Erros Frequentes nos Trabalhos de ADA

Estrutura do Programa

- Constantes e Identificadores
- Tipos e Interfaces
- Entrada, Saída e Cálculo
- Encapsulamento
- *Enhanced for-loop*

Constantes e Identificadores

Definam constantes

Escolham nomes apropriados para as variáveis, parâmetros, métodos, etc.

Tipos e Interfaces (1)

Usem as interfaces para declarar os tipos dos objetos

Exemplo: Declarar e criar um vetor de listas ligadas de inteiros, com comprimento *vecLen* (= 4), e criar as *vecLen* listas.

- **Declaração da variável:** um vetor de listas de inteiros.
  - LinkedList<Integer>[] vec; **ERRO**
  - List<Integer>[] vec; **CORRETO**

- **Criação do vetor:**
  - vec = **new** LinkedList<>[vecLen]; **ERRADO**
  - vec = **new** List<>[vecLen]; **QUASE CORRETO**

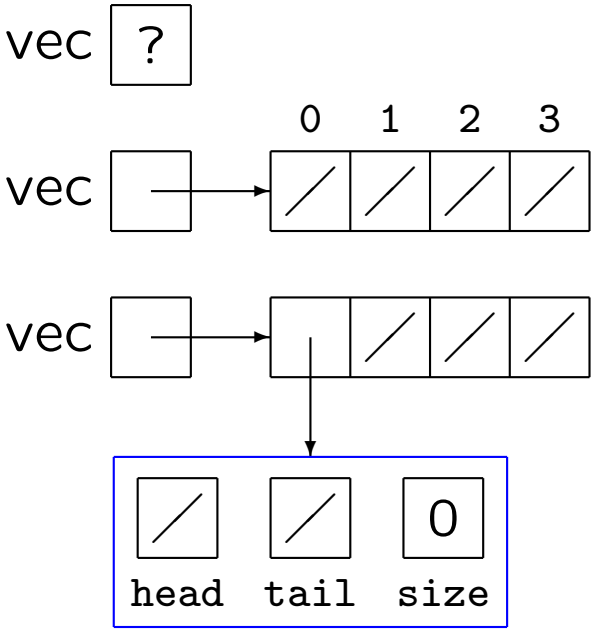
Tipos e Interfaces (2)

```
List<Integer>[] vec;
```

```
vec = new List<>[vecLen];
```

ERRADO

```
vec[0] = new LinkedList<>();
```




```
@SuppressWarnings( "unchecked" )
void createDataStructure( int vecLength ) {
    vec = new List[vecLength];
    for ( int i = 0;  i < vec.length;  i++ )
        vec[i] = new LinkedList<>();
}
```

CORRETO

Entrada, Saída e Cálculo (1)

A funcionalidade deve ser independente da sintaxe de entrada dos dados e da sintaxe de saída dos resultados

Exemplo: A entrada e a saída são compostas por dois inteiros, separados por um espaço.

```
public static void main( String[] args ) throws IOException {
    BufferedReader input =
        new BufferedReader( new InputStreamReader(System.in) );
    String line = input.readLine();
    Problem prob = new Problem(line);
    String ans = prob.answer();
    System.out.println(ans);
}
```

2 ERROS

Entrada, Saída e Cálculo (2)

A funcionalidade deve ser independente da sintaxe de entrada dos dados e da sintaxe de saída dos resultados

```
public static void main( String[] args ) throws IOException {
    BufferedReader input =
        new BufferedReader( new InputStreamReader(System.in) );
    String[] tokens = input.readLine().split(" ");
    int n1 = Integer.parseInt(tokens[0]);
    int n2 = Integer.parseInt(tokens[1]);
    Problem prob = new Problem(n1, n2);
    int[] ans = prob.answer();
    System.out.println(ans[0] + " " + ans[1]);
}
```

CORRETO

Encapsulamento

Os métodos não devem retornar membros de dados alteráveis do exterior

```
public class Graph {
    private List<Integer>[] edges;

    public List<Integer> adjacentNodes( int node ) {
        return edges[node];
    }

}
```

ERRO

A correção deste erro tem de ser encontrada caso a caso.

Enhanced for-loop

Prefiram-no para iterar

```
Iterator<ElemType> iter = struct.iterator();
while ( iter.hasNext() ) {
    ElemType elem = iter.next();
    // PROCESS elem
}
```

MUITO LIXO

```
for ( ElemType elem : struct ) {
    // PROCESS elem
}
```

MAIS LEGÍVEL

Eficiência

- Criação de Objetos
  - Objetos Desnecessários
  - EDs Pseudodinâmicas
  - Acesso aos Caracteres das Strings
- Repetição de Cálculos
  - Repetição Visível
  - *containsKey* + *get* em Dicionários
  - *get* em Listas Ligadas

Criação de Objectos Desnecessários

Não criem objetos que nunca serão usados

```
int capacity = 5000;
```

```
Map<String, List<Integer>> map = new HashMap<>(capacity);
map = this.createMap(capacity);
```

ERRO

```
Map<String, List<Integer>> map;
map = this.createMap(capacity);
```

CORRETO

Criação de EDs Pseudodinâmicas

Não criem EDs **estáticas** com a capacidade por omissão

```
int capacity = 5000;        // Maximum size of the list.
```

```
List<T> list = new ArrayList<>();
```

ERRO

Inicialmente o vetor tem capacidade 10,

ao inserir o 11º elem., copia-se a informação para um vetor com capacidade 20,

ao inserir o 21º elem., copia-se a informação para um vetor com capacidade 40,

e assim sucessivamente, para as capacidades 80, 160, 320, 640, 1280, 2560 e 5120.

```
List<T> list = new ArrayList<>(capacity);
```

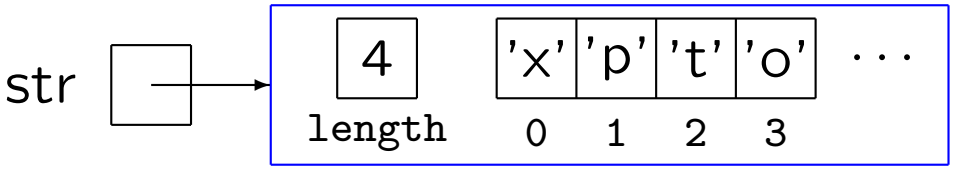
CORRETO

Acesso aos Caracteres das Strings

Usem o método *charAt*

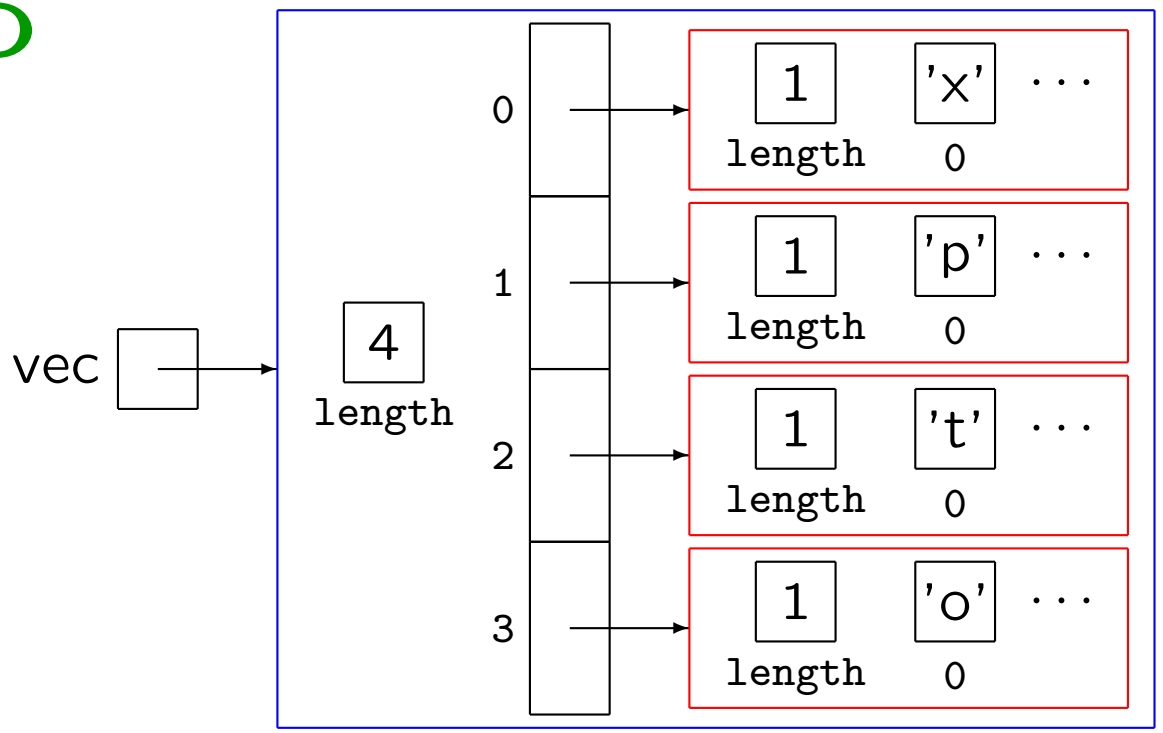
```
String str = "xpto";
char firstC = str.charAt(0);
if ( firstC == 'y' )
    ...
```

CORRETO




```
String[] vec = str.split("");
if ( vec[0].equals("y") )
    ...
```

ERRO



Repetição de Cálculos Visível

Não repitam cálculos para poupar variáveis

```
String[] vec = str.split(" ");   // The first element is an integer.
int res;
```

```
if ( Integer.parseInt(vec[0]) >= 0 )
    res = Integer.parseInt(vec[0]) + 1;
else
    res = Integer.parseInt(vec[0]) / 2;
```

ERRO

```
int num = Integer.parseInt(vec[0]);
if ( num >= 0 )
    res = num + 1;
else
    res = num / 2;
```

CORRETO

containsKey + *get* em Dicionários

se chave não existe, insere entrada; obtém sempre valor

```
Map<String,Integer> map = new ··· ;
```

```
int value;
if ( !map.containsKey(key) )
    value = this.computeValue(key);
    map.put(key, value);
else
    value = map.get(key);
```

ERRO

2 pesquisas, se chave existe

```
Integer value = map.get(key);
if ( value == null )
    value = this.computeValue(key);
    map.put(key, value);
```

CORRETO

1 pesquisa, se chave existe

get em Listas Ligadas

Percorram uma lista ligada iterando-a

```
List<Integer> list = new LinkedList<>();   // n = list.size()

for ( int i = 0;  i < list.size();  i++ ) {
    int elem = list.get(i);
    // PROCESS elem
}
```

ERRO

Complexidade $\Theta(n^2)$ †

```
for ( int elem : list ) {
    // PROCESS elem
}
```

CORRETO

Complexidade $\Theta(n)$ †

†Assume-se que o custo de processar cada elemento da lista é constante.