TÓPICO 04: SINGLETON

Design Patterns - Professor Ramon Venson - SATC 2026.1

Roteiro

- Ideia do Singleton
- Problema que resolve
- Implementação básica em Java
- Cuidados e anti-padrão
- Variações úteis

O que é Singleton?

- Garante **apenas uma instância** de uma classe
- Fornece um **ponto global de acesso**
- Muito usado para:
  - configurações
  - log central
  - gerenciadores/caches únicos

Problema

Alguns recursos devem ser **únicos** na aplicação:

- Configurações da aplicação
- Serviço de log
- Gerenciador de conexões

Se cada módulo fizer `new`:

- configurações diferentes espalhadas
- consumo excessivo de recursos
- difícil coordenar quem manda no estado

Ideia da Solução

- **Construtor privado** → ninguém faz `new` por fora
- **Campo estático** → guarda a instância única
- **Método estático** (`getInstancia`) → devolve sempre o mesmo objeto

Fluxo:

1. Primeira chamada → cria a instância
2. Próximas chamadas → reutilizam a mesma instância

Singleton simples em Java

```java
public class ConfiguracaoSistema {
  private static ConfiguracaoSistema instancia;

  private ConfiguracaoSistema() {
    // carregar config, por exemplo
  }

  public static ConfiguracaoSistema getInstancia() {
    if (instancia == null) {
      instancia = new ConfiguracaoSistema();
    }
    return instancia;
  }
}
```

Uso:

```
ConfiguracaoSistema c1 = ConfiguracaoSistema.getInstancia();
ConfiguracaoSistema c2 = ConfiguracaoSistema.getInstancia();
System.out.println(c1 == c2); // true
```

Analogia

- **Central de emergências** (tipo 190/192/193):
  - Vários telefones, uma mesma central
  - Vários módulos do sistema, uma mesma instância
- Não faz sentido cada telefone ter sua própria “mini central”
- Singleton funciona como esse **serviço único** na aplicação

Quando usar Singleton?

- Precisa de **uma única instância**:
  - config global
  - logger
  - cache em memória
- Precisa de **ponto de acesso central**
- Cuidado: use onde unicidade faz sentido de verdade (não por preguiça)

Quando é problema? (Anti-padrão)

- Vira **variável global chique**
- Aumenta **acoplamento** (todo mundo chama `getInstancia()`)
- **Dificulta testes** (estado global entre testes)
- Em cenários multiusuário ou multi-thread pode gerar:
  - dados misturados
  - condições de corrida

Regra: use só quando unicidade é um requisito real.

Singleton e Testes

```
public class Logger {
  private static Logger instancia = new Logger();
  private Logger() { }

  public static Logger getInstancia() { return instancia; }

  public void log(String msg) {
    System.out.println(msg);
  }
}
```

Problema em testes:

- Não dá para trocar `Logger` por um mock facilmente
- Estado e saída se misturam entre testes

Solução comum: preferir **injeção de dependência** em vez de Singletons diretos.

Singleton com enum (forma segura)

```
public enum CentralEmergencia {
  INSTANCIA;

  public void atender(String tipo) {
    System.out.println("Atendendo: " + tipo);
  }
}
```

Uso:

```
CentralEmergencia c1 = CentralEmergencia.INSTANCIA;
CentralEmergencia c2 = CentralEmergencia.INSTANCIA;
System.out.println(c1 == c2); // true
```

- Thread-safe
- Simples
- Lida bem com serialização

Prós

- Garante **uma única instância**
- Acesso simples: `MinhaClasse.getInstancia()`
- Pode usar **lazy loading** (criar só quando precisar)
- Centraliza configuração/estado compartilhado

Contras

- **Acoplamento global**
- Dificulta **testes unitários**
- Pode violar SRP (vira “objeto Deus”)
- Implementação ingênua não é thread-safe

Resumo

- Singleton: **uma instância + acesso global controlado**
- Útil para recursos realmente **únicos**
- Use com cuidado: fácil virar anti-padrão
- Considere alternativas:
  - injeção de dependências
  - gerenciamento por contêiner (ex.: Spring singleton scope)