogin = 0;
        this.bloqueado = false;
    }

    public boolean login(String password) {
        if (bloqueado) {
            System.out.println("Usuário bloqueado!");
            return false;
        }

        if (this.password.equals(hashPassword(password))) {
            tentativasLogin = 0;
            System.out.println("Login bem-sucedido!");
            return true;
        } else {
            tentativasLogin++;
            if (tentativasLogin >= 3) {
```

```
                bloqueado = true;
                System.out.println("Usuário bloqueado após 3 tentativas falhas!");
            }
            return false;
        }
    }

    public void desbloquearUsuario(String adminPassword) {
        if ("admin123".equals(adminPassword)) {
            bloqueado = false;
            tentativasLogin = 0;
            System.out.println("Usuário desbloqueado!");
        }
    }

    private String hashPassword(String password) {
        // Simulação de hash (em aplicação real usar bcrypt, etc.)
        return Integer.toString(password.hashCode());
    }

    // Getters e Setters com validação
    public void setPassword(String password) {
        if (password.length() >= 8) {
            this.password = hashPassword(password);
        } else {
            throw new IllegalArgumentException("Senha deve ter pelo menos 8 caracteres");
        }
    }

    public String getUsername() {
        return username;
    }

    public boolean isBloqueado() {
        return bloqueado;
    }
}

// Uso
public class Main {
    public static void main(String[] args) {
        Usuario user = new Usuario("joao", "senha123");

        user.login("senhaErrada");
        user.login("senhaErrada");
        user.login("senhaErrada"); // Bloqueia após 3 tentativas

        user.desbloquearUsuario("admin123");
        user.login("senha123"); // Login bem-sucedido
```

```
    }
}
```

Exemplo 2: Carrinho de Compras

java
Copy
Download

```java
import java.util.ArrayList;
import java.util.List;

public class CarrinhoCompras {
	private List<Item> itens;
	private double desconto;

	public CarrinhoCompras() {
		this.itens = new ArrayList<>();
		this.desconto = 0;
	}

	public void adicionarItem(Produto produto, int quantidade) {
		if (quantidade <= 0) {
			throw new IllegalArgumentException("Quantidade deve ser positiva");
		}

		// Verifica se produto já existe no carrinho
		for (Item item : itens) {
			if (item.getProduto().equals(produto)) {
				item.setQuantidade(item.getQuantidade() + quantidade);
				return;
			}
		}

		itens.add(new Item(produto, quantidade));
	}

	public void removerItem(Produto produto, int quantidade) {
		if (quantidade <= 0) return;

		for (Item item : itens) {
			if (item.getProduto().equals(produto)) {
				int novaQuantidade = item.getQuantidade() - quantidade;
				if (novaQuantidade <= 0) {
					itens.remove(item);
				} else {
					item.setQuantidade(novaQuantidade);
				}
				return;
			}
		}
	}

	public double calcularTotal() {
```

```
        double total = 0;
        for (Item item : itens) {
            total += item.getSubtotal();
        }
        return total * (1 - desconto);
    }

    public void aplicarDesconto(double percentual) {
        if (percentual < 0 || percentual > 50) {
            throw new IllegalArgumentException("Desconto deve estar entre 0% e 50%");
        }
        this.desconto = percentual / 100;
    }

    // Classe interna encapsulada
    private class Item {
        private Produto produto;
        private int quantidade;

        public Item(Produto produto, int quantidade) {
            this.produto = produto;
            this.quantidade = quantidade;
        }

        public double getSubtotal() {
            return produto.getPreco() * quantidade;
        }

        // Getters
        public Produto getProduto() { return produto; }
        public int getQuantidade() { return quantidade; }
        public void setQuantidade(int quantidade) {
            this.quantidade = quantidade;
        }
    }
}

class Produto {
    private String nome;
    private double preco;

    public Produto(String nome, double preco) {
        this.nome = nome;
        this.preco = preco;
    }

    public double getPreco() { return preco; }
    public String getNome() { return nome; }
}
```

2. Herança - Exemplos Avançados

Exemplo 1: Sistema de Formas Geométricas

java
Copy
Download

```
// Classe abstrata base
public abstract class Forma {
	protected String cor;

	public Forma(String cor) {
		this.cor = cor;
	}

	// Métodos abstratos
	public abstract double calcularArea();
	public abstract double calcularPerimetro();

	// Método concreto
	public void descrever() {
		System.out.println("Forma " + cor + " - Área: " + calcularArea());
	}
}

// Herança com construtores
public class Circulo extends Forma {
	private double raio;

	public Circulo(String cor, double raio) {
		super(cor);
		this.raio = raio;
	}

	@Override
	public double calcularArea() {
		return Math.PI * raio * raio;
	}

	@Override
	public double calcularPerimetro() {
		return 2 * Math.PI * raio;
	}
}

public class Retangulo extends Forma {
	private double largura;
	private double altura;

	public Retangulo(String cor, double largura, double altura) {
		super(cor);
		this.largura = largura;
```

```
        this.altura = altura;
    }

    @Override
    public double calcularArea() {
        return largura * altura;
    }

    @Override
    public double calcularPerimetro() {
        return 2 * (largura + altura);
    }
}

// Herança múltipla com interfaces
public class Quadrado extends Retangulo {
    public Quadrado(String cor, double lado) {
        super(cor, lado, lado);
    }
}

// Uso
public class Main {
    public static void main(String[] args) {
        Forma[] formas = {
            new Circulo("Vermelho", 5),
            new Retangulo("Azul", 4, 6),
            new Quadrado("Verde", 5)
        };

        for (Forma forma : formas) {
            forma.descrever();
            System.out.println("Perímetro: " + forma.calcularPerimetro());
            System.out.println("---");
        }
    }
}
```

Exemplo 2: Sistema de Funcionários com Hierarquia

java
Copy
Download

```
public class Funcionario {
    protected String nome;
    protected double salarioBase;
    protected int tempoServico;

    public Funcionario(String nome, double salarioBase, int tempoServico) {
        this.nome = nome;
        this.salarioBase = salarioBase;
        this.tempoServico = tempoServico;
    }
```

```
    public double calcularSalario() {
        return salarioBase + (tempoServico * 100);
    }

    public void trabalhar() {
        System.out.println(nome + " está trabalhando");
    }
}

public class Gerente extends Funcionario {
    private double bonus;

    public Gerente(String nome, double salarioBase, int tempoServico, double bonus) {
        super(nome, salarioBase, tempoServico);
        this.bonus = bonus;
    }

    @Override
    public double calcularSalario() {
        return super.calcularSalario() + bonus;
    }

    public void gerenciarEquipe() {
        System.out.println(nome + " está gerenciando a equipe");
    }
}

public class Diretor extends Gerente {
    private double participacaoLucros;

    public Diretor(String nome, double salarioBase, int tempoServico,
                   double bonus, double participacaoLucros) {
        super(nome, salarioBase, tempoServico, bonus);
        this.participacaoLucros = participacaoLucros;
    }

    @Override
    public double calcularSalario() {
        return super.calcularSalario() + participacaoLucros;
    }

    public void tomarDecisoesEstrategicas() {
        System.out.println(nome + " tomando decisões estratégicas");
    }
}
```

3. Polimorfismo - Exemplos Avançados

Exemplo 1: Sistema de Pagamentos

java

Copy
Download

```
interface MetodoPagamento {
	boolean processarPagamento(double valor);
	String getStatus();
}

class CartaoCredito implements MetodoPagamento {
	private String numeroCartao;
	private boolean ativo;

	public CartaoCredito(String numeroCartao) {
		this.numeroCartao = numeroCartao;
		this.ativo = true;
	}

	@Override
	public boolean processarPagamento(double valor) {
		if (!ativo || valor <= 0) return false;
		System.out.println("Processando pagamento de R$" + valor + " via cartão");
		return true;
	}

	@Override
	public String getStatus() {
		return ativo ? "Cartão ativo" : "Cartão inativo";
	}
}

class PayPal implements MetodoPagamento {
	private String email;
	private double saldo;

	public PayPal(String email, double saldo) {
		this.email = email;
		this.saldo = saldo;
	}

	@Override
	public boolean processarPagamento(double valor) {
		if (saldo >= valor) {
			saldo -= valor;
			System.out.println("Pagamento PayPal de R$" + valor + " processado");
			return true;
		}
		return false;
	}

	@Override
	public String getStatus() {
		return "Saldo: R$" + saldo;
	}
}

class TransferenciaBancaria implements MetodoPagamento {
```

```
    @Override
    public boolean processarPagamento(double valor) {
        System.out.println("Transferência bancária de R$" + valor + " iniciada");
        return true;
    }

    @Override
    public String getStatus() {
        return "Transferência pendente";
    }
}

// Uso polimórfico
public class Loja {
    public void processarCompra(MetodoPagamento pagamento, double valor) {
        System.out.println("Tentando processar pagamento...");

        if (pagamento.processarPagamento(valor)) {
            System.out.println("Pagamento aprovado!");
            System.out.println("Status: " + pagamento.getStatus());
        } else {
            System.out.println("Pagamento recusado!");
        }
    }

    public static void main(String[] args) {
        Loja loja = new Loja();

        MetodoPagamento[] metodos = {
            new CartaoCredito("1234-5678-9012-3456"),
            new PayPal("cliente@email.com", 1000),
            new TransferenciaBancaria()
        };

        for (MetodoPagamento metodo : metodos) {
            loja.processarCompra(metodo, 250);
            System.out.println("---");
        }
    }
}
```

Exemplo 2: Sistema de Notificações

java
Copy
Download

```
interface Notificacao {
    void enviar(String mensagem);
    boolean verificarStatus();
}

class Email implements Notificacao {
    private String destinatario;
```

```
	public Email(String destinatario) {
		this.destinatario = destinatario;
	}

	@Override
	public void enviar(String mensagem) {
		System.out.println("Enviando email para " + destinatario + ": " + mensagem);
	}

	@Override
	public boolean verificarStatus() {
		return true; // Simulação
	}
}

class SMS implements Notificacao {
	private String telefone;

	public SMS(String telefone) {
		this.telefone = telefone;
	}

	@Override
	public void enviar(String mensagem) {
		System.out.println("Enviando SMS para " + telefone + ": " + mensagem);
	}

	@Override
	public boolean verificarStatus() {
		return true;
	}
}

class PushNotification implements Notificacao {
	private String dispositivoId;

	public PushNotification(String dispositivoId) {
		this.dispositivoId = dispositivoId;
	}

	@Override
	public void enviar(String mensagem) {
		System.out.println("Enviando push para " + dispositivoId + ": " + mensagem);
	}

	@Override
	public boolean verificarStatus() {
		return true;
	}
}

// Serviço que usa polimorfismo
class ServicoNotificacao {
	public void enviarParaTodos(Notificacao[] notificacoes, String mensagem) {
		for (Notificacao notificacao : notificacoes) {
```

```
                notificacao.enviar(mensagem);
            }
        }
}
```

4. Abstração - Exemplos Avançados

Exemplo 1: Sistema de Banco de Dados

java
Copy
Download

```
// Interface para abstração de banco de dados
public interface BancoDados {
    void conectar(String url, String usuario, String senha);
    void desconectar();
    Resultado consultar(String query);
    int executar(String comando);
    boolean estaConectado();
}

// Implementação concreta para MySQL
public class MySQL implements BancoDados {
    private boolean conectado;

    @Override
    public void conectar(String url, String usuario, String senha) {
        System.out.println("Conectando ao MySQL: " + url);
        this.conectado = true;
    }

    @Override
    public void desconectar() {
        System.out.println("Desconectando do MySQL");
        this.conectado = false;
    }

    @Override
    public Resultado consultar(String query) {
        if (!conectado) throw new IllegalStateException("Não conectado");
        System.out.println("Executando query MySQL: " + query);
        return new Resultado(); // Simulação
    }

    @Override
    public int executar(String comando) {
        if (!conectado) throw new IllegalStateException("Não conectado");
        System.out.println("Executando comando MySQL: " + comando);
        return 1; // Simulação
    }
```

```
	@Override
	public boolean estaConectado() {
		return conectado;
	}
}

// Implementação concreta para PostgreSQL
public class PostgreSQL implements BancoDados {
	private boolean conectado;

	@Override
	public void conectar(String url, String usuario, String senha) {
		System.out.println("Conectando ao PostgreSQL: " + url);
		this.conectado = true;
	}

	@Override
	public void desconectar() {
		System.out.println("Desconectando do PostgreSQL");
		this.conectado = false;
	}

	@Override
	public Resultado consultar(String query) {
		if (!conectado) throw new IllegalStateException("Não conectado");
		System.out.println("Executando query PostgreSQL: " + query);
		return new Resultado();
	}

	@Override
	public int executar(String comando) {
		if (!conectado) throw new IllegalStateException("Não conectado");
		System.out.println("Executando comando PostgreSQL: " + comando);
		return 1;
	}

	@Override
	public boolean estaConectado() {
		return conectado;
	}
}

// Classe de resultado (simulação)
class Resultado {
	// Simulação de resultado
}

// Uso com abstração
public class Aplicacao {
	private BancoDados banco;

	public Aplicacao(BancoDados banco) {
		this.banco = banco;
	}
```

```
    public void executarOperacoes() {
        banco.conectar("jdbc:mysql://localhost:3306/meubanco", "user", "pass");

        if (banco.estaConectado()) {
            banco.executar("CREATE TABLE IF NOT EXISTS usuarios (id INT, nome
VARCHAR(100))");
            Resultado resultado = banco.consultar("SELECT * FROM usuarios");

            banco.desconectar();
        }
    }
}
```

Exemplo 2: Sistema de Logs

java
Copy
Download

```
// Abstração para logger
public abstract class Logger {
    protected String nome;
    protected Level nivel;

    public Logger(String nome) {
        this.nome = nome;
        this.nivel = Level.INFO;
    }

    public abstract void log(Level nivel, String mensagem);
    public abstract void flush();
    public abstract void close();

    // Métodos concretos usando métodos abstratos
    public void info(String mensagem) {
        log(Level.INFO, mensagem);
    }

    public void error(String mensagem) {
        log(Level.ERROR, mensagem);
    }

    public void debug(String mensagem) {
        if (nivel.ordinal() <= Level.DEBUG.ordinal()) {
            log(Level.DEBUG, mensagem);
        }
    }

    public void setNivel(Level nivel) {
        this.nivel = nivel;
    }
}

// Implementações concretas
```

```
public class FileLogger extends Logger {
	public FileLogger(String nome) {
		super(nome);
	}

	@Override
	public void log(Level nivel, String mensagem) {
		System.out.println("[" + nivel + "] " + nome + ": " + mensagem + " (arquivo)");
	}

	@Override
	public void flush() {
		System.out.println("Flush do arquivo");
	}

	@Override
	public void close() {
		System.out.println("Fechando arquivo de log");
	}
}

public class ConsoleLogger extends Logger {
	public ConsoleLogger(String nome) {
		super(nome);
	}

	@Override
	public void log(Level nivel, String mensagem) {
		System.out.println("[" + nivel + "] " + nome + ": " + mensagem);
	}

	@Override
	public void flush() {
		// Não faz nada para console
	}

	@Override
	public void close() {
		// Não faz nada para console
	}
}

public class DatabaseLogger extends Logger {
	public DatabaseLogger(String nome) {
		super(nome);
	}

	@Override
	public void log(Level nivel, String mensagem) {
		System.out.println("[" + nivel + "] " + nome + ": " + mensagem + " (banco)");
	}

	@Override
	public void flush() {
		System.out.println("Commit no banco");
	}
```

```
	@Override
	public void close() {
		System.out.println("Fechando conexão com banco");
	}
}

enum Level {
	DEBUG, INFO, WARN, ERROR
}

// Uso
public class Main {
	public static void main(String[] args) {
		Logger[] loggers = {
			new FileLogger("app"),
			new ConsoleLogger("app"),
			new DatabaseLogger("app")
		};

		for (Logger logger : loggers) {
			logger.info("Aplicação iniciada");
			logger.error("Erro crítico");
			logger.close();
			System.out.println("---");
		}
	}
}
```