Aula 02: 21/02/2019

Tópicos

- análise amortizada
- redimensionamento
- comentários sobre a provinha 1

Pilhas

Resumo:

```
- pilha e sua API
- implementações em vetor
- pilhas genéricas
- redimensionamento
```

Pré-requisitos:

```
- vetores
- ADT, API, cliente, interface, implementação
```

API

`public class`	`Stack<Item>`	
	`Stack()`	construtor: cria uma pilha de `Items` vazia
`void`	`push(Item item)`	insere item nesta pilha
`Item`	`pop()`	remove o item mais recente desta pilha
`boolean`	`isEmpty()`	esta pilha está vazia?
`int`	`size()`	número de itens nesta pilha

Pilhas genéricas em vetores

`Item` é uma tipo genérico, ou parâmetro de tipo, que deve ser substituído por um tipo concreto quando uma instância da pilha é criada.

```
public class Stack<Item> {

    private Item[] a = null;
    private int n = 0;

    public Stack(int cap) {
        a = (Item[]) new Object[cap];
    }

    public boolean isEmpty() {
        return n == 0;
    }
```

```
    public int size() {
        return n;
    }

    public void push(Item item) {
        a[n++] = item;
    }

    public Item pop() {
        return a[--n];
    }
}

public class ClientStackString {

   public static void main(String[] args) {
      Stack<String> pilha;
      pilha = new Stack<String>(20);

      while (!StdIn.isEmpty()) {
         String str = StdIn.readString();
         if (!str.equals("-"))
            pilha.push(str);
         else if (!pilha.isEmpty())
                 StdOut.println(pilha.pop() + " ");
      }
      StdOut.println("(" + pilha.size() + " left on stack)");
   }
}
```

Pilhas com redimensionamento

Classe `Stack` é uma pilha implementada em vetor com redimensionamento (*resizing array*).

O método privado `resize()` faz um redimensionamento: aumenta ou diminui o vetor que abriga a pilha.

Depois do redimensionamento, não é necessário liberar o espaço ocupado pelo antigo vetor pois o mecanismo de coleta de lixo do Java cuida disso automaticamente.

Graças à maneira como o redimensionamento é invocado por `pop()` e `push()`, o vetor `a[]` sempre está pelo menos 25% cheio.

```
public class Stack<Item> {
    private Item[] a = null;
    private int n = 0;

    public Stack() {
        a = (Item[]) new Object[1];
        n = 0;
    }
```

```
    public boolean isEmpty() {
        return n == 0;
    }

    public int size() {
        return n;
    }

    public void push(Item item) {
        if (n == a.length) resize(2*a.length);
        a[n++] = item;
    }

    public Item pop() {
        Item item = a[--n];
        a[n] = null; // Avoid loitering
        if (n > 0 && n == a.length/4) resize(a.length/2);
        return item;
    }

    private void resize(int max) {
        Item[] tmp = (Item[]) new Object[max];
        for (int i = 0; i < n; i++) {
            tmp[i] = a[i];
        }
        a = tmp;
    }
}
```

O **custo amortizado** de uma operação é o custo médio da operação quando considerada em uma sequência de operações do ADT. Não se trata, portanto, da média de um conjunto aleatório de execuções da operação.

Loitering object: o coletor de lixo do `Java` retoma a memória associada com objetos que não podem ser acessados.

Orfão o item em pop() é orfão, mas o coletor de lixo não tem como saber.

Loitering object objeto ocioso

Atribuir `null` para avisar o coletor de lixo que a memória não é mais necessária.

Pilhas redimensionáveis e iteráveis

```
import java.util.Iterator; // passo 0

public class Stack<Item> implements Iterable<Item> { // passo 1
    private Item[] a = null;
    private int n = 0;

    public Stack() {
```

```
        a = (Item[]) new Object[1];
        n = 0;
    }

    public boolean isEmpty() {
        return n == 0;
    }

    public int size() {
        return n;
    }

    public void push(Item item) {
        if (n == a.length) resize(2*a.length);
        a[n++] = item;
    }

    public Item pop() {
        Item item = a[--n];
        a[n] = null; // Avoid loitering
        if (n > 0 && n == a.length/4) resize(a.length/2);
        return item;
    }

    private void resize(int max) {
        Item[] tmp = (Item[]) new Object[max];
        for (int i = 0; i < n; i++) {
            tmp[i] = a[i];
        }
        a = tmp;
    }

    // passo 2
    public Iterator<Item> iterator() {
        return new ReverseArrayIterator();
    }

    // passa 3
    private class ReverseArrayIterator implements Iterator<Item> {
        private int t = n;

        public boolean hasNext() {
            return t > 0;
        }

        public Item next() {
            return a[--t];
        }

        public void remove() {
```

```
            throw new UnsupportedOperationException();
        }
    }

    public static void main(String[] args) {
        Stack<String> s = new Stack<String>();

        while (!StdIn.isEmpty()) {
            String item = StdIn.readString();
            if (!item.equals("-")) s.push(item);
            else if (!s.isEmpty()) StdOut.println(s.pop() + " ");
        }

        StdOut.println("\nConteúdo usando while (...): ");
        StdOut.println("----");
        Iterator<String> it = s.iterator();
        while (it.hasNext()) {
            StdOut.println(it.next());
        }
        StdOut.println("----\n");

        StdOut.println("\nConteúdo usando foreach statement: ");
        StdOut.println("----");
        for (String bla: s) {
            StdOut.println(bla);
        }
        StdOut.println("----\n");

        StdOut.println("(" + s.size() + " left on stack)");
    }
}
```

Custo amortizado

O **custo amortizado** de uma operação é o custo médio da operação quando considerada em uma sequência de operações do ADT. não se trata, portanto, da média de um conjunto aleatório de execuções da operação.

Consideremos uma sequência de m operações `push()`.

Sejam n_i e t_i o número de itens e o tamanho da pilha depois da operação i.

O custo real da i-ésima operação `push()` é 1 se há espaço e n_i se a pilha está cheia.

Custo de uma operação é $O(m)$. Portanto, o custo das m operações $O(m^2)$. Isso está correto, mas é um exagero.

Exemplo

operação (n)	t	custo

operação (n)	t	custo
1	1	1
2	2	$1 + 1$
3	4	$1 + 2$
4	4	1
5	8	$1 + 4$
6	8	1
7	8	1
8	8	1
9	16	$1 + 8$
10	16	1
...	16	1
17	32	$1 + 16$
33	64	$1 + 32$

Custo total:

$\sum_{i=1}^{m} c_i = m + \sum_{i=0}^{k} 2^i = m + 2^{k+1} - 1 < m + 2m - 1 < 3m$, onde $k = \lfloor \lg(m-1) \rfloor$.

Análise de algoritmos

Leitura: Analysis of Algoriths, S&W

Escrevemos $\sim f(n)$ para representar qualquer função que, quando dividida por $f(n)$ vai para 1 quano $n \to \infty$.

$g(n) \sim f(n)$ indica que $g(n)/f(n) \to 1$ quando n cresce.

Note a diferença com a notação O().

$f(n)$ é $O(g(n))$ se existem constantes c e n_0 tais que $f(n) \leq g(n)$ para $n > n_0$.

Memória

Memória consumida por objetos primitivos

```
tipo      bytes
---------------
boolean     1
byte        1
char        2
int         4
float       4
long        8
double      8

int[] a = new int[n];
--------------------
```

```
    overhead = 16 bytes
    valor n = 4 bytes
    total 24 + 4n

    double[] c = new double[n];
    ---------------------------
    overhead = 16 bytes
    valor n = 4 bytes
    total 24 + 8n

    double[][] t = new double[n][m];
    -------------------------------
    valor   m = 4 bytes
    m referências = 8m
    valores n = 4m bytes
    valores nm doubles = 8nm
    Total = 24 + 8m + m*(24+8n) = 24 + 32m + 8mn

    Resumo:

    tipo       bytes
    -------------------
    int[]      ~4n
    double[]   ~8n
    double[][] ~8nm
```

Apêndice

API:

```
    public class Counter
    --------------------
           Counter(String id)   cria um contador de nome id
      void increment()          incrementa o contador
      int  tally()              numero de incrementos
    String toString()           representação de um contador como string
```

```
// https://www.ime.usp.br/~pf/sedgewick-wayne/stdlib/documentation/index.html
import edu.princeton.cs.algs4.StdOut;

public class Counter {
    private final String name; // variável de instância
    private int count;         // variável de instância
    private static int noCounters = 0; // variável de classe

    // construtor
    public Counter(String id) {
        name = id;
        count = 0; // supérfluo
```

```
        noCounters++;
    }

    public void increment() {
        count++;
    }

    public int tally() {
        return count;
    }

    // retorna um string usado por print...
    public String toString() {
        return count + " " + name;
    }

    // retorna o número de contadores
    public static int size() {
        return noCounters;
    }

    public static void main(String[] args) {
        Counter pares = new Counter("pares");
        Counter impares = new Counter("ímpares");

        pares.increment();
        pares.increment();
        impares.increment();

        StdOut.println(pares + " " + impares);
        StdOut.println(pares.tally() + impares.tally());
        StdOut.println("número de contadores: " + Counter.size());
    }
}
```

`public`: métodos e variáveis acessíveis a todos os clientes (API).

`private`: métodos e variáveis auxiliares, clientes não têm acesso.

`static`: métodos e variáveis são da classe. demais são de instância (cada instância tem sua cópia).

Objetos

- objetos: referências
- apelidos versus clones
- classes nativas *versus* classes definidas pelo usuário
- classes nativas: `Integer`, `Float`, `Boolean`, `String`
- atributos de estado e métodos
- método especial/mágico construtor `public NomeDaMinhaClasse()`
- método especial/mágico `toString()`

Classes

Classes são formadas por atributos que podem ser variáveis ou funções que são chamadas de métodos.

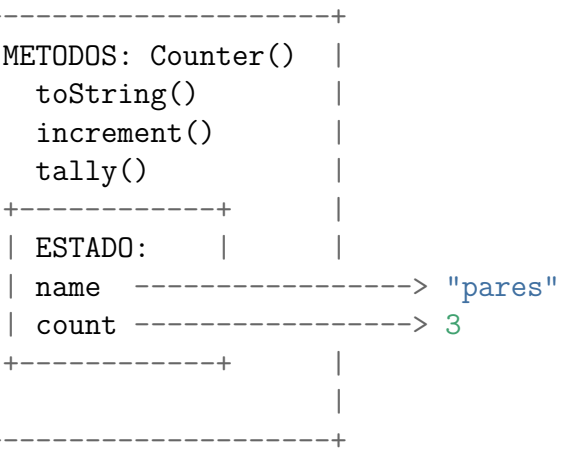
A primeira letra em um nome de uma classe deve ser maiúscula e usamos camelCase.

Objetos

Um objeto contém as informações/valores de um tipo definido pelo programador.

Esquema geral da classe Counter

```
Counter pares = new Counter("pares")
```



Métodos

Métodos são funções associadas com uma determinada classe.

```
pares.increment();
```

Métodos são como funções, mas há duas diferenças:

- métodos são definidos dentro de uma classe
- a sintaxe para executar um método é diferente
- função imprima, aqui está um objeto para você imprimir
- `r1.imprima()` sugere `r1`, imprima a si mesmo
- essa mudança de perspectiva pode ser polida, mas não é óbvio que seja útil.

Algumas vezes mover a responsabilidade de uma função para um objeto faz com que seja possível escrever um código mais versátil que é mais fácil de ser reutilizado e mantido.

No momento o que está escrito acima é longe de óbvio...

Construtores

O método especial `public NomeDaClasse()` é responsável por construir e retornar um objeto.

Chamado quando um objeto é criado (= *instanciado* é um nome mais bonito) através da palavra `new`.

Imprimindo um objeto

O método especial `toString` cria e retorna um string que diz como o objeto deve ser impresso por printxx().

Cliente de Counter

Com leitura de arquivo e entrada padrão.

```
import edu.princeton.cs.algs4.In;
import edu.princeton.cs.algs4.StdOut;

public class ParesImpares {
    public static void main(String[] args) {
        In in = new In(args[0]);
        Counter pares = new Counter("pares");
        Counter impares = new Counter("impares");

        while (!in.isEmpty()) {
            int valor = in.readInt();
            StdOut.printf("%d\n", valor);
            if (valor % 2 == 0) pares.increment();
            else impares.increment();
        }

        StdOut.println("Relatório: ");
        StdOut.println(pares);
        StdOut.println(impares);
    }
}
```

Java

Tecnologia Java

Links para fontes: Java virtual machine; OpenJDK; e The top 11 Free IDE for Java Coding, Development & Programming.

Java virtual machine (JVM)

É um computador abstrato que permite a um computador executar um programa em Java. Existem três noções associadas a uma JVM: *especificação*, *implementação* e *instância*.

A **especificação** é um documento que define o comportamento de uma JVM.

Uma **implementação** e um programa que executa os programas Java.

Uma **instância** de um JVM é um implementação durante a execução de um programa em Java bytecode.

A JVM da Oracle tem o nome *HotSpot*.

Java Runtime Environment (JRE)

É um pacote que contém tudo que é necessário para executar um programa Java. Um JRE inclui a implementação de uma JVM junto com uma implementação de uma *Java Class Library*.

Java Development Kit (JDK)

É um superconjunto de uma JRE que contém ferreamentas para um programador Java, como o compilador java `javac`. JDK são distribuidas gratuitamente pela *Oracle Corporation* ou pelo *OpenJDK source project*.

DrJava

DrJava é uma IDE extremamente leve que é usada para escrever programas em Java. Foi projetada especialmente para estudantes e possui uma interface intuitiva e a habilidade de executar código Java interativamente.

A sua característica principal é ser usada como uma *unit testing tool*, um depurador em código fonte, um painel interativo para executar um programa. Possui um editor "inteligente" e pode ser usado para mais coisas dependendo das suas necessidades.

Está disponível gratuitamente sob a licensa BSD e está sendo desenvolvido ativamente por *JavaPLT group* da universidade de Rice.

Classpath

Fonte: Mastering the Java CLASSPATH

Uma das regras mais fundamentais de organização de um código Java é *package name = directory name*. Começamos definindo um diretório que é idêntico ao nome do package. No caso de um pacote `com.web_tomorrow` teríamos:

```
[root]
  com
    web_tomorrow
       CPTest1.java
       CPTest2.java
```

Java Packages

Fontes: Java package; What Is a Package?; e Java™ Platform, Standard Edition 8 API Specification.

Um *package Java* organiza classes Java em *namespaces*, fornecendo um único namespace para cada tipo. Classes em um mesmo package podem acessar outros acessar outros membros privados e protegidos de outras

classes. Java packages podem ser armazenado em arquvos comprimidos chamdos **JAR files**, permitindo que classes possam ser baixadas rapidamento como grupos em vez de individualmente.

Em geral, um package pode conter: classes, interfaces, enumerations, e annotation types. Um package permite que um desenvolvedor agrupar classes (e interfaces). Essas classes são relacionadas de alguma forma – todas podem estar ligadas a alguma aplicação ou executar uma tarefa específica.

Programadores tipicamente usam packages para organizar classes pertencentes a uma mesma categoria ou que tem funcionalidades semelhantes.