Nota: Este material complementar, disponível em https://prettore.github.io/lectures.html representa uma cópia resumida de conteúdos bibliográficos disponíveis gratuitamente na Internet.

Comandos de Repetição

Introdução

Os comandos de repetição, também conhecidos como estruturas de repetição ou laços (loops), são recursos fundamentais em programação que permitem executar um bloco de código repetidamente enquanto uma condição específica for verdadeira ou por um número determinado de vezes. Essas estruturas são essenciais para automatizar tarefas repetitivas, processar coleções de dados e implementar algoritmos iterativos. Sem os comandos de repetição, seria necessário duplicar código, tornando os programas maiores, mais difíceis de

manter e menos eficientes. Este material explora os principais tipos de comandos de repetição utilizados em programação, suas sintaxes, aplicações e boas práticas.

Comando while

O comando while é uma estrutura de repetição que executa um bloco de código repetidamente enquanto uma condição específica for verdadeira. O nome "while" vem do inglês e significa "enquanto", indicando que o bloco de código será executado enquanto a condição especificada for verdadeira.

Sintaxe básica do while:

```
while (condição) {
   // bloco de código a ser repetido
}
```

O funcionamento do while segue os seguintes passos:

1. A condição é avaliada.
2. Se a condição for falsa, o bloco de código é ignorado e a execução continua após o bloco while.
3. Se a condição for verdadeira, o bloco de código é executado.
4. Após a execução do bloco, o controle volta ao passo 1.

Exemplo:

```
int contador = 1;
while (contador <= 5) {
   System.out.println("Contador: " + contador);
   contador++;
}
```

Neste exemplo, o bloco de código será executado cinco vezes, imprimindo os valores de 1 a 5. A variável contador é inicializada com 1 e incrementada a cada iteração. Quando contador se torna 6, a condição contador <= 5 se torna falsa, e o loop termina.

O comando while é especialmente útil quando:

- Não sabemos exatamente quantas vezes o loop deve ser repetido.
- O teste deve ser feito antes de iniciar a execução do bloco de código.
- Há casos em que o loop não deve ser executado nenhuma vez (se a condição inicial for falsa).

É importante garantir que a condição do while eventualmente se torne falsa, caso contrário, o loop continuará indefinidamente, criando o que chamamos de "loop infinito". Isso geralmente ocorre quando esquecemos de atualizar a variável usada na condição dentro do bloco de código.

Comando do-while

O comando do-while é uma variação do while que garante que o bloco de código seja executado pelo menos uma vez, independentemente da condição. Isso ocorre porque a condição é verificada após a execução do bloco, e não antes, como no while.

Sintaxe básica do do-while:

```
do {
    // bloco de código a ser repetido
} while (condição);
```

O funcionamento do do-while segue os seguintes passos:
1. O bloco de código é executado.
2. A condição é avaliada.
3. Se a condição for verdadeira, o controle volta ao passo 1.
4. Se a condição for falsa, o loop termina.

Exemplo:

```
int contador = 1;
do {
    System.out.println("Contador: " + contador);
    contador++;
} while (contador <= 5);
```

Neste exemplo, o resultado será o mesmo do exemplo anterior com while, mas a diferença está no comportamento quando a condição inicial é falsa. Por exemplo:

```
int contador = 6;
do {
    System.out.println("Contador: " + contador);
    contador++;
} while (contador <= 5);
```

Neste caso, mesmo que contador já seja maior que 5 inicialmente, o bloco de código será executado uma vez, imprimindo "Contador: 6", antes de verificar a condição e terminar o loop.

O comando do-while é especialmente útil quando:

- Queremos garantir que o bloco de código seja executado pelo menos uma vez.
- O teste deve ser feito após a execução do bloco de código.
- Estamos implementando menus ou solicitando entrada do usuário até que uma condição seja atendida.

A diferença do while para o do-while é que, no do-while, sempre acontece a primeira execução do bloco de comandos e a expressão booleana só é avaliada ao final da primeira execução.

Comando for

O comando for é uma estrutura de repetição mais compacta que combina inicialização, condição e atualização em uma única linha. É especialmente útil quando sabemos exatamente quantas vezes o loop deve ser repetido.

Sintaxe básica do for:

```
for (inicialização; condição; atualização) {
   // bloco de código a ser repetido
}
```

Os componentes do for são:

- Inicialização: executada apenas uma vez, no início do loop. Geralmente usada para inicializar uma variável de controle.
- Condição: avaliada antes de cada iteração. Se for verdadeira, o bloco de código é executado; se for falsa, o loop termina.
- Atualização: executada após cada iteração. Geralmente usada para incrementar ou decrementar a variável de controle.

O funcionamento do for segue os seguintes passos:

1. A inicialização é executada.
2. A condição é avaliada.
3. Se a condição for falsa, o loop termina.
4. Se a condição for verdadeira, o bloco de código é executado.
5. A atualização é executada.
6. O controle volta ao passo 2.

Exemplo:

```
for (int i = 1; i <= 5; i++) {
   System.out.println("Contador: " + i);
}
```

Neste exemplo, a variável i é inicializada com 1, a condição i <= 5 é verificada, e i é incrementado após cada iteração. O resultado será a impressão dos números de 1 a 5.

O comando for é especialmente útil quando:

- Sabemos exatamente quantas vezes o loop deve ser repetido.
- Estamos iterando sobre uma sequência de números.
- Queremos uma sintaxe mais compacta que combine inicialização, condição e atualização.

É possível omitir qualquer um dos componentes do for, mas os ponto-e-vírgulas devem ser mantidos. Por exemplo:

```
int i = 1;
for (; i <= 5; i++) {
    System.out.println("Contador: " + i);
}
```

Neste caso, a inicialização foi omitida porque a variável i já foi inicializada fora do loop.

Comando for-each (for aprimorado)

O comando for-each, também conhecido como "for aprimorado" ou "enhanced for", é uma variação do for tradicional introduzida em linguagens modernas para simplificar a iteração sobre coleções de dados, como arrays e listas.

Sintaxe básica do for-each (em Java):

```
for (Tipo elemento : coleção) {
    // bloco de código a ser repetido
}
```

Onde:
- Tipo: é o tipo dos elementos na coleção.
- elemento: é uma variável que representa o elemento atual da coleção durante cada iteração.
- coleção: é a coleção de dados sobre a qual estamos iterando.

Exemplo:

```
int[] numeros = {1, 2, 3, 4, 5};
for (int numero : numeros) {
    System.out.println("Número: " + numero);
}
```

Neste exemplo, o for-each itera sobre o array numeros, atribuindo cada elemento à variável numero durante cada iteração. O resultado será a impressão de todos os números no array.

O comando for-each é especialmente útil quando:

- Queremos iterar sobre todos os elementos de uma coleção.
- Não precisamos do índice dos elementos durante a iteração.
- Queremos uma sintaxe mais limpa e menos propensa a erros.

A principal limitação do for-each é que não temos acesso ao índice dos elementos durante a iteração, o que pode ser necessário em alguns casos. Além disso, não podemos modificar a coleção durante a iteração usando o for-each.

Comandos break e continue

Os comandos break e continue são usados para controlar o fluxo dentro de loops, permitindo interromper ou pular iterações.
O comando break é usado para sair imediatamente de um loop, independentemente da condição. Quando o break é executado, o controle é transferido para a primeira instrução após o loop.

Exemplo:

```
for (int i = 1; i <= 10; i++) {
    if (i == 5) {
        break;
    }
    System.out.println("Contador: " + i);
}
```

Neste exemplo, o loop será interrompido quando i for igual a 5, resultando na impressão apenas dos números de 1 a 4.
O comando continue é usado para pular a iteração atual e continuar com a próxima. Quando o continue é executado, o controle é transferido de volta para a condição do loop (no caso do while e do-while) ou para a atualização (no caso do for).

Exemplo:

```
for (int i = 1; i <= 10; i++) {
    if (i % 2 == 0) {
        continue;
    }
    System.out.println("Contador: " + i);
}
```

Neste exemplo, o continue será executado quando i for par, resultando na impressão apenas dos números ímpares de 1 a 9.

Loops aninhados

Loops aninhados são loops dentro de outros loops. Para cada iteração do loop externo, o loop interno é executado completamente. Isso é útil para trabalhar com estruturas de dados multidimensionais, como matrizes, ou para realizar operações mais complexas.

Exemplo:

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        System.out.println("i = " + i + ", j = " + j);
    }
}
```

Neste exemplo, para cada valor de i, o loop interno itera sobre todos os valores de j, resultando em 9 combinações diferentes de i e j.
Os loops aninhados são poderosos, mas devem ser usados com cuidado, pois podem levar a problemas de desempenho se o número de iterações for muito grande. Por exemplo, um loop aninhado com três níveis onde cada nível itera 100 vezes resultará em 1.000.000 de iterações totais.

Variáveis contadoras e acumuladoras

Em muitos algoritmos que utilizam loops, é comum usar variáveis contadoras e acumuladoras para rastrear informações durante as iterações.
Variáveis contadoras são usadas para contar o número de ocorrências de um evento ou condição. Geralmente são inicializadas com 0 e incrementadas quando o evento ocorre.

Exemplo:

```
int contador = 0;
for (int i = 1; i <= 100; i++) {
    if (i % 2 == 0) {
        contador++;
    }
}
System.out.println("Quantidade de números pares: " + contador);
```

Neste exemplo, a variável contador é usada para contar quantos números pares existem entre 1 e 100.
Variáveis acumuladoras são usadas para acumular valores durante as iterações. Geralmente são inicializadas com 0 (para soma) ou 1 (para produto) e atualizadas a cada iteração.

Exemplo:

```
int soma = 0;
for (int i = 1; i <= 100; i++) {
    soma += i;
}
System.out.println("Soma dos números de 1 a 100: " + soma);
```

Neste exemplo, a variável soma é usada para acumular a soma dos números de 1 a 100.

Boas práticas no uso de comandos de repetição

Para garantir que seus comandos de repetição sejam claros, eficientes e livres de erros, considere as seguintes boas práticas:

- Evite loops infinitos: Certifique-se de que a condição do loop eventualmente se torne falsa, ou use break para sair do loop em algum ponto.
- Use o tipo de loop apropriado: Use for quando souber o número exato de iterações, while quando a condição de término não for conhecida antecipadamente, e do-while quando o bloco de código precisar ser executado pelo menos uma vez.
- Mantenha os loops simples: Se um loop se tornar muito complexo, considere dividi-lo em loops menores ou extrair parte da lógica para funções separadas.
- Evite modificar a variável de controle dentro do bloco de código em loops for: Isso pode levar a comportamentos inesperados e dificultar a leitura do código.
- Use nomes significativos para variáveis de controle: Em vez de usar i, j, k para todas as variáveis de controle, considere usar nomes que descrevam o propósito da variável.
- Tenha cuidado com loops aninhados: Eles podem levar a problemas de desempenho se o número de iterações for muito grande.
- Use break e continue com moderação: O uso excessivo desses comandos pode tornar o código difícil de entender e manter.
- Considere o uso de algoritmos e estruturas de dados eficientes: Em alguns casos, um algoritmo mais eficiente pode eliminar a necessidade de loops aninhados ou reduzir significativamente o número de iterações.

Conclusão

Os comandos de repetição são ferramentas essenciais em programação, permitindo executar blocos de código repetidamente de forma eficiente e organizada. Neste material, exploramos os principais tipos de comandos de repetição: while, do-while, for e for-each, além de conceitos relacionados como break, continue, loops aninhados e variáveis contadoras e acumuladoras.

Cada tipo de comando de repetição tem suas próprias características, vantagens e limitações, e a escolha entre eles depende do contexto específico e das necessidades do programa. Ao dominar os comandos de

repetição e seguir as boas práticas apresentadas, você estará equipado para criar programas mais eficientes, organizados e fáceis de manter.
Lembre-se de que a prática é fundamental para dominar os comandos de repetição. Experimente diferentes tipos de loops em diferentes cenários para desenvolver uma intuição sobre qual é o mais apropriado para cada situação.

Referências

- 📖 Livros e Apostilas
  - CORMEN, T. H. Introduction to Algorithms. MIT Press.
  - GOODRICH, M. Data Structures and Algorithms in Python.
  - Tenenbaum, A. M. Estruturas de Dados e Algoritmos em C
  - P. Feofiloff. Algoritmos em Linguagem C. Campus-Elsevier, 1a. edição, 2009 H. M. Deitel, P. J. Deitel. C - Como Programar, 6a. edição, Pearson Education, 2011.
  - B. W. Kernighan, D. M. Ritchie. The C Programming Language, 2a. edição, Prentice-Hall, 1988 [Tradução: C - A Linguagem de Programação. Editora Campus, 1989].
  - J. L. Szwarcfiter, L. Markenzon. Estruturas de Dados e seus Algoritmos, 3a. edição, Editora LTC, 2010.
  - W. Celes, R. Cerqueira, J.L. Rangel. Introdução a Estruturas de Dados, 1a. edição, Editora Campus, 2004.
  - N. Ziviani. Projeto de Algoritmos com Implementações em Pascal e C, 3a. edição, Editora Cengage Learning, 2011.
  - T. Cormen, C. Leiserson, R. Rivest, C. Stein. Algoritmos - Teoria e Prática, 3a. edição, Editora Campus, 2012.
  - R. Sedgewick, K. Wayne. Algorithms, 4a. edição, Addison -Wesley, 2011.
  - A. Kelley, I. Pohl. A Book on C, 4a. edição, Addison Wesley, 1998.
- 🌐 Recursos Online
  - Programação C/C++ - Comandos de Repetição - PUCRS - https://www.inf.pucrs.br/~pinho/LaproI/ComandosDeRepeticao/Repeticao.html
  - Estruturas de Repetição - UFPR - https://www.inf.ufpr.br/cursos/ci067/Docs/NotasAula/notas-14_Estruturas_Repeti_c_cao.html
  - Laços de repetição – for, while, do-while - UFPE - https://www.cin.ufpe.br/~luciano/cursos/ce/lacos.pdf
  - Comandos de Repetição - Introdução à Programação - https://ip.oberlan.com/repeticoes/
  - Estruturas de repetições em Java (for, while, do-while, for-each) - https://blog.formacao.dev/estruturas-de-repeticoes-em-java-for-while-e-do-while-for-each/
  - W3Schools - JavaScript Loops
  - GeeksforGeeks - Loops in C/C++
- 📽️ Vídeos e Cursos
  - Curso em Vídeo - Estruturas de Repetição
  - Programação de Computadores - Estruturas de Repetição - UFOP
  - Khan Academy - Introdução à Programação

Isenção de Responsabilidade:

Anexo - Resumo Estruturado/Mapa Mental Gerado por IA

🔹 Introdução

- Estruturas fundamentais para repetir blocos de código.
- Usadas em **tarefas repetitivas, coleções de dados e algoritmos iterativos**.
- Evitam duplicação de código → tornam programas mais eficientes e legíveis.

🔹 Tipos de Laços (Loops)

1. **While**
   - Executa enquanto a condição for **verdadeira**.
   - Teste feito **antes** da execução.
   - Útil quando **não se sabe** o número exato de repetições.
   - Risco: **loop infinito** se a condição nunca se tornar falsa.
2. **Do-While**
   - Executa o bloco **ao menos uma vez**, pois a condição é testada **depois**.
   - Usado em menus, entradas de usuário.
3. **For**
   - Estrutura compacta com **inicialização, condição e atualização**.
   - Ideal quando o número de repetições é **conhecido**.
   - Pode ter componentes omitidos (mas sempre mantém os ;).
4. **For-Each (aprimorado)**
   - Itera diretamente sobre coleções (arrays, listas).
   - Sintaxe mais simples, evita erros.
   - Limitações: não acessa índice nem permite alterar coleção.

🔹 Controle de Fluxo dentro de Loops

- **break** → encerra o loop imediatamente.
- **continue** → pula a iteração atual e segue para a próxima.

🔹 Estruturas Compostas

- **Loops Aninhados**
  - Um loop dentro de outro.
  - Usado em **matrizes, tabelas, cálculos multidimensionais**.
  - Cuidado com desempenho (explosão de iterações).
- **Variáveis Contadoras**

    - Contam ocorrências (ex: contador++).
- **Variáveis Acumuladoras**
    - Somam/produtos ao longo das iterações (ex: soma += valor).

🔹 Boas Práticas

- Evitar **loops infinitos**.
- Escolher o **tipo de loop apropriado**:
    - for → quando sabe número fixo de repetições.
    - while → quando não sabe de antemão.
    - do-while → quando precisa garantir 1ª execução.
- Loops simples > loops complexos.
- Evitar alterar variáveis de controle dentro do loop.
- Usar nomes significativos para variáveis (i, j → apenas em loops curtos).
- Usar **break/continue** com moderação.
- Atenção ao desempenho em loops aninhados.

🔹 Conclusão

- Comandos de repetição são **essenciais** para eficiência e organização.
- Diferentes tipos oferecem **flexibilidade** conforme a situação.
- Boas práticas evitam erros e garantem **código legível e manutenível**.