1. Problemas encontrados no código

Classe Pedido

- Os atributos são públicos, o que quebra o encapsulamento.
- O status é uma string solta (“novo”, “fechado”), o que dificulta controle e padronização.
- Os itens são armazenados apenas como strings, sem preço associado; isso força o cálculo do total a ser feito manualmente.
- A lógica de total é acoplada diretamente ao método addItem, tornando difícil aplicar regras como descontos ou impostos.
- Falta validação de parâmetros (cliente vazio, preço negativo etc.).

Classe PedidoRepository

- O método buscarPorId retorna null, o que pode gerar NullPointerException.
- Não há controle de duplicidade de IDs.
- O método buscarTodos devolve a lista interna diretamente, o que permite que seja alterada de fora.

Classe PedidoService

- Cria o repositório internamente, gerando acoplamento forte.
- Mistura responsabilidades: regras de negócio e impressão no console.
- Métodos não tratam adequadamente casos de erro (por exemplo, pedido inexistente).
- O método criarPedido já cria o pedido com um item inicial, o que não é muito flexível.

Main

- Depende diretamente dos prints feitos no service.
- Não separa apresentação da lógica de negócio.

---

2. Versão revisada do código

```
import java.util.*;

// ---------- MODELO ----------

enum StatusPedido {
NOVO, FECHADO;
}

class ItemPedido {
private final String nome;
private final double preco;

public ItemPedido(String nome, double preco) {
if (nome == null || nome.isBlank()) {
throw new IllegalArgumentException("Nome do item não pode ser vazio.");
}
if (preco < 0) {
throw new IllegalArgumentException("Preço não pode ser negativo.");
}
this.nome = nome;
this.preco = preco;
}

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

@Override
```

```
public String toString() {
return nome + " (R$ " + preco + ")";
}
}

class Pedido {
private final int id;
private final String cliente;
private final List<ItemPedido> itens = new ArrayList<>();
private StatusPedido status = StatusPedido.NOVO;

public Pedido(int id, String cliente) {
this.id = id;
this.cliente = cliente;
}

public void adicionarItem(ItemPedido item) {
itens.add(item);
}

public double calcularTotal() {
return itens.stream()
.mapToDouble(ItemPedido::getPreco)
.sum();
}

public void fechar() {
status = StatusPedido.FECHADO;
```

```
}

public int getId() { return id; }
public String getCliente() { return cliente; }
public List<ItemPedido> getItens() { return List.copyOf(itens); }
public StatusPedido getStatus() { return status; }
}
```

```
// ---------- REPOSITÓRIO ----------

class PedidoRepository {
private final Map<Integer, Pedido> pedidos = new HashMap<>();

public void salvar(Pedido pedido) {
if (pedidos.containsKey(pedido.getId())) {
throw new IllegalArgumentException("Pedido com esse ID já existe.");
}
pedidos.put(pedido.getId(), pedido);
}

public Optional<Pedido> buscarPorId(int id) {
return Optional.ofNullable(pedidos.get(id));
}

public List<Pedido> buscarTodos() {
return new ArrayList<>(pedidos.values());
}
}
```

```
// ---------- SERVIÇO ----------

class PedidoService {
private final PedidoRepository repository;

public PedidoService(PedidoRepository repository) {
this.repository = repository;
}

public void criarPedido(int id, String cliente) {
Pedido pedido = new Pedido(id, cliente);
repository.salvar(pedido);
}

public void adicionarItem(int idPedido, String nome, double preco) {
Pedido pedido = repository.buscarPorId(idPedido)
.orElseThrow(() -> new NoSuchElementException("Pedido não encontrado"));
pedido.adicionarItem(new ItemPedido(nome, preco));
}

public void fecharPedido(int id) {
Pedido pedido = repository.buscarPorId(id)
.orElseThrow(() -> new NoSuchElementException("Pedido não encontrado"));
pedido.fechar();
}

public double calcularTotal(int id) {
return repository.buscarPorId(id)
```

```
.map(Pedido::calcularTotal)
.orElseThrow(() -> new NoSuchElementException("Pedido não encontrado"));
}

public String gerarResumoPedido(int id) {
Pedido pedido = repository.buscarPorId(id)
.orElseThrow(() -> new NoSuchElementException("Pedido não encontrado"));

return """
Pedido %d
Cliente: %s
Itens: %s
Total: R$ %.2f
Status: %s
""".formatted(
pedido.getId(),
pedido.getCliente(),
pedido.getItens(),
pedido.calcularTotal(),
pedido.getStatus()
);
}
}



// ---------- MAIN ----------
```

```
public class Main {
public static void main(String[] args) {
PedidoRepository repo = new PedidoRepository();
PedidoService service = new PedidoService(repo);

service.criarPedido(1, "João");
service.adicionarItem(1, "Pizza", 30.0);

service.criarPedido(2, "Maria");
service.adicionarItem(2, "Hamburguer", 25.0);

service.fecharPedido(1);

System.out.println(service.gerarResumoPedido(1));
System.out.println(service.gerarResumoPedido(2));
}
}
```

---

3. Justificativa das principais alterações

Primeiro, os atributos de Pedido deixaram de ser públicos para preservar o encapsulamento. Isso evita mudanças indevidas de estado fora da própria classe. O status foi transformado em um enum em vez de strings soltas, porque isso reduz erros e deixa mais claro quais estados são permitidos.

O cálculo do total passou a ser derivado dos itens, em vez de um valor acumulado manualmente, porque assim a lógica fica centralizada e mais fácil de manter caso regras novas precisem ser incluídas. Também foi criada a classe ItemPedido para separar nome e preço, o que deixa o código mais organizado e evita misturar dados heterogêneos.

No repositório, o retorno agora é Optional<Pedido> para evitar o uso de null. Além disso, o repositório usa um Map para permitir busca mais eficiente e evitar duplicidades de IDs. A lista interna não é mais exposta diretamente.

O serviço deixou de imprimir informações diretamente e passou apenas a gerar dados prontos para exibição. Isso separa as camadas de lógica e apresentação. Também foram adicionadas validações e exceções para pedidos inexistentes. Outra mudança importante foi receber o repositório via construtor, diminuindo o acoplamento.

Por fim, a classe Main ficou mais simples e passou a depender apenas dos métodos públicos do serviço, mantendo sua função apenas como ponto de execução.