

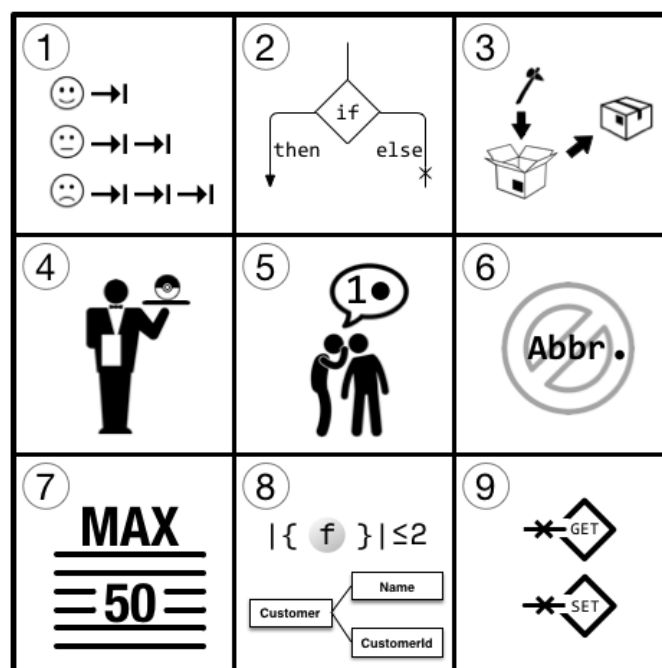
## Object Calistenics

Object Calisthenics é um conjunto de **9 princípios de programação orientada a objetos**, com o objetivo de incentivar a escrita de código mais limpo, modular e sustentável. A ideia é aplicar restrições artificiais para forçar boas práticas.

Os **Object Calisthenics** foram apresentados em **2008**, no livro *The ThoughtWorks Anthology: Essays on Software Technology and Innovation*, da editora Pragmatic Bookshelf.

O autor do capítulo foi **Jeff Bay**, que propôs esses **9 exercícios** como uma forma de treinar disciplina em programação orientada a objetos, assim como a calistenia é um treino físico de disciplina corporal.

Ou seja, não nasceu como uma “regra formal da engenharia de software”, mas como um **conjunto de práticas pedagógicas** para forçar o desenvolvedor a escrever código mais limpo e orientado a objetos de verdade.



Aqui estão os nove princípios:

### 1. Só um nível de indentação por método

- Evite estruturas de código muito aninhadas (ifs dentro de loops dentro de switches etc.);
- Quebra em métodos menores facilita a leitura e manutenção.

### 2. Não use a palavra-chave else

- O else costuma indicar complexidade ou lógica condicional excessiva;
- Prefira *early return*, polimorfismo ou tratamento de exceções para clareza.

### 3. Envolver todos os tipos primitivos e strings

- Em vez de usar `int`, `double` ou `String` diretamente, crie objetos que representem conceitos;
- Exemplo: em vez de `int idade`, use uma classe `Idade`;
- Isso adiciona semântica, validações e clareza.

### 4. Use apenas uma instrução de ponto por linha

- Respeite o princípio da Lei de Deméter;
- Em vez de `cliente.getEndereco().getCidade()`, encapsule a lógica em métodos apropriados (`cliente.getCidade()`).

### 5. Não abrevie

- Use nomes descritivos e completos para variáveis, métodos e classes;
- Evita ambiguidades e melhora a compreensão do código.

### 6. Mantenha todas as entidades pequenas

- Classes pequenas, métodos curtos e responsabilidades únicas;
- Cada unidade de código deve ter um propósito claro.

### 7. Não use mais de duas variáveis de instância por classe

- Incentiva a composição de objetos;
- Classes ficam focadas em um único conceito, evitando “Deus objects”.

### 8. Use coleções de primeira classe

- Não manipule coleções diretamente em várias partes do código;
- Encapsule coleções em classes próprias que representem seu significado;
- Exemplo: `class CarrinhoDeCompras` em vez de `List<Item>` espalhada no sistema.

### 9. Sem getters/setters

- Em vez de expor dados, exponha comportamento;
- Substitua `pessoa.getIdade()` por `pessoa.ehMaiorDedade()`;
- Isso evita objetos anêmicos e promove encapsulamento verdadeiro.

Esses princípios não são leis obrigatórias, mas **exercícios de disciplina** para treinar a mentalidade orientada a objetos. Com o tempo, ajudam a criar sistemas mais coesos, de fácil evolução e menos frágeis.

## Detalhamentos

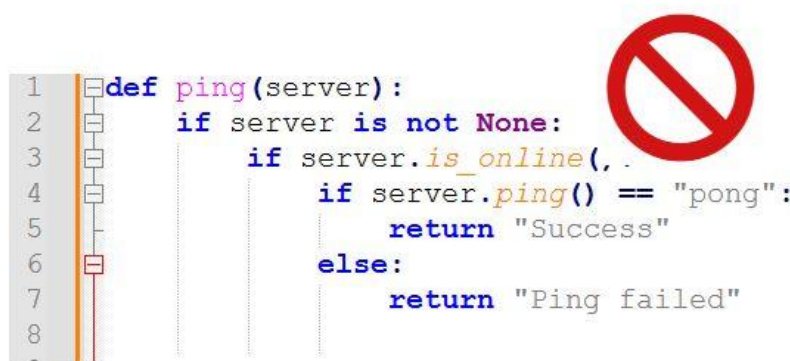
1. **"Só um nível de indentação por método."** (Evitar estruturas muito aninhadas e quebrar em métodos menores para ganhar clareza e manutenibilidade.).

Os métodos com muitos ifs, for e switch dentro de outros criam **pirâmides de código** (aninhamento profundo). Código wide dificulta a leitura e aumenta a chance de erros.

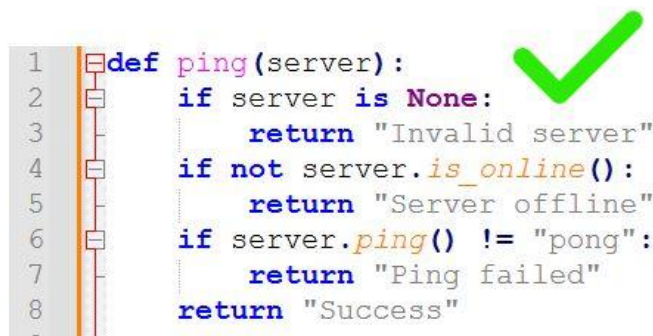
A proposta é: **um único nível de indentação por método**, quebrando lógicas em **métodos auxiliares** ou usando **cláusulas de guarda** (*guard clauses*).

### 2 – "Não use else."

- O else muitas vezes deixa o código mais aninhado;
- A alternativa é usar **cláusulas de guarda** (*guard clauses*) com return para sair cedo (*fail fast*);
- Isso mantém o **caminho feliz** no final, limpo e direto.



```
1 def ping(server):
2     if server is not None:
3         if server.is_online():
4             if server.ping() == "pong":
5                 return "Success"
6             else:
7                 return "Ping failed"
```



```
1 def ping(server):
2     if server is None:
3         return "Invalid server"
4     if not server.is_online():
5         return "Server offline"
6     if server.ping() != "pong":
7         return "Ping failed"
8     return "Success"
```

Código problema: Temos aqui 3rês níveis de indentação (for → if → if). Difícil de ler e manter. Para incluir novas regras, a pirâmide cresce ainda mais.

```
public class ProcessadorPedidos {  
    public void processar(List<Pedido> pedidos) {  
        for (Pedido pedido : pedidos) {  
            if (pedido.estaPago()) {  
                if (pedido.temEstoqueDisponivel()) {  
                    if (!pedido.estaCancelado()) {  
                        System.out.println("Processando pedido: " +  
                                           pedido.getId());  
                    } else {  
                        System.out.println("Pedido cancelado.");  
                    }  
                } else {  
                    System.out.println("Estoque indisponível.");  
                }  
            } else {  
                System.out.println("Pedido não pago.");  
            }  
        }  
    }  
}
```

A ideia de correção é iniciar pelas bordas da indentação e aplicar cláusulas de guarda, tratando as falhas mais cedo. Fazer return cedo (Early Return) e dar clareza ao caminho feliz (happy path).

### 1. Fail Fast (Falhar Rápido)

O princípio do *fail fast* defende que um código deve **detectar e sinalizar problemas o mais cedo possível**, interrompendo a execução em vez de tentar continuar em estado inválido.

- A ideia é evitar que o erro se propague e cause efeitos colaterais piores;
- Em vez de mascarar ou adiar o problema, o sistema "falha" imediatamente, forçando o desenvolvedor a tratar a causa.

### 2. Cláusula de Guarda (Guard Clause)

Uma cláusula de guarda é uma **condição no início do método** que verifica casos inválidos ou especiais e retorna imediatamente, evitando o aninhamento de if/else.

- Funciona como uma "sentinela": protege o método de estados inválidos;
- Melhora a legibilidade, pois elimina níveis de indentação.

### 3. Early Return (Retorno Antecipado)

O *early return* é a **aplicação prática** do fail fast e das cláusulas de guarda: consiste em **retornar de um método o mais cedo possível**, assim que uma condição crítica é detectada.

- Evita prolongar a execução quando já sabemos que não faz sentido continuar;
- Garante clareza: os casos inválidos ou triviais são resolvidos logo de início.

### Happy Path (Caminho Feliz)

O *happy path* é o **fluxo principal e desejado de execução**, quando tudo ocorre da forma esperada, sem erros ou condições especiais.

- Ele representa o cenário “feliz”, em que todas as pré-condições são atendidas;
- A clareza do *happy path* é maior quando usamos *fail fast*, *guard clauses* e *early return*, porque os desvios (erros, exceções, condições inválidas) ficam tratados logo no início, deixando o fluxo normal limpo e legível.

Resumo:

- **Fail Fast** : princípio: falhe cedo quando algo está errado;
- **Cláusula de Guarda** : técnica: proteja o método com condições no início;
- **Early Return** : prática: retorne logo que detectar a condição inválida;
- **Happy Path** : resultado: o fluxo principal fica limpo, direto e fácil de ler.

```
public class ProcessadorPedidos {  
    public void processar(List<Pedido> pedidos) {  
        for (Pedido pedido : pedidos) {  
            processarPedido(pedido); // 1 nível de indentação  
        }  
    }  
    private void processarPedido(Pedido pedido) {  
        //Guard Clause  
        //Fail Fast  
        if (!pedido.estaPago()) {  
            System.out.println("Pedido não pago.");  
            //Early return  
            return;  
        }  
  
        if (!pedido.temEstoqueDisponivel()) {  
            System.out.println("Estoque indisponível.");  
            return;  
        }  
  
        if (pedido.estaCancelado()) {  
            System.out.println("Pedido cancelado.");  
            return;  
        }  
        //happy path  
        System.out.println("Processando pedido: " + pedido.getId());  
    }  
}
```

### Vantagens

- **Legibilidade:** código mais fácil de ler e entender;
- **Testabilidade:** métodos menores podem ser testados isoladamente;
- **Manutenção:** adicionar novas regras é mais simples;
- **Evolução natural:** incentiva a extração de comportamentos em novas classes/métodos.

### **Desvantagens (quando aplicado cegamente)**

- **Excesso de métodos curtos:** pode gerar fragmentação, com muitas chamadas pequenas;
- **Curva de aprendizado:** para iniciantes, pode parecer “mais trabalhoso” do que apenas usar if aninhados;
- **Context switching:** às vezes exige navegar entre muitos métodos para entender um fluxo simples.

### 3. Envolver todos os tipos primitivos e strings

- Em vez de usar `int`, `double` ou `String` diretamente, crie objetos que representem conceitos;
- Exemplo: em vez de `int idade`, use uma classe `Idade`;
- Isso adiciona semântica, validações e clareza.

### 4. Use apenas uma instrução de ponto por linha

- Respeite o princípio da Lei de Deméter;
- Em vez de `cliente.getEndereco().getCidade()`, encapsule a lógica em métodos apropriados (`cliente.getCidade()`).

### 5. Não abrevie

- Use nomes descritivos e completos para variáveis, métodos e classes;
- Evita ambiguidades e melhora a compreensão do código.

### 6. Mantenha todas as entidades pequenas

- Classes pequenas, métodos curtos e responsabilidades únicas;
- Cada unidade de código deve ter um propósito claro.

### 7. Não use mais de duas variáveis de instância por classe

- Incentiva a composição de objetos;
- Classes ficam focadas em um único conceito, evitando “Deus objects”.

### 8. Use coleções de primeira classe

- Não manipule coleções diretamente em várias partes do código;
- Encapsule coleções em classes próprias que representem seu significado;
- Exemplo: `class CarrinhoDeCompras` em vez de `List<Item>` espalhada no sistema.

### 9. Sem getters/setters

- Em vez de expor dados, exponha comportamento;
- Substitua `pessoa.getIdade()` por `pessoa.ehMaiorDeIdade()`;
- Isso evita objetos anêmicos e promove encapsulamento verdadeiro.