Aula 5 - Gerenciamento de Memória

Docupedia Export

Author:Silio Leonardo (SO/OPM-TS21-BR)
Date:06-May-2024 14:49

Table of Contents

- Heap, Stack e ponteiros
- Tipos por Referência e Tipos por valor
- Null
- Associação, Agregação e Composição
- Garbage Collector
- Exemplo 1: Fazendo uma Lista Encadeada com Ponteiros para armazenar uma quantidade variável de clientes

1 Heap, Stack e ponteiros

Todo programa java pode guardar seus dados em 2 lugares: O Heap e o Stack. O Heap é uma área na memória onde dados ficam armazenados fora de ordem, e são referenciados para uso. Isso significa que para acessar algo no Heap nós precisamos de algo chamado ponteiro. O ponteiro é uma variável que armazena não o valor, mas sim o endereço de onde algo está no Heap. Pode não parecer, mas ao criarmos um objeto de uma classe estamos usando um ponteiro escondido.
Por outro lado, alguns valores podem ser armazenados na Stack. A stack é uma região onde os valores são colocados sequencialmente. Quando criamos a variável aluno, os dados do objeto Aluno foram colocados no Heap, porém a referência foi colocada na stack. Ao usarmos a variável acessamos a Stack, buscamos a referência e assim vamos buscar os dados na Stack. Não só isso, vários dados como int's e outros tipos primitivos são armazenados na Stack. Quando chamamos uma função, o ponto de retorno, ou seja, para onde a função deve retornar depois de ser executada fica guardada na Stack. E quando chamamos muitas funções a Stack pode crescer tanto que invade a área do Heap resultado no famoso erro Stack Overflow.

2 Tipos por Referência e Tipos por valor

No java existem tipos que são por referência e tipos que são por valor. Isso significa que ao chamar uma função ou fazer qualquer atribuição, os dados são copiados se for um tipo por valor, e caso for um tipo por referência, uma referência é armazenada na nova variável para os mesmos dados. Observe o exemplo a seguir que demonstra essa dinâmica:

```
1  int idade = 16;
2  int idade2 = idade;
3  idade2++;
4  System.out.println(idade); // 16
5  System.out.println(idade2); // 17
6
7  Aluno aluno = new Aluno();
8  aluno.setIdade(16);
9  Aluno aluno2 = aluno;
10 aluno2.setIdade(aluno2.getIdade() + 1);
11 System.out.println(aluno.Idade); // 17
12 System.out.println(aluno2.Idade); // 17
13
14 Aluno aluno = new Aluno();
15 aluno.setIdade(16);
16 Aluno aluno2 = new Aluno();
17 aluno2.setIdade(aluno.getIdade());
18 aluno2.setIdade(aluno2.getIdade() + 1);
19 System.out.println(aluno.Idade); // 16
20 System.out.println(aluno2.Idade); // 17
```

Podemos observar que ao passar o valor 16 de uma variável int para outra, esse valor é copiado. Ao alterar o 'idade2', o 'idade' não é alterado pois a variável 'idade2' tem só uma mera cópia do seu valor. No segundo exemplo, criamos um objeto do tipo Aluno que é por referência, ao escrevermos 'aluno2 = aluno', estamos copiando a referência que a variável 'aluno' tem na Stack para a variável 'aluno2'. Já no terceiro exemplo, os dados de idade que estão no Heap são copiados de um objeto para outro, mas dois objetos diferentes são criados.
Ao chamarmos uma função, fenômenos semelhantes acontecem:

```
1  int numero = 76;
2  alterarNumero(numero);
3  // Aqui número vale 76, seu valor foi copiado e não foi alterado
4
5  Aluno aluno = new Aluno();
6  aluno.setNome("Pamella");
```

```
7   alterarNome(aluno);
8   // Aqui o nome do Aluno é Gilmar, foi passado a referência do objeto que foi alterado dentro da função
9
10  void alterarNumero(int valor)
11  {
12      valor = valor + 1;
13  }
14
15  void alterarNome(Aluno aluno)
16  {
17      aluno.setNome("Gilmar");
18  }
```

Todos os tipos primitivos (que tem todas as letas minúsculas) são tipos por valor. Referências (ponteiros) também são tipos por valor. Todos os valores de objetos que são criados através de classes são tipos por referência.

3 Null

Para tipos por referência, você pode ainda criar ponteiros que não apontam para nada. Isso é útil quando queremos criar variáveis que ainda não tem valor algum. Para isso usamos a palavra reservada 'null'.

```
1  Aluno aluno = null;
2
3  String nome = aluno.Nome; // Erro, aluno é nulo e não podemos ler seu nome, o famoso NullPointerException (erro /exceção
   de ponteiro nulo)
```

4 Associação, Agregação e Composição

Além dos tipos que usamos para Campos e Propriedades de uma classe, também podemos usar um ponteiro a outros tipos. O nome disso é Associação, quando dois tipos estão ligados por uma referência. Ainda temos a Agregação e Composição que essencialmente são a mesma coisa mas com formas de utilização diferente. Enquanto a Associação é fraca e objetos sabem que outros existem mas são independentes, uma Agregação ocorre quando um objeto B está referenciado por um objeto A e o segundo não faz sentido sozinho como objeto sem o A. Uma Composição é quando A não faz sentido sem o B. Abaixo um exemplo de Agregação: A data não faz sentido se não tiver um significado associado ao cliente, mas o cliente continua a ser um cliente mesmo sem data de aniversário.

```
1   Cliente cliente = new Cliente();
2   cliente.setNome("Pamella");
3
4   Data data = new Data();
5   data.setDia(7);
6   data.setMes(6);
7   data.setAno(2000);
8
9   cliente.setDataNascimento(data);
10
11  public class Data
12  {
13      private int dia;
14      private int mes;
15      private int ano;
16
17      // getters e setters
18  }
19
20  public class Cliente
21  {
22      private string nome;
23      private Data dataNascimento;
24
25      // getters e setters
26  }
```

O importante desta discussão é apenas perceber que é possível usar seus tipos como variáveis dentro de outras classes e fazer estruturas muito complexas.

5 Garbage Collector

Para finalizar estes tópicos avançados sobre memória e ponteiros, é importante mencionar o Garbage Collector, Coletor de Lixo ou simplesmente GC. O GC é basicamente um subsistema do Java que tem a missão de limpar dados não utilizados ao longo do tempo.
Em muitas linguagens você faz o gerenciamento de memória. Isso significa que se você criar dados dinâmicos, caso você esqueça de apagá-los, eles ficarão para sempre na sua memória até que a aplicação seja finalizada. Isso é o que chamamos de Memory Leak e que faz com que algumas aplicações fiquem muito mais pesadas do que deveriam após intenso uso.
O GC automatiza isso para você. Todas as informações gerenciadas, ou seja, que ficam no Heap são então gerenciadas pelo GC e quando não existem mais ponteiros para esses dados, eles são automaticamente limpos.
É claro que isso gera alguns efeitos colaterais. Abusar do Heap pode deixar sua aplicação muito mais lenta e criar gargalos por muitos motivos. Um deles é a desfragmentação, que é o que ocorre quando o GC move os dados para tirar buracos que aparecem quando dados são limpos no meio da memória. É um processo pesado, mas necessário para otimizar seu uso.
Isso nos torna ainda mais fãs dos dados não gerenciados, que podemos usar utilizando tipos por valor em funções que ficarão na Stack, que não é gerenciada pelo GC.

6 Exemplo 1: Fazendo uma Lista Encadeada com Ponteiros para armazenar uma quantidade variável de clientes

```
1  // Main.java
2  public class Main
3  {
4      public static void main(String[] args)
5      {
6          LinkedList clientes = new LinkedList();
7
8          Cliente cliente1 = new Cliente();
9          cliente1.setNome("Gilmar");
10         clientes.add(cliente1);
11
12         Cliente cliente2 = new Cliente();
13         cliente2.setNome("Pamella");
14         clientes.add(cliente2);
15
16         // Se Get retornar null, Nome deve retornar null ao invés de estourar um erro, se Nome retornar null deve-se
   substituir pelo valor padrão
17         // 'Cliente não encontrado'
18         Cliente result = clientes.get(1);
19         System.out.println(
20             result == null ?
21             "Cliente não encontrado" :
22             result.getNome()
23         );
24     }
25 }
26
27 // Cliente.java
28 class Cliente
29 {
30     private String nome;
31     public String getNome() {
32         return nome;
33     }
```

```
34      public void setNome(String nome) {
35          this.nome = nome;
36      }
37
38      private Long cpf;
39      public Long getCpf() {
40          return cpf;
41      }
42      public void setCpf(Long cpf) {
43          this.cpf = cpf;
44      }
45  }
46
47  // LinkedListNode.java
48  class LinkedListNode
49  {
50      private Cliente value = null;
51      public Cliente getValue() {
52          return value;
53      }
54      public void setValue(Cliente value) {
55          this.value = value;
56      }
57
58      private LinkedListNode next = null;
59      public LinkedListNode getNext() {
60          return next;
61      }
62      public void setNext(LinkedListNode next) {
63          this.next = next;
64      }
65  }
66
67  //LinkedList.java
68  class LinkedList
69  {
70          // Ponteiro vazio = lista vazia
71          private LinkedListNode first = null;
72
```

```
73          private int size = 0;
74          int size() {
75              return size;
76          }
77
78          // Função para adicionar um valor no final da lista
79          void add(Cliente value)
80          {
81              size++;
82
83              // Se first for nulo, vamos inicializá-lo com um novo elemento
84              if (first == null)
85              {
86                  first = new LinkedListNode();
87                  first.setValue(value);
88                  return;
89              }
90
91              // Caso first != null precisamos buscar o último elemento da lista para então
92              // preenche-lo
93
94              // Nó atual
95              LinkedListNode crr = first;
96              // Enquanto existir um próximo, avance para ele
97              while (crr.getNext() != null)
98                  crr = crr.getNext();
99
100             // Aqui crr.Next é nulo
101             LinkedListNode next = new LinkedListNode();
102             next.setValue(value);
103             crr.setNext(next);
104         }
105
106         // Função para ler em uma posição específica do vetor, retornal null se o indice for inválido (fora da lista)
107         Cliente get(int index)
108         {
109             if (index < 0)
110                 return null;
111
```

```
112             // Busca até atingir o índice ou ter crr nulo
113             LinkedListNode crr = first;
114             for (int i = 0; i < index && crr != null; i++)
115                 crr = crr.getNext();
116 
117             // Se crr for nulo, retorna nulo, caso contrário retorna seu Value
118             return crr == null ? null : crr.getValue();
119         }
120 }
```