



# Programação Avançada

Introdução ao TAD –Graph  
Noções de Grafos  
Programação Avançada – 2020-21

Bruno Silva, Patrícia Macedo

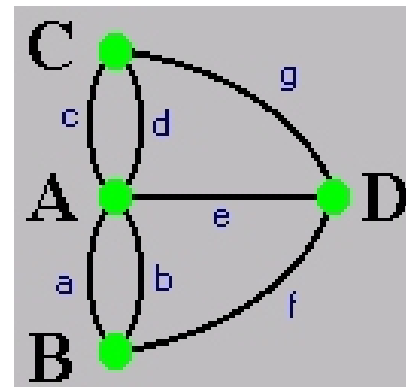
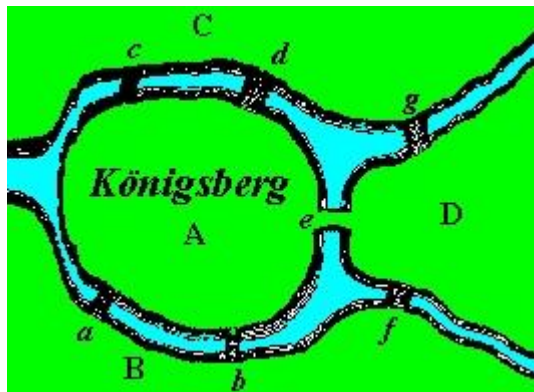
# Sumário



- Contexto Histórico
- Aplicações
- Definições
  - Grafo
  - Dígrafo
  - Ordem, adjacência e grau
  - Tipos de grafos
- TAD Graph
  - Estruturas de Dados para implementar os grafos

# Contexto Histórico

- Problema das Pontes de Königsberg
  - No século 18, na cidade de Königsberg (antiga Prússia), um conjunto de sete pontes cruzavam o rio Pregel. Elas conectavam duas ilhas entre si (A e D) e as ilhas com as margens (C e B).
  - Os habitantes perguntavam: É possível cruzar as sete pontes numa caminhada contínua sem passar duas vezes por qualquer uma delas?
  - Em 1736, Euler apresenta a solução deste problema na Academia de São Petersburgo, sendo este o primeiro trabalho sobre **Grafos**.



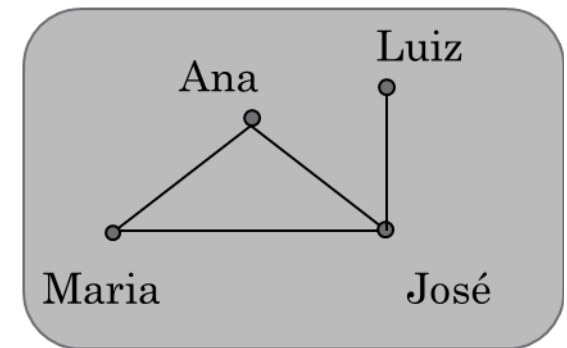
# Definição formal de Grafo

- **Grafo (Graph)**

- Um grafo  $G(V, A)$  é definido pelos conjuntos  $V$  e  $A$ , onde:
  - $V$  é um conjunto não vazio: Vértices, Nodos ou Nós do grafo (**vertex**)
  - $A$  é um conjunto de pares ordenados  $a=(v,w)$  com  $v$  e  $w$  pertencente a  $V$ : Arestas, Linhas ou Ramos do grafo (**edge**).

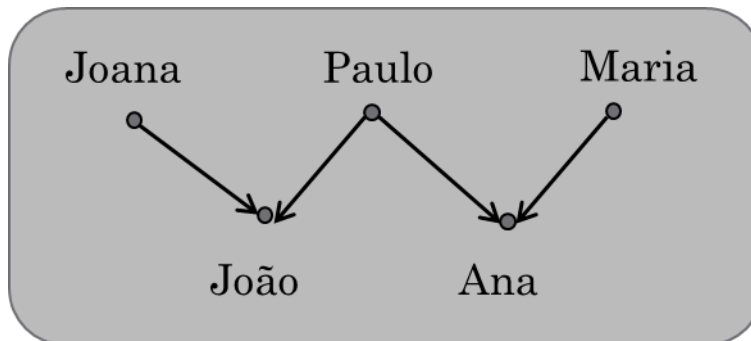
- **Exemplo**

- $V = \{p \mid p \text{ é uma pessoa}\}$
- $A = \{(v,w) \mid v \text{ é amiga de } w\}$ 
  - $V = \{\text{Maria, José, Ana, Luiz}\}$
  - $A = \{(\text{Maria, José}), (\text{Maria, Ana}), (\text{José, Luiz}), (\text{José, Ana})\}$



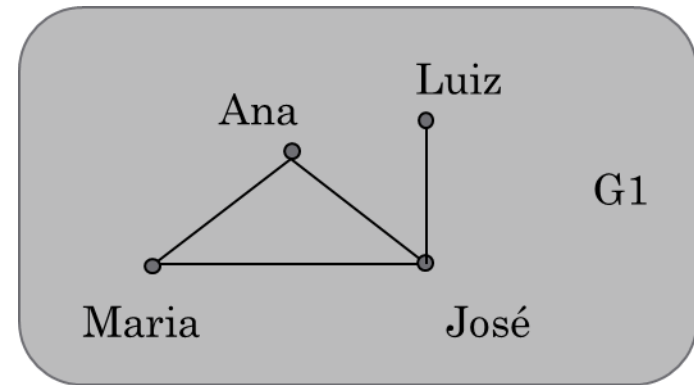
# Definição de Dígrafo (grafo orientado)

- Dígrafo
  - Grafo orientado
- Exemplo
  - $V = \{p \mid p \text{ é uma pessoa}\}$
  - $A = \{(v,w) \mid v \text{ é pai ou mãe de } w\}$



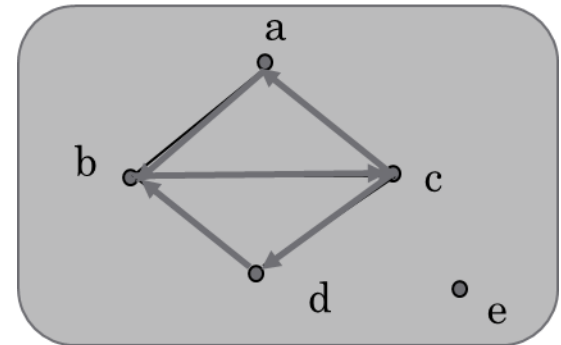
# Ordem e Adjacência

- Ordem
  - É o número de vértices do grafo
    - $\text{Ordem}(G1) = 4$
- Adjacência
  - Dois vértices -  $v$  e  $w$  de um grafo - são adjacentes se há uma aresta  $a=(v,w)$  em  $G$ 
    - Ex: José e Luiz em  $G1$
  - Duas arestas são adjacentes se incidem sobre o mesmo vértice
    - Ex:  $(\text{Ana}, \text{Maria})$  e  $(\text{Ana}, \text{José})$  em  $G1$



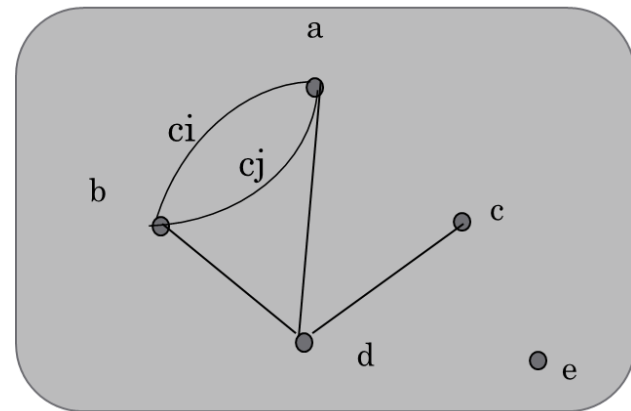
# Graus

- Grau de um vértice
  - É o número de arestas incidentes no vértice
    - $\text{Grau}(b)=3$
- Grau de saída (*outdegree*)
  - Número de arestas que têm ponta inicial no vértice
    - $\text{GrauSaída}(b) = 1$
- Grau de entrada (*indegree*)
  - Número de arestas têm ponta final no vértice
    - $\text{GrauEntrada}(b) = 2$



# Vértice Isolado, Arestas Paralelas

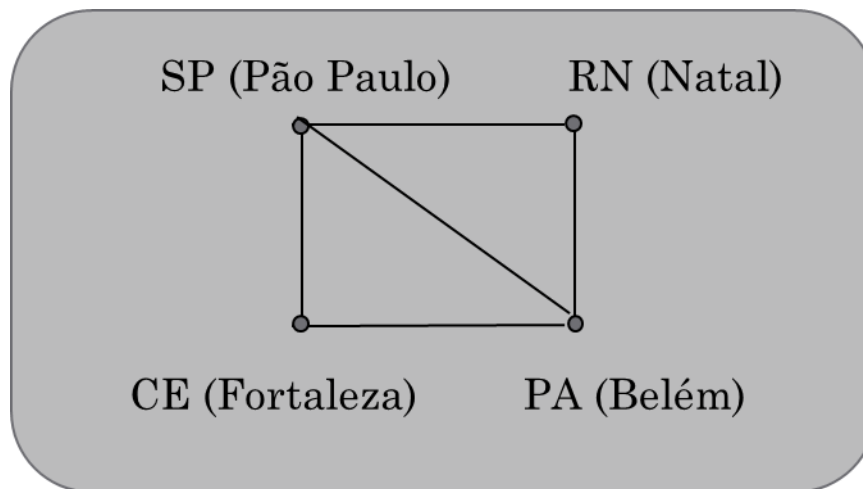
- Vértice isolado
  - É aquele que possui grau igual a zero
    - Ex: Vértice e
- Arestas paralelas
  - Possuem os mesmos vértices terminais
    - Exemplo  $ci=(a,b)$  e  $cj=(a,b)$





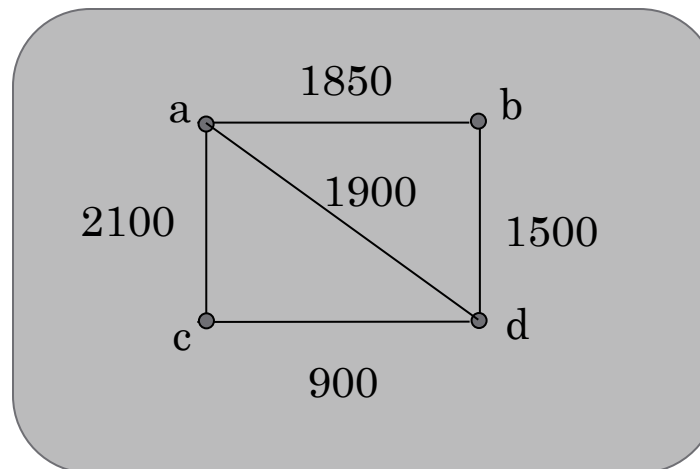
# Grafo Rotulado

- Grafo rotulado
  - Grafo em que cada vértice está associado a um rótulo



# Grafo Valorado

- Grafo valorado
  - Um grafo  $G(V, A)$  é denominado **valorado** quando cada aresta tem associado um valor.



# Algumas áreas de aplicação dos Grafos

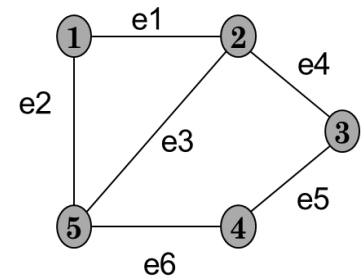
GRAPH	VERTICES	EDGES
circulatory	organ	blood vessel
skeletal	joint	bone
nervous	neuron	synapse
social	person	relationship
epidemiological	person	infection
chemical	molecule	bond
n-body	particle	force
genetic	gene	mutation
biochemical	protein	interaction
transportation	intersection	road
Internet	computer	cable
Web	web page	link
distribution	power station	power line
mechanical	joint	beam
software	module	call
financial	account	transaction

# TAD Graph| Exercícios(1)



Para o grafo da imagem indica quais as frases verdadeiras

- A ordem do grafo é 3
- Os vértice com maior grau são os vértices 5 e 2
- As arestas incidentes ao vertice2 são as arestas e1 e e4
- O vértice oposto ao vértice 2 na aresta e3 é o vértice 5
- O vértice 1 e 5 são adjacentes



Desenha um grafo que obedeça aos seguintes critérios

- É um digrafo
- Tem ordem 4
- Tem um vértice isolado
- Tem duas arestas paralelas.
- O vértice com maior grau de saída tem grau 4
- O vértice com maior grau de entrada tem grau 2

# TAD Graph - Especificação

Um grafo  $G(V, E)$  é definido pelos conjuntos  $V$  e  $E$ , onde:

- $V$  é um conjunto não vazio: Vértices.
- $E$  é um conjunto de pares ordenados  $e=(v,w)$  com  $v$  e  $w$  pertencente a  $V$ : arestas (edges).

As operações modificadoras do TAD Graph são:

- **insertVertex(v)**: insere  $v$  como sendo vértice do grafo.
- **insertEdge(u, v, e)**: insere uma aresta  $x$  entre os vértices  $u$  e  $v$ ; devolve erro se  $v$  e  $x$  não correspondem a vértices do grafo
- **removeVertex(v)**: remove o vértice  $v$  e todas as suas arestas adjacentas, devolve erro se  $v$  não existir no grafo.
- **removeEdge(e)**: remove a aresta  $e$ , devolve erro se  $e$  não existir no grafo.

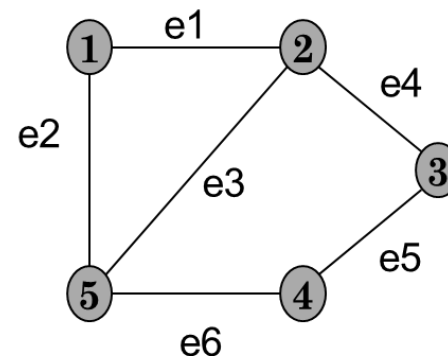
## TAD Graph – Especificação (cont)

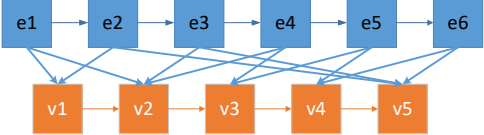
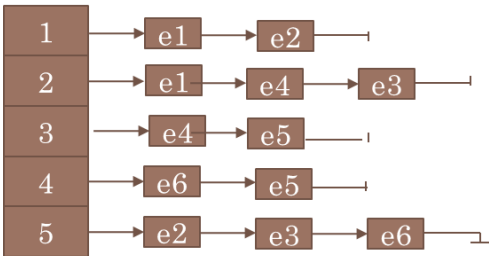
As operações inspetoras (devolvem informação sobre o estado do grafo):

- **numvertices()**: devolve o número de vértices
- **numEdges()**: devolve o número de arestas
- **edges()** : devolve uma coleção iterável das arestas do grafo.
- **vertices()** : devolve uma coleção iterável dos vértices do grafo.
- **opposite(v, e)**: devolve o vértice da aresta e oposto ao vértice v, devolve erro se v ou e não existem no grafo, ou se v não é vértice da aresta e.
- **degree(v)**: devolve o grau do vértice v, devolve erro se v não existe no grafo.
- **incidentEdges(v)**: devolve uma coleção iterável das arestas incidentes ao vértice v, devolve erro se v não existe no grafo.
- **areAdjacent(v,w)**: devolve um valor lógico (true/false) que indica se os vértices v e w são adjacentes , devolve erro se v ou w não existirem como vértices do grafo.

# Estruturas de Dados para implementar o TAD Graph

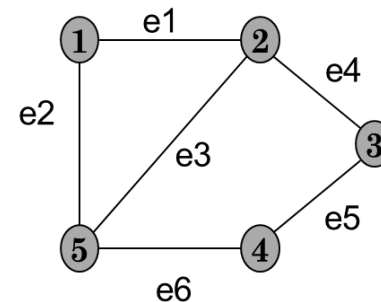
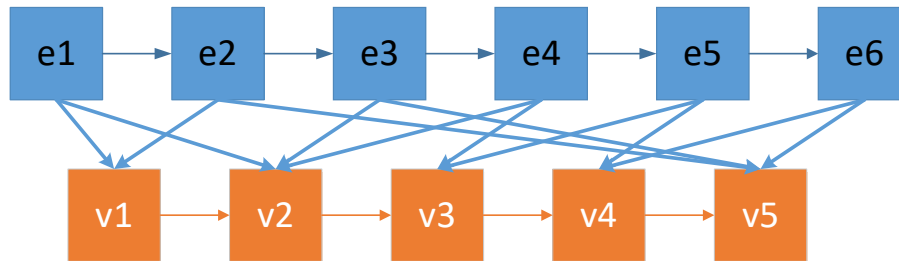
- Existem várias estruturas de dados possíveis para implementar o grafo TAD.
- A seleção da estrutura de dados influencia diretamente a complexidade algorítmica das operações.



Lista de Arestas	Lista de Adjacências	Matriz de Adjacências																																				
		<table><tr><th></th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th></tr><tr><th>1</th><td></td><td>e1</td><td></td><td></td><td>e2</td></tr><tr><th>2</th><td>e1</td><td></td><td>e4</td><td></td><td>e3</td></tr><tr><th>3</th><td></td><td>e4</td><td></td><td>e5</td><td></td></tr><tr><th>4</th><td></td><td></td><td>e5</td><td></td><td>e6</td></tr><tr><th>5</th><td>e2</td><td>e3</td><td></td><td>e6</td><td></td></tr></table>		1	2	3	4	5	1		e1			e2	2	e1		e4		e3	3		e4		e5		4			e5		e6	5	e2	e3		e6	
	1	2	3	4	5																																	
1		e1			e2																																	
2	e1		e4		e3																																	
3		e4		e5																																		
4			e5		e6																																	
5	e2	e3		e6																																		

# (1) Estrutura de dados - Listas de arestas

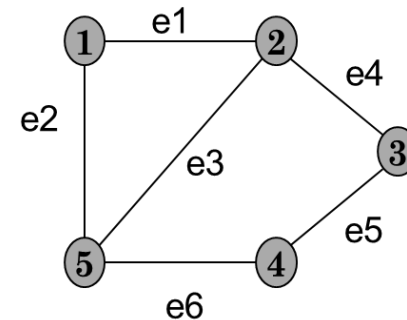
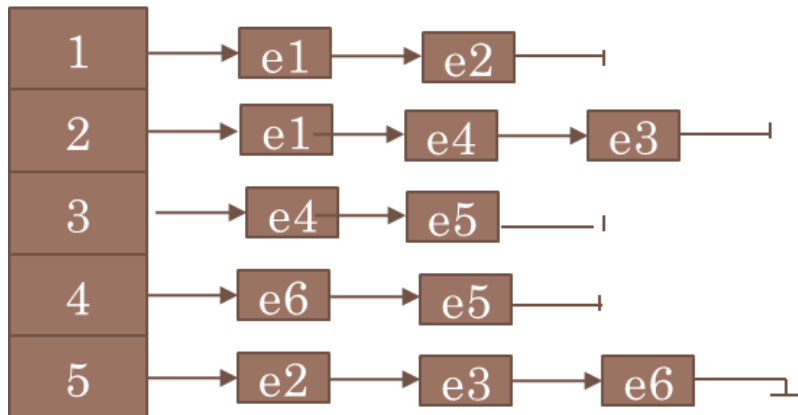
- Lista de arestas (ou dicionário-map).
- Lista de vértices (ou dicionário-map).
- Cada Aresta guarda as referências para os vértices que liga.





## (2) Estrutura de dados - Listas de adjacências

- Lista de vértices (ou dicionário-map).
- Cada vértice, guarda a lista de arestas adjacentes.
- A utilização de lista ligada facilita a “ampliação” da estrutura.

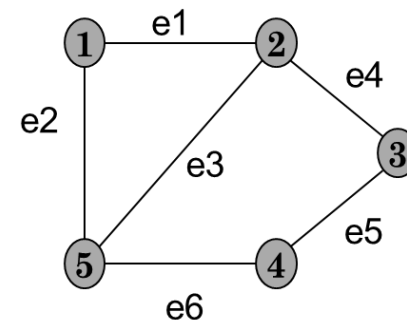


### (3) Estrutura de dados - Matriz de adjacências

- Lista de vértices.
- As adjacências são representadas por uma matriz  $N \times N$  onde  $N$  é o número de vértices. Espaço:  $O(n^2)$ .



	1	2	3	4	5
1		e1			e2
2	e1		e4		e3
3		e4		e5	
4			e5		e6
5	e2	e3		e6	

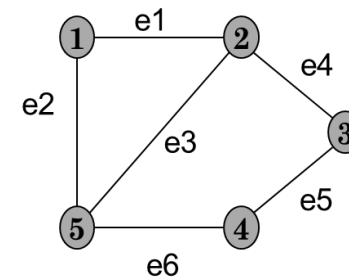


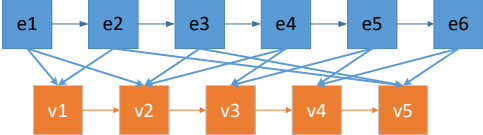
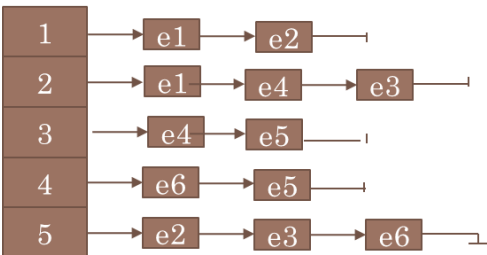
# TAD Graph| Exercícios(2)



Para cada uma das estruturas de dados apresentadas complete os esquemas abaixo de forma a ilustrar como ficaria cada uma delas após efetuar as seguintes operações no grafo dado.

- adicionar o vértice 6
- remover a aresta e3
- adicionar a aresta e7 que liga o vértice 6 com o vértice 4.

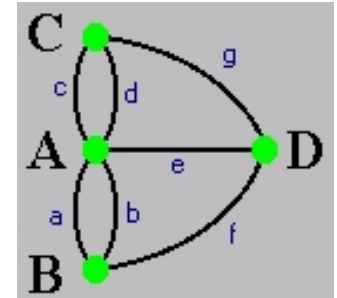


Lista de Arestas	Lista de Adjacências	Matriz de Adjacências																																				
		<table><tr><th></th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th></tr><tr><th>1</th><td></td><td>e1</td><td></td><td></td><td>e2</td></tr><tr><th>2</th><td>e1</td><td></td><td>e4</td><td></td><td>e3</td></tr><tr><th>3</th><td></td><td>e4</td><td></td><td>e5</td><td></td></tr><tr><th>4</th><td></td><td></td><td>e5</td><td></td><td>e6</td></tr><tr><th>5</th><td>e2</td><td>e3</td><td></td><td>e6</td><td></td></tr></table>		1	2	3	4	5	1		e1			e2	2	e1		e4		e3	3		e4		e5		4			e5		e6	5	e2	e3		e6	
	1	2	3	4	5																																	
1		e1			e2																																	
2	e1		e4		e3																																	
3		e4		e5																																		
4			e5		e6																																	
5	e2	e3		e6																																		

# TAD Graph| Exercícios(3)



- Para cada uma das estruturas de dados apresentadas:
  - (1) lista de arestas
  - (2) lista de adjacências
  - (3) matriz de adjacências



**instancie** o grafo que modela o problema das pontes de Königsberg.

- Conseguiu criar o grafo em todas as estruturas de dados ?



# Programação Avançada

Implementação do TAD –Graph  
Programação Avançada – 2020-21

Bruno Silva, Patrícia Macedo

# Sumário

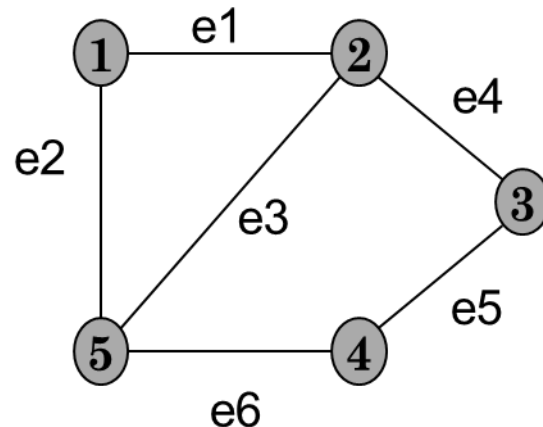


- Especificação do TAD Graph (revisão)
- Noção de Tipo de Dados Vertex e de Edge
- Interface Graph
- Implementação usando uma Lista de Arestas

# TAD Graph - Especificação

Um grafo  $G(V, E)$  é definido pelos conjuntos  $V$  e  $E$ , onde:

- $V$  é um conjunto não vazio: Vértices.  
 $\{v1, v2, v3, v4, v5\}$
- $E$  é um conjunto de pares ordenados  $e=(v,w)$  com  $v$  e  $w$  pertencente a  $V$ :  
arestas (edges).  
 $\{ e1=(v1,v2) , e2=(v1,v5), e3=(v2,v5), e4=(v2,v3), e5=(v4,v3), e6=(v5,v4) \}$



# TAD Graph – Especificação das operações

- **insertVertex(v):** insere v como sendo vértice do grafo.
- **insertEdge(u, v, e):** insere uma aresta **x** entre os vértices **u** e v; devolve erro se v e x não correspondem a vértices do grafo
- **removeVertex(v):** remove o vértice **v** e todas as suas arestas adjacentas, devolve erro se v não existir no grafo.
- **removeEdge(e):** remove a aresta **e** , devolve erro se e não existir no grafo.
- **numvertices():** devolve o número de vértices
- **numEdges():** devolve o número de arestas
- **edges() :** devolve uma coleção iterável das arestas do grafo.
- **vertices() :** devolve uma coleção iterável dos vértices do grafo.
- **opposite(v, e):** devolve o vértice da aresta e oposto ao vértice v, devolve erro se v ou e não existem no grafo, ou se v não é vértice da aresta e.
- **degree(v):** devolve o grau do vértice v, devolve erro se v não existe no grafo.
- **incidentEdges(v):** devolve uma coleção iterável das arestas incidentes ao vértice v, devolve erro se v não existe no grafo.
- **areAdjacent(v,w):** devolve um valor lógico (true/false) que indica se os vértices v e w são adjacentes , devolve erro se v ou w não existirem como vértices do grafo.



# Implementação em JAVA

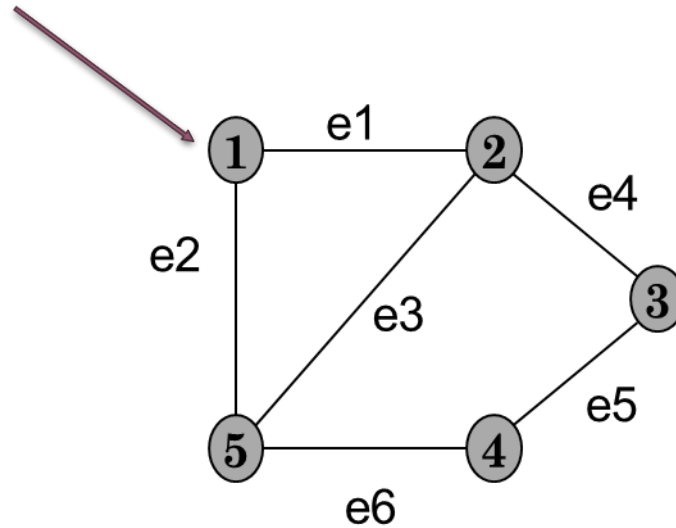
Para implementarmos o TAD Graph em JAVA, temos que decidir como implementar a noção de Vértice e de Aresta (Edge).

Considerações:

- Os grafos tem as arestas e os vértices rotulados
- Uma aresta e um vértice, podem ter elementos de tipos diferentes.  
Exemplo: Um mapa com os Caminho de Ferro de Portugal, os vértices teriam elementos do tipo Estação de Caminho de Ferro e as Arestas informação sobre o percurso que liga as duas estações.
- Usam-se as interfaces em Java para definir o tipo Vertex (vértice) e o tipo Edge (aresta).
- O tipo Vertex caracteriza-se pelo seu rótulo (elemento)
- O tipo Edge caracteriza-se pelo seu rótulo (elemento) e pelos par de vértices que conecta.

# Tipo Vertex - Implementação

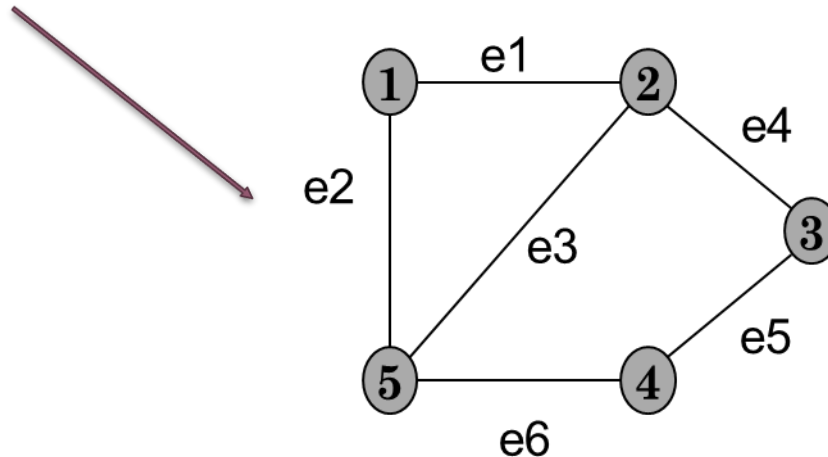
- Um vértice caracteriza-se por guardar um elemento do tipo genérico V.



```
public interface Vertex<V> {  
    public V element();  
}
```

# Tipo Edge - Implementação

- Uma aresta (edge) caracteriza-se por guardar a referência para o par de vértices que conecta e um elemento do tipo E .



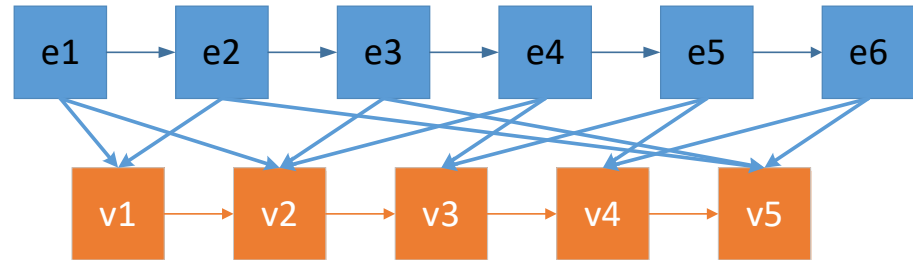
```
public interface Edge<E, V> {  
    public E element();  
    public Vertex<V>[] vertices();  
}
```

# Interface Graph

```
public interface Graph<V, E> {  
    public int numVertices();  
    public int numEdges();  
    public Collection<Vertex<V>> vertices();  
    public Collection<Edge<E, V>> edges();  
    public Collection<Edge<E, V>> incidentEdges(Vertex<V> v) throws InvalidVertexException;  
    public Vertex<V> opposite(Vertex<V> v, Edge<E, V> e) throws InvalidVertexException, InvalidEdgeException;  
    public boolean areAdjacent(Vertex<V> u, Vertex<V> v) throws InvalidVertexException;  
    public Vertex<V> insertVertex(V vElement) throws InvalidVertexException;  
    public Edge<E, V> insertEdge(Vertex<V> u, Vertex<V> v, E edgeElement)  
        throws InvalidVertexException, InvalidEdgeException;  
    public V removeVertex(Vertex<V> v) throws InvalidVertexException;  
    public E removeEdge(Edge<E, V> e) throws InvalidEdgeException;  
}
```

# Implementação usando a estrutura de dados : Listas de arestas

- Map de Arestas.
- Map de Vertices
- Usam-se dicionarios apra mais facilmente obter uma aresta ou um vertice em função do seu elemnto.



```
public class GraphEdgeList<V, E> implements Graph<V, E>

private Map<V,Vertex<V>> vertices;
private Map<E,Edge<E, V>> edges;

public GraphEdgeList() {
    this.vertices = new HashMap<>();
    this.edges = new HashMap<>();
}
```

# Implementação da interface Vertex<V>

- A interface Vertex<V> é implementada como uma inner class de GraphEdgeList: **MyVertex**

```
class MyVertex implements Vertex<V> {  
    V element;  
  
    public MyVertex(V element) {  
        this.element = element;  
    }  
  
    @Override  
    public V element() {  
        return this.element;  
    }  
  
    @Override  
    public String toString() {  
        return "Vertex{" + element + '}';  
    }  
}
```

# Implementação da interface Edge<E,V>

- A interface Edge<E,V> é implementada como uma inner class de GraphEdgeList : **MyEdge**

```
class MyEdge implements Edge<E, V> {  
  
    E element;  
    Vertex<V> vertexOutbound;  
    Vertex<V> vertexInbound;  
  
    public MyEdge(E element, Vertex<V> vertexOutbound, Vertex<V> vertexInbound) {  
        this.element = element;  
        this.vertexOutbound = vertexOutbound;  
        this.vertexInbound = vertexInbound;  
    }  
  
    @Override  
    public E element() {  
        return this.element;  
    }  
  
    public boolean contains(Vertex<V> v) {  
        return (vertexOutbound == v || vertexInbound == v);  
    }  
  
    @Override  
    public Vertex<V>[] vertices() {  
        Vertex[] vertices = new Vertex[2];  
        vertices[0] = vertexOutbound;  
        vertices[1] = vertexInbound;  
  
        return vertices;  
    }  
}
```

# CheckVertex : método auxiliar

Método que tem como objetivo converter o tipo Vertex no objecto concreto, verificando se o mesmo pertence ao grafo.

```
private MyVertex checkVertex(Vertex<V> v) throws InvalidVertexException {  
    if(v == null) throw new InvalidVertexException("Null vertex.");  
  
    MyVertex vertex;  
    try {  
        vertex = (MyVertex) v;  
    } catch (ClassCastException e) {  
        throw new InvalidVertexException("Not a vertex.");  
    }  
  
    if (!vertices.containsKey(vertex.element)) {  
        throw new InvalidVertexException("Vertex does not belong to this graph.");  
    }  
  
    return vertex;  
}
```



# CheckEdge : método auxiliar

Método que tem como objetivo converter o tipo Edge no objecto concreto, verificando se o mesmo pertence ao grafo.

```
private MyEdge checkEdge(Edge<E, V> e) throws InvalidEdgeException {  
    if(e == null) throw new InvalidEdgeException("Null edge.");  
  
    MyEdge edge;  
    try {  
        edge = (MyEdge) e;  
    } catch (ClassCastException ex) {  
        throw new InvalidVertexException("Not an adge.");  
    }  
  
    if (!edges.containsKey(edge.element)) {  
        throw new InvalidEdgeException("Edge does not belong to this graph.");  
    }  
  
    return edge;  
}
```

# Implementação: Inserir um Vértice

- Não é permitido haver dois vértices idênticos.
- Insere-se o vértice no map de vértices.

```
public Vertex<V> insertVertex(V vElement) throws InvalidVertexException {  
    if (existsVertexWith(vElement)) {  
        throw new InvalidVertexException("There's already a vertex with this element.");  
    }  
  
    MyVertex newVertex = new MyVertex(vElement);  
  
    vertices.put(vElement, newVertex);  
  
    return newVertex;  
}
```

# Implementação: Remover um Vértice

1. Remove todas as arestas incidentes
  - Usa o método `incidentEdges` para determinar a lista de arestas a remover
  - Remove cada aresta da lista de arestas
2. Remove o vértice da lista

```
public synchronized V removeVertex(Vertex<V> v) throws InvalidVertexException {  
    checkVertex(v);  
  
    V element = v.element();  
  
    //remove incident edges  
    Iterable<Edge<E, V>> incidentEdges = incidentEdges(v);  
    for (Edge<E, V> edge : incidentEdges) {  
        edges.remove(edge.element());  
    }  
  
    vertices.remove(v.element());  
  
    return element;  
}
```

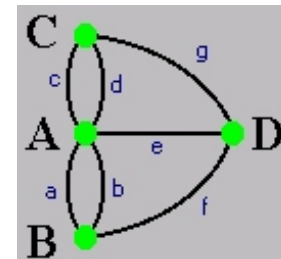
# ADT Graph | Exercícios de implementação



1. Faça *clone* do projeto base **ADTGraph\_Template** (projeto IntelliJ) do *GitHub*:

[https://github.com/pa-estsetubal-ips-pt/ADTGraph\\_FXSmartGraph\\_Template](https://github.com/pa-estsetubal-ips-pt/ADTGraph_FXSmartGraph_Template)

2. Forneça o código dos métodos por implementar, i.e., os que estão a lançar `NotImplementedException` ;
  3. Compile e teste o programa fornecido.
- 
2. Altere o método `main`, de forma a construir o grafo da figura abaixo.





# Programação Avançada

Implementação do TAD –Graph  
Programação Avançada – 2020-21

Bruno Silva, Patrícia Macedo

# Sumário



- Revisão dos algoritmos para percorrer Árvores
- Algoritmos para percorrer Grafos
  - Percorrer Grafos em Largura
  - Percorrer Grafos em Profundidade

# Percorrer Grafos

## **Qual a finalidade de Percorrer um grafo?**

- Procurar se um vértice, ou uma aresta existem.
- Cópia de um grafo ou conversão entre representações diferentes.
- Contagem de números de vértices e/ou arestas.
- Determinação de caminho entre dois vértices ou ciclos, caso existam.

# Percorrer Grafos: Algoritmos

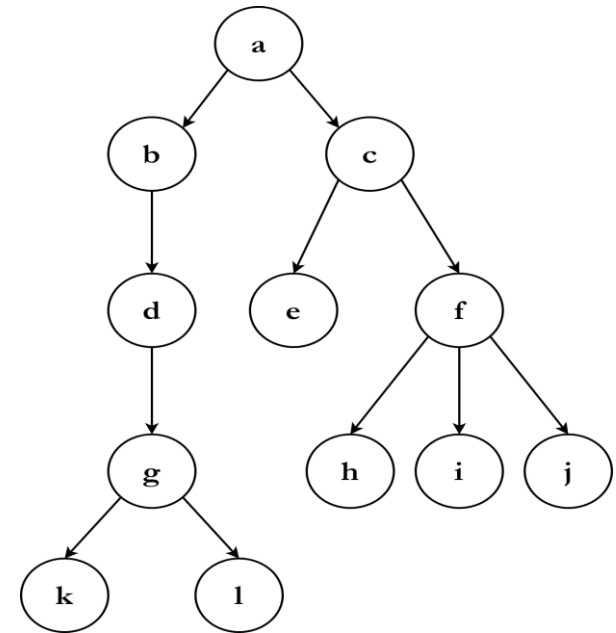
- As estruturas lineares são percorridas normalmente sequencialmente de uma extremidade para a outra.
- As estruturas não lineares existem podem ser percorridas de diversas formas.
- As árvores pode ser percorridas em
  - largura
  - profundidade (pos-order e pre-order)
- Os grafos também podem ser percorridos em
  - largura
  - profundidade



# Percorrer árvores algoritmos (revisão)

**As árvores podem ser percorridas usando duas estratégias base**

- Em Largura (breadth-first)
  - [esq/dir] a b c d e f g h i j k l
  - [dir/esq] a c b f e d j i h g l k
- Em Profundidade (depth-first)
  - Pre-Ordem - a b d g k l c e f h i j
  - Pós-Order - k l g d b e h i j f c a

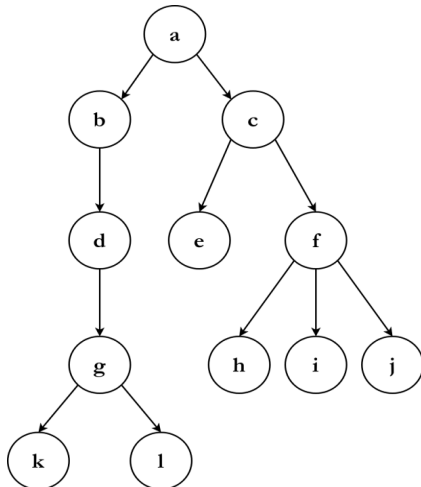


# Percorrer árvores algoritmos

## Algoritmo breadth-first

BFS(arvore)

Coloque a raiz da árvore na **fila**  
 Enquanto a **fila** não está vazia faça:  
     seja **n** o nó que retira da **fila**  
     processe **n**  
     para todo o **f** nó filho de **n**  
         coloque **f** na **fila**



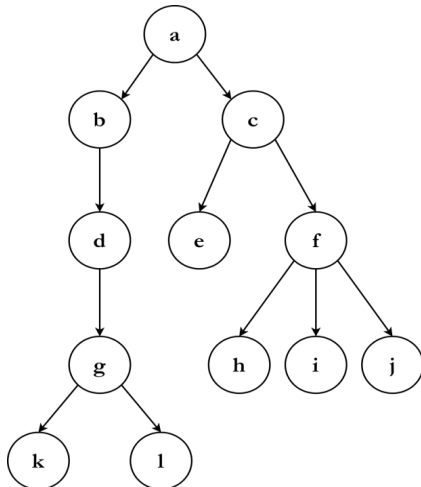
elementos	Fila
	a
a	b c
a b	c d
a b c	d e f
a b c d	e f g
a b c d e	f g
a b c d e f	g h i j
a b c d e f g	h i j k l
a b c d e f g h	i j k l
a b c d e f g h i	j k l
a b c d e f g h i j	k l
a b c d e f g h i j k	l
a b c d e f g h i j k l	

# Percorrer árvores algoritmos

## Algoritmo depth-first

BFS(arvore)

Coloque a raiz da árvore na pilha  
 Enquanto a pilha não está vazia faça:  
     seja **n** o nó que retira da pilha  
     processe **n**  
     para todo o **f** nó filho de **n**  
         coloque **f** na pilha



elementos	Pilha
	a
a	c b
a b	c d
a b d	c g
a b d g	c l k
a b d g k	c l
a b d g k l	c
a b d g k l c	f e
a b d g k l c e	f
a b d g k l c e f	j i h
a b d g k l c e f h	j i
a b d g k l c e f h i	j
a b d g k l c e f h i j	

# Percorrer Grafos :Algoritmos

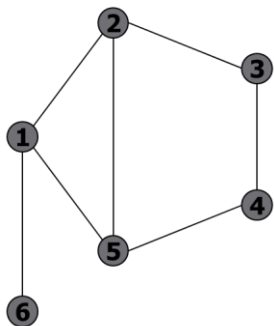
Nos algoritmos de percorrer um grafo, devemos ter em conta:

- Eficiência  $\Rightarrow$  um mesmo local não deve ser visitado repetidamente.
  - Marcam-se os vertices já visitados
- Corretude  $\Rightarrow$  o percurso deve ser feito de modo que não se perca nada.
  - Mantem-se uma coleção (pilha ou fila) com todos os vertices ainda por visitar.
    - Fila [FIFO] – Percorrer em Largura
    - Pilha [LIFO] – Percorrer em Profundidade

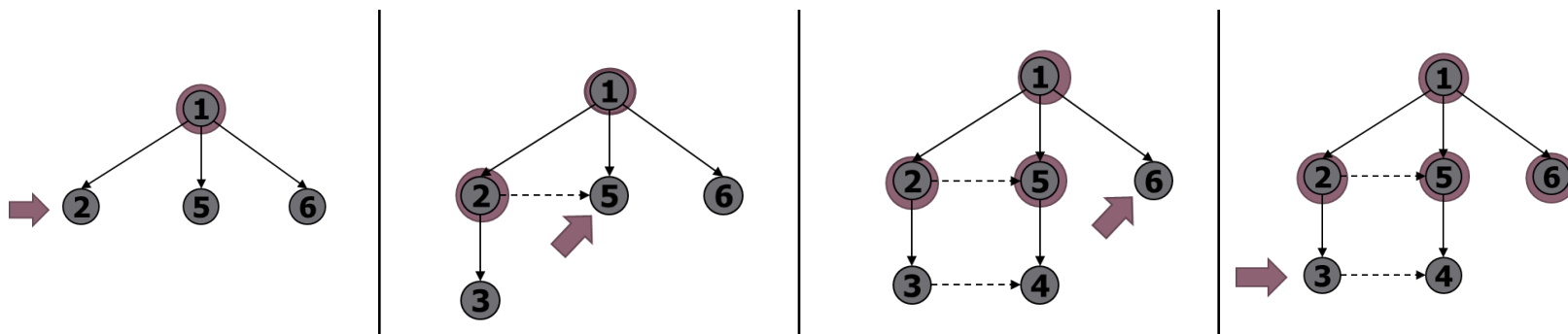
# Pesquisa em Largura

## Breadth-First Search - BFS

Inicia-se por um vértice (denominado vértice raiz) e exploram-se todos os vértices adjacentes a esse. Então, para cada um dos vértices adjacentes, explora-se os seus vértices vizinhos ainda não visitados e assim por diante, até já não existirem mais vértices por visitar.



Grafo a percorrer em Largura a partir do vértice 1 – [ 1,2,5,6,3,4]



# Algoritmo de Pesquisa em Largura

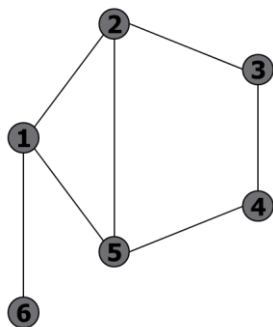
## Algoritmo

```
BFS(Graph, vértice_raiz)
  Marque vértice_raiz como visitado
  Coloque o vértice na fila
  Enquanto a fila não está vazia faça
  • seja v o vértice que retira da fila
  • para cada vértice w adjacente a v faça
    • se w não está marcado como visitado
    • então
      • marque w como visitado
      • insira w na fila
```

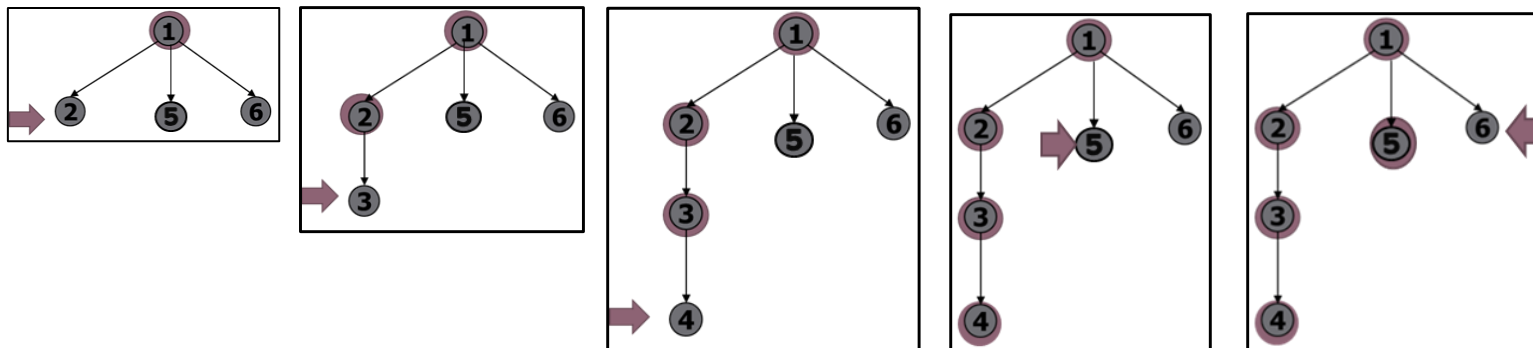
# Pesquisa em Profundidade

## Depth-First Search - DFS

Inicia-se por um vértice (vértice\_raiz) e explora-se tanto quanto possível cada um dos seus ramos, antes de retroceder.



Grafo a percorrer em Profundidade a partir do vértice 1 – [ 1,2,3,4,5,6]



# Algoritmo de Pesquisa em Largura

Algoritmo genérico

```
DFS(Graph, vértice_raiz)
  Marque vértice como visitado
  Coloque o vértice na pilha
  Enquanto a pilha não está vazia faça
  • seja v o vértice que retira da pilha
    para cada vértice w adjacente a v faça
    • se w não está marcado como visitado
    • então
      • marque w como visitado
      • insira w na pilha
```



# Implementação dos Algoritmos

- Existem 2 abordagens para implementação:
  - Interna à classe – o método é implementado como método da classe que implementa a interface Graph.
  - Externa a classe – o método é implementado numa classe externa à classe que implementa Graph.

## Interna

- Utiliza-se um conjunto para guardar os vértices já visitados, ou coloca-se um atributo visitado no nó do vértice.
- Implementa-se o método como método da classe Graph.

## Externa

- Utiliza-se um conjunto para guardar os vértices já visitados.
- Implementa-se o método como método externo à classe.

# ADT Graph | Exercícios de implementação



Continuando com o código iniciado na aula anterior sobre implementação do ADT Grafo.

1.

a) Acrescente na interface Graph o método

```
Iterable<Vertex<V>> DFS(<Vertex<V>> v) throws InvalidVertexException;
```

b) Faça a Implementação do método na classe GraphEdgeList.

c) Construa um teste JUnit para testar o método DFS.

Sugestão: use os dados do exemplo do slide 11.

2. Repita o exercício 1 para o algoritmo BFS.

# ADT Graph | Exercícios de implementação

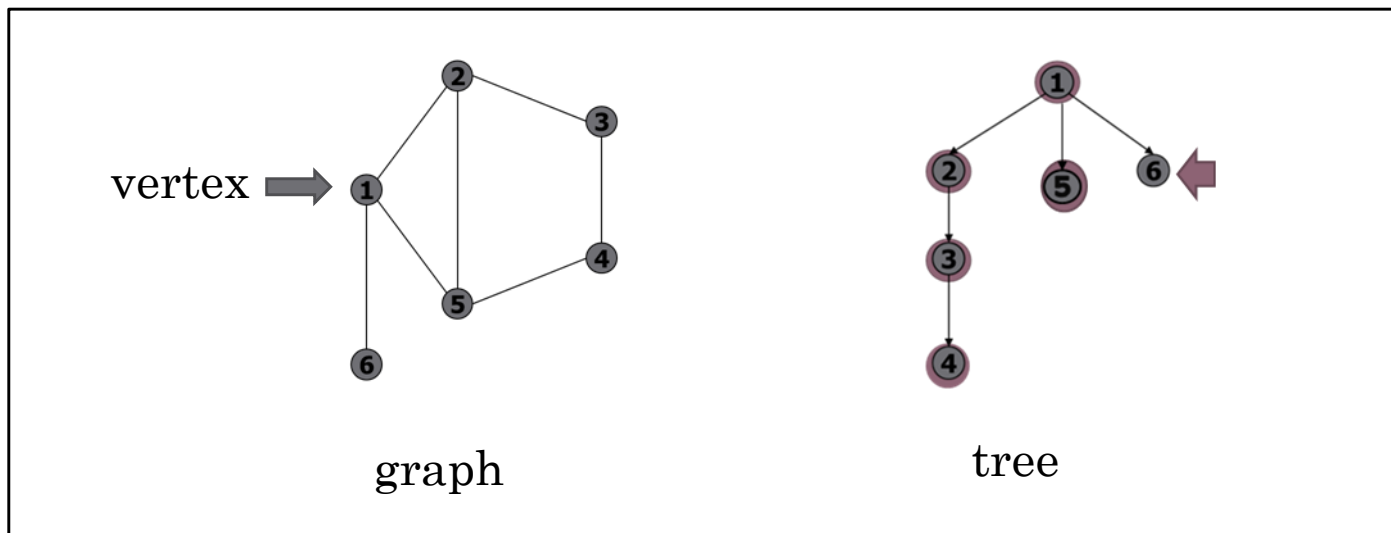


3. Utilize o ADT Tree disponibilizado, nas aulas integrando-o no projeto em que está a trabalhar.

Crie a classe `GraphUtils<V,E>`, e implemente o seguinte método estático.

`Tree<V> treeDFS(Graph<V,E> graph , Vertex<V> vertex)`

que devolve a árvore resultante de percorrer o grafo em profundidade



# ADT Graph | Exercícios de implementação

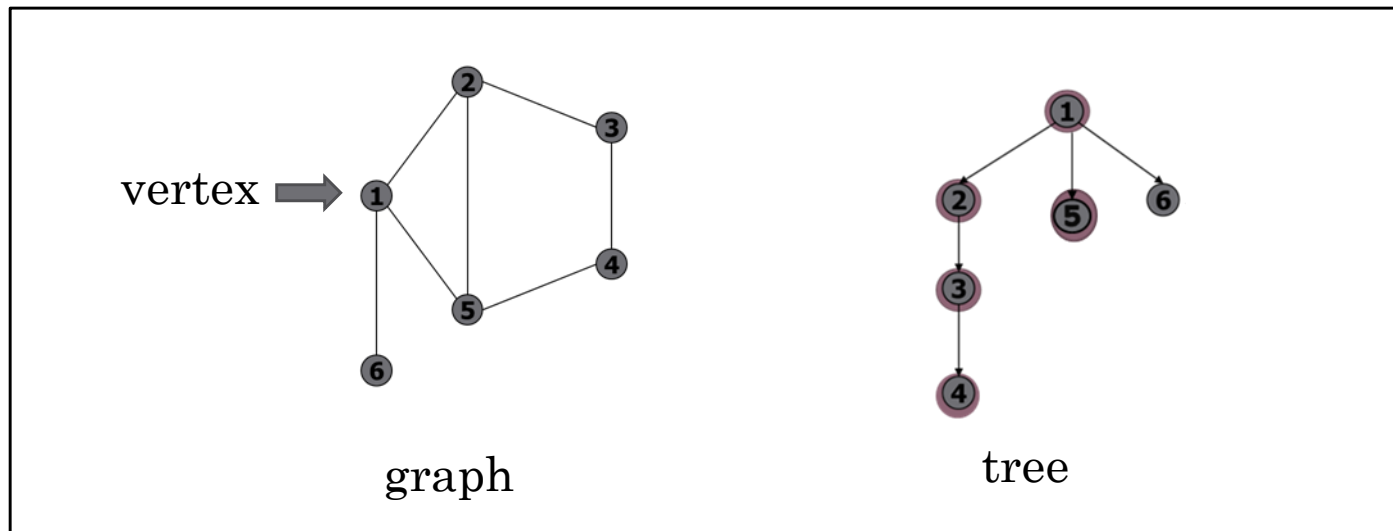


3. Utilize o ADT Tree disponibilizado, nas aulas integrando-o no projeto em que está a trabalhar.

Crie a classe GraphUtils, e implemente o seguinte método estático.

```
public static <V> Tree<V> treeDFS(Graph<V,?> graph , Vertex<V> vertex)
```

que devolve a árvore resultante de percorrer o grafo em profundidade



# ADT Graph | Exercício Solução

```
public class GraphUtils {

    public static <V> Tree<V> treeDFS(Graph<V,?> graph , Vertex<V> vertex)throws InvalidVertexException{
        Vertex<V> v,w;

        TreeLinked<V> tree = new TreeLinked(vertex.element());
        Stack<Vertex<V>> stack = new Stack();
        HashMap<V,Position<V>> treeMap= new HashMap<>();
        Set<Vertex<V>> visited= new HashSet();

        stack.push(vertex);
        visited.add(vertex);
        treeMap.put(vertex.element(), tree.root());

        while(!stack.empty()){
            v=stack.pop();
            Position<V> parent = treeMap.get(v.element());

            for(Edge edge: graph.incidentEdges(v)){
                w=graph.opposite(v,edge);
                if(!visited.contains(w))
                {
                    visited.add(w);
                    stack.push(w);
                    Position<V> pos = tree.insert(parent, w.element());
                    treeMap.put(w.element(),pos);
                }
            }
        }
        return tree;
    }
}
```

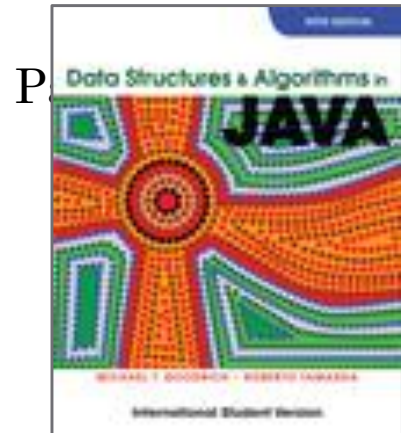
- Instanciar estruturas de suporte.
- O Map servirá para fazer uma ligação entre os elementos do vértices do grafo e os nós colocados na árvore.

- Atualizar vértices visitados
- Atualizar pilha
- Construir árvore: inserir nó.
- Atualizar map

# Estudar e Explorar

- Percorrer Grafos: Em profundidade, em largura
- Visualizar:  
<https://www.cs.usfca.edu/~galles/visualization/DFS.html>  
<https://www.cs.usfca.edu/~galles/visualization/BFS.html>

- Ler mais em:  
P (87)





# Programação Avançada

Implementação do TAD –Graph  
Programação Avançada – 2020-21

Bruno Silva, Patrícia Macedo

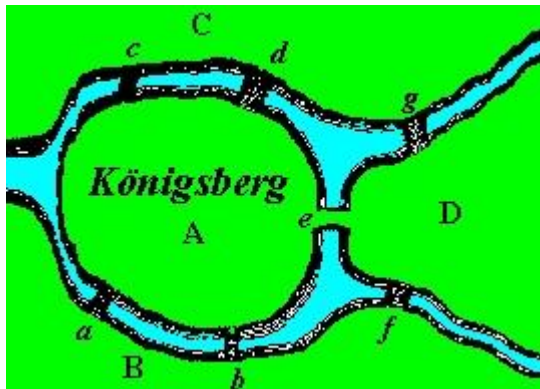
# Sumário



- Noção de Caminho Mais Curto ou de Menor Custo
- Algoritmo Dijkstra
- Algoritmo do Caminho mais Curto
- Implementação dos Algoritmos no TAD Graph

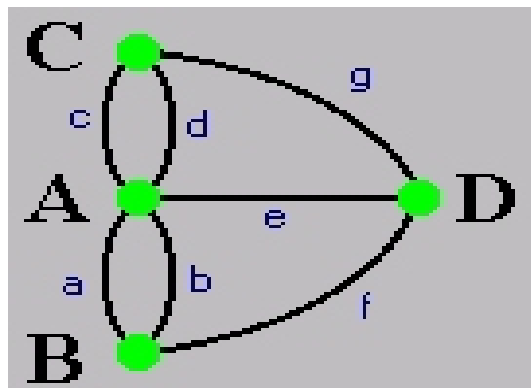


# Procura do Caminho mais curto



Qual o caminho mais curto para ir de A-D ?

Se as pontes não tiverem uma distância, ou custo associada, diremos que o caminho mais curto, é ir de A a D pela ponte *e*.

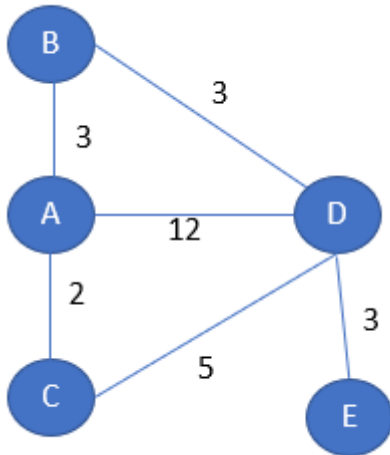


Se as pontes tiverem portagem.

- *a, d, g, f* - 2 Euros
- *c, b* - 3 euros
- *e* - 10 euros

Neste caso o caminho de menor custo é seguir pelas pontes *d, g* para chegar de A a D.

# Procura do Caminho mais curto



- Qual o caminho de menor custo entre A e E?
- Qual o caminho de menor curso ( que percorre um menor numero de arestas), entre A e E ?

# Procura do Caminho mais curto /menor custo

1. Qual o caminho mais curto (ou de menor valor) entre dois vértices de um graph?
  - O caminho mais curto entre o ponto A e o ponto B, é aquele em que a soma do valor das arestas percorridas é menor.
2. Como se determina o valor de uma aresta ?
  - Caso estejamos perante um graph valorado usa-se o valor de associado a cada aresta para determinar o caminho de menor valor entre dois pontos.
  - Nos restantes casos, considera-se que cada aresta percorrida tem o valor 1. O caminho mais curto é aquele que percorrer menos arestas.

# Algoritmo Dijkstra

- **Input:**

- Um graph valorado  $G = (V, E)$ ,  $w : E \rightarrow \mathbb{R}$  (cada aresta tem um valor associado / peso)
- um nó de partida  $s$ .

- **Output:**

- Caminho mais curto de  $s$  para todos os outros nós do graph.

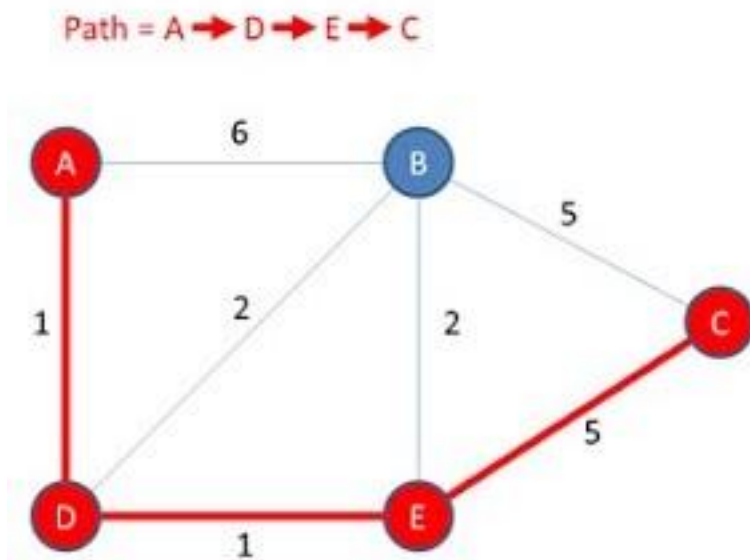
**O custo (ou distância, ou peso) de um caminho é igual ao somatório do custo das arestas que constituem o caminho, ou infinito se não existir caminho.**

# Algoritmo Dijkstra – Descrição informal

1. É atribuída uma distância para todos os pares de vértices. Todos os vértices são inicializados com uma distância infinita, menos para o vértice de origin, que é inicializado a zero.
2. Marque todos os vértices como **não visitados** e defina o vértice inicial como vértice corrente.
3. Para este vértice corrente, considere todos os seus vértices vizinhos não visitados e calcule a distância a partir do vértice. Se a distância for menor do que a definida anteriormente, substitua a distância pela nova distância calculada.
4. Quando todos os nós adjacentes do vértice corrente forem visitados, marque-o como visitado, o que fará com que ele não seja mais analisado (sua distância é mínima e final).
5. Selecione para vértice corrente o vértice não visitado com a menor distância (a partir do vértice inicial) e continue a partir do passo 3, **até todos os nós já terem sido visitados**.

# Ver o Algoritmo em funcionamento

- <https://www.youtube.com/watch?v=pVfj6mxhdMw>



Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

# Algoritmo Dijkstra – Pseudocódigo

## Dijkstra

Input - (**graph**, **origin**)

Output - **costs**[] e **predecessors**[]

BEGIN

FOR EACH VERTEX **v** in **graph**

**costs**[**v**] <- Infinit

**predecessor**[**v**] <- -1

END\_FOR

**costs**[**origin**] <- 0

**S** <- {all vértices of **graph**}

WHILE (**S** IS NOT EMPTY) DO:

**u** <- findLowerVertex(**graph**, **costs**[]) é o vértice do **graph** com menor custo

IF (**costs**[**u**] = Infinit) RETURN

remove(**S**, **u**)

FOR EACH adjacent vertex **v** of vertex **u** DO:

**cost** <- **costs**[**u**] + cost between **u** and **v**;

IF (**cost** < **costs**[**v**]) THEN

**costs**[**v**] <- **cost**;

**predecessor**[**v**] <- **u**;

END\_IF

END\_FOR

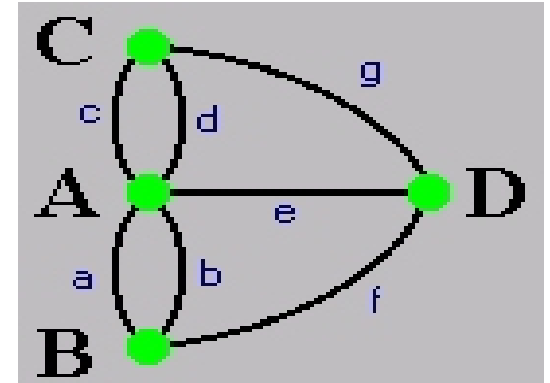
END\_WHILE

END Dijkstra

# Implementação do algoritmo Dijkstra

## Considerações Iniciais

- Consideramos que temos a situação do problema das Pontes de Königsberg.
- Os vértices são instanciados pela classe Local
- As arestas são instanciadas pela classe Bridge (representa as pontes)



```
public class Local {  
    private String name;  
  
    @Override  
    public String toString() {  
        return "Local{" +  
            name +  
            '}';  
    }  
  
    public Local(String name) {  
        this.name = name;  
    }  
}
```

```
public class Bridge {  
    private String name;  
    private int cost;  
  
    public Bridge(String name, int cost) {  
        this.name = name;  
        this.cost = cost;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public int getCost() {  
        return cost;  
    }  
}
```



# Implementação do algoritmo Dijkstra

```
private void dijkstra(Vertex<Local> orig,
                    Map<Vertex<Local>, Double> costs,
                    Map<Vertex<Local>, Vertex<Local>> predecessors) {

    costs.clear();
    predecessors.clear();
    List<Vertex<Local>> unvisited = new ArrayList<>();

    for (Vertex<Local> v : graph.vertices()) {
        costs.put(v, Double.POSITIVE_INFINITY);
        predecessors.put(v, null);

        unvisited.add(v);
    }
    costs.put(orig, 0.0);
    while(!unvisited.isEmpty()) {
        Vertex<Local> lowerCostVertex = findLowerCostVertex(unvisited, costs);
        unvisited.remove(lowerCostVertex);
        for (Edge<Bridge, Local> incidentEdge : graph.incidentEdges(lowerCostVertex)) {
            Vertex<Local> opposite = graph.opposite(lowerCostVertex, incidentEdge);
            if( unvisited.contains(opposite) ) {
                double cost = incidentEdge.element().getCost();
                double pathCost = costs.get(lowerCostVertex) + cost;
                if( pathCost < costs.get(opposite) ) {
                    costs.put(opposite, pathCost);
                    predecessors.put(opposite, lowerCostVertex);
                }
            }
        }
    }
}
```

- **input:** local de **orig** variáveis de entrada
- **outputs:** **costs** e **predecessor** vão ser o resultado da execução do algoritmo. Usamos **Map** em vez de vetores, pois os vértices não são numerados, mas identificados pelo seu rótulo

# Implementação do algoritmo calcula caminho de menor custo

- O algoritmo do cálculo do caminho de menor custo, usa o algoritmo Dijkstra.
- Assim a execução do algoritmo de Dijkstra é um **passo** do algoritmo para cálculo de caminho de menor custo e do seu valor.

Minumum\_Cost\_Path

Input - (`graph`,`origin`,`destination`)

output - `paths[]` e `cost`

```
vOri <- pesquisa_vertice(graph,origin)
vDst <- pesquisa_vertice(graph,destination)
Disjktra(graph,vOri, costs[], predecessors[])
path<-[]
v <- vDst
```

```
WHILE v ≠ vOri DO
    path<-insert(path, 0,v)
    v<-predecessors[v]
```

```
END_WHILE
```

```
path<-insert(path, 0,v)
```

```
cost<-costs[vDst]
```

```
END Minumum_Cost_Path
```

`costs[]` e `predecessor[]` vão ser o resultado da execução do algoritmo Dijkstra

# Implementação do algoritmo calcula caminho de menor custo

```
public int minimumCostPath(String origName, String dstName, List<Local> path) {

    Vertex<Local> vOrig = findLocal(origName);
    Vertex<Local> vDst= findLocal(dstName);

    if( vOrig==null) throw new IllegalArgumentException("orig does not exist");
    if(vDst==null) throw new IllegalArgumentException("dst does not exist");
    if(path == null ) throw new IllegalArgumentException("path list reference is null");

    Map<Vertex<Local>, Double> costs = new HashMap<>();
    Map<Vertex<Local>, Vertex<Local>> predecessors = new HashMap<>();

    dijkstra(vOrig, costs, predecessors);

    path.clear();

    boolean complete = true;
    Vertex<Local> actual = vDst;
    while( actual != vOrig) {
        path.add(0, actual.element());
        actual = predecessors.get(actual);
        if( actual == null) {
            complete = false;
            break;
        }
    }
    path.add(0, vOrig.element());
    if(!complete) {
        path.clear();
        return -1;
    }
    return costs.get(vDst).intValue();
}
```

# ADT Graph | Exercícios de implementação

Continuando com o código iniciado na aula anterior sobre implementação do ADT Grafo. Faça download do package model

[https://github.com/pa-estsetubal-ips-pt/Model\\_Bridges.git](https://github.com/pa-estsetubal-ips-pt/Model_Bridges.git)

e integre-o no projeto do ADT Graph que tem vindo a trabalhar.

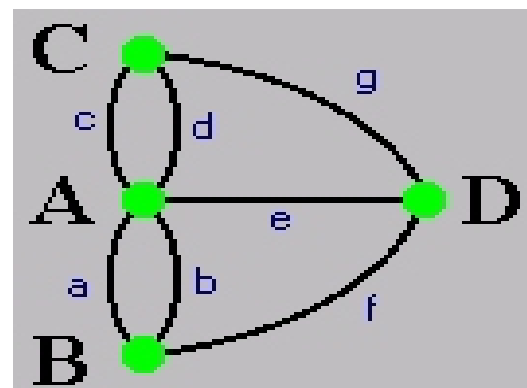
1. Analise a classe Brige Manager e execute o main de teste disponibilizado – MainTest.
2. Presentemente o método do caminho mais curto indica-nos apenas os locais que o constituem. No caso de haverem pontes paralelas (arestas paralelas), é importante saber quais as pontes que compõe o caminho

**Altere** os métodos dijkstra e minimumCostPath de forma a este último devolver também a listas das pontes (Edges) que compõe o caminho de menor custo.

Se as pontes tiverem portagem.

- $a, d, g, f$  - 2 Euros
- $c, b$  - 3 euros
- $e$  - 10 euros

Neste caso o caminho de menor custo é seguir pelas pontes  $d, g$  para chegar de A a D.





# Programação Avançada

## **Introdução aos Padrões de Software** **Iterator Pattern**

Programação Avançada – 2020-21

Bruno Silva, Patrícia Macedo

# Sumário



- Padrões de Desenho versus Padrões de Arquitetura
- Definição de Padrão de Desenho
- O Padrão Iterator

# Introdução

Em Engenharia de Software, um padrão é **uma solução geral** para um problema que ocorre com frequência dentro de um determinado contexto do desenho de software.

- Existem varias classificações para os padrões, na uc de Programação Avançada vamos usar a seguinte classificação:
  - Padrões de Arquitetura (Architectural Patterns)
  - Padrões de Desenho (Design Patterns)

# Padrões de Software

- Padrões de Arquitetura (Architectural Patterns)

Resolvem um problema típico da arquitetura de software. Definem formas de organizar os vários componentes de um sistema de software

MVC, MVP, DAO

- Padrões de Desenho (Design Patterns)

- Resolvem problemas específicos e localizados (como criar uma variável global, como adaptar o comportamento de uma classe, como criar uma família de classes etc...)



# Vantagens da Utilização de Padrões

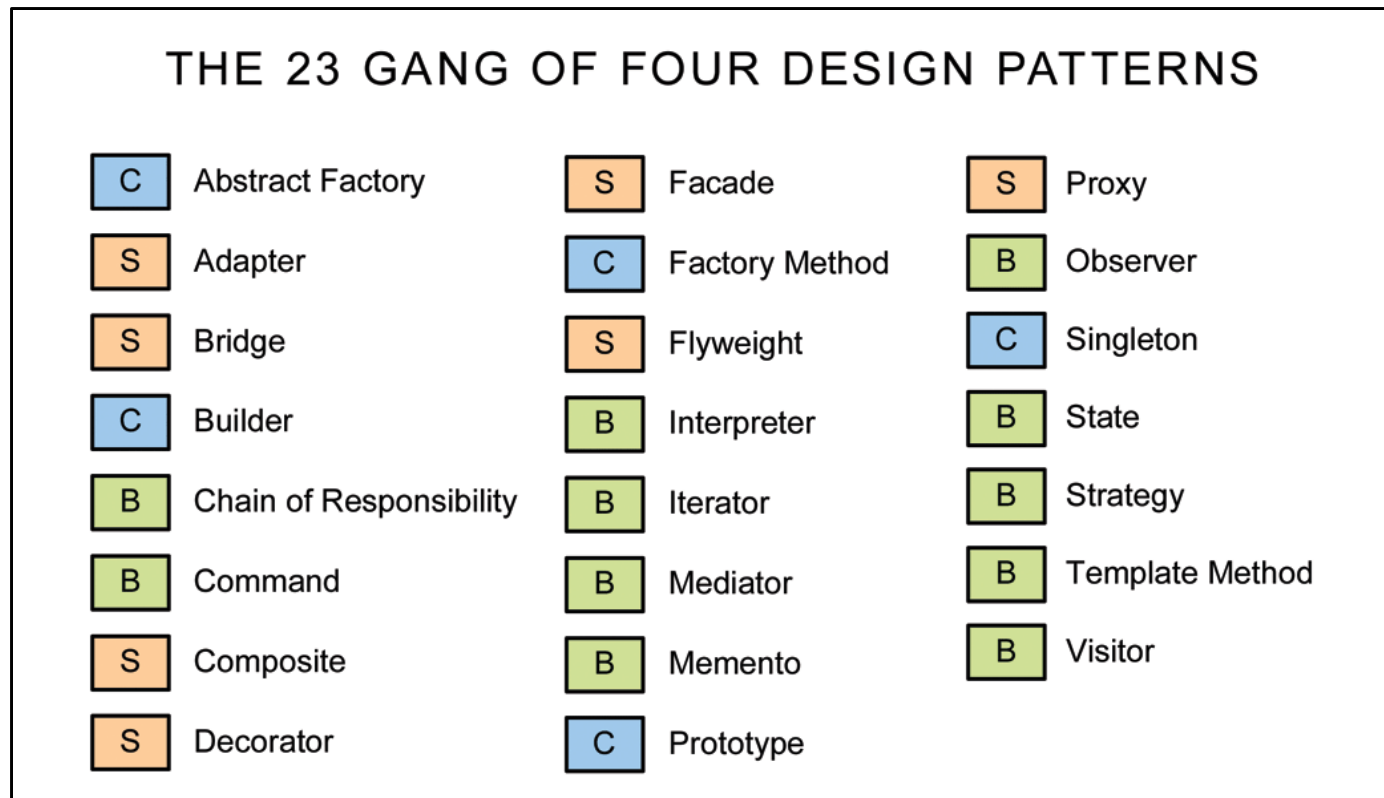
- Aprende-se com a experiência de outros;
- Utiliza-se soluções amplamente testadas;
- Permite utilizar uma linguagem comum entre os *designers* e programadores. Melhora-se assim a comunicação entre a equipa e a documentação dos sistemas;
- Leva ao uso de boas práticas no desenvolvimento de software orientado a objetos, obtendo-se assim software de melhor qualidade.

# Especificação de um Padrão de Desenho

- Um padrão de desenho deverá ser especificado indicando:
  - O **nome** (cria um vocabulário de padrões)
  - O **problema** (diz quando aplicar o padrão)
  - A **solução** (descreve os elementos do design)
  - As **consequências** (de aplicar o padrão de desenho)

# Padrões de desenho - GoF

“Design Patterns: Elements of Reusable Object-Oriented Software” é um livro de engenharia de software escrito por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides **que descreve 23 padrões clássicos de desenho**. Os 4 autores do livro ficaram conhecidos por GoF (Gang of Four).



# O Padrão Iterator (Motivação)

- **Problema Genérico**

- Uma coleção é um contendor de elementos, mas muitas vezes precisamos de percorre-los sequencialmente. **Como o Fazer?**

Quando estamos perante uma coleção do tipo List, normalmente percorremos os elementos em função do seu índice.

```
for (int i = 0; i < list.size(); i++)  
    System.out.print(list.get(i) + " ");
```

- Mas se quisermos percorrer uma coleção do tipo Set, Map, Tree...então normalmente usamos um ciclo foreach

```
for (Integer i: set)  
    System.out.print(i + " ");
```

- Os ciclos *foreach* só são possíveis de realizar se a coleção for *Iterable*
- Uma coleção que implementa a interface *Iterable*, é uma coleção que implementa o padrão *Iterator*

# O Padrão Iterator (Motivação)

- **Problema Concreto**

- Temos o nosso TAD Stack, que é uma coleção tipo STACK e queremos percorrer uma instancia de STACK sequencialmente do topo para a base, sem destruir o stack.

```
Stack<Integer> stack = new StackArray();  
  
for (int i = 0; i < 20; i++)  
    stack.push( 100 - i);  
  
for (Integer i: stack)  
    System.out.print(i + " ");
```

- **Solução**

- Para percorrermos o stack, usando um ciclo foreach, temos que implementar o padrão Iterator.

# O Padrão Iterator (Motivação)

- **Objectivo:** Tornar o classes do tipo Stack Iteráveis

Como ?

1. Estender a interface Stack de **Iterable**
2. Implementar o método **iterator** na classe de implementação do Stack
3. Criar uma inner classe na implementação do Stack que implemente a **interface Iterator**

---

```
public interface Iterable<E>{  
    Iterator<E> iterator();  
}
```

```
public interface Iterator<E>{  
    //devolve true se existe proximo elemento  
    Boolean hasNext();  
    //devolve o proximo element da sequência  
    T next();  
}
```

# O Padrão Iterator (Motivação)

- Resolução

```
public interface Stack<E> extends Iterable<E>
```



```
public class StackArray<E> implements Stack<E> {

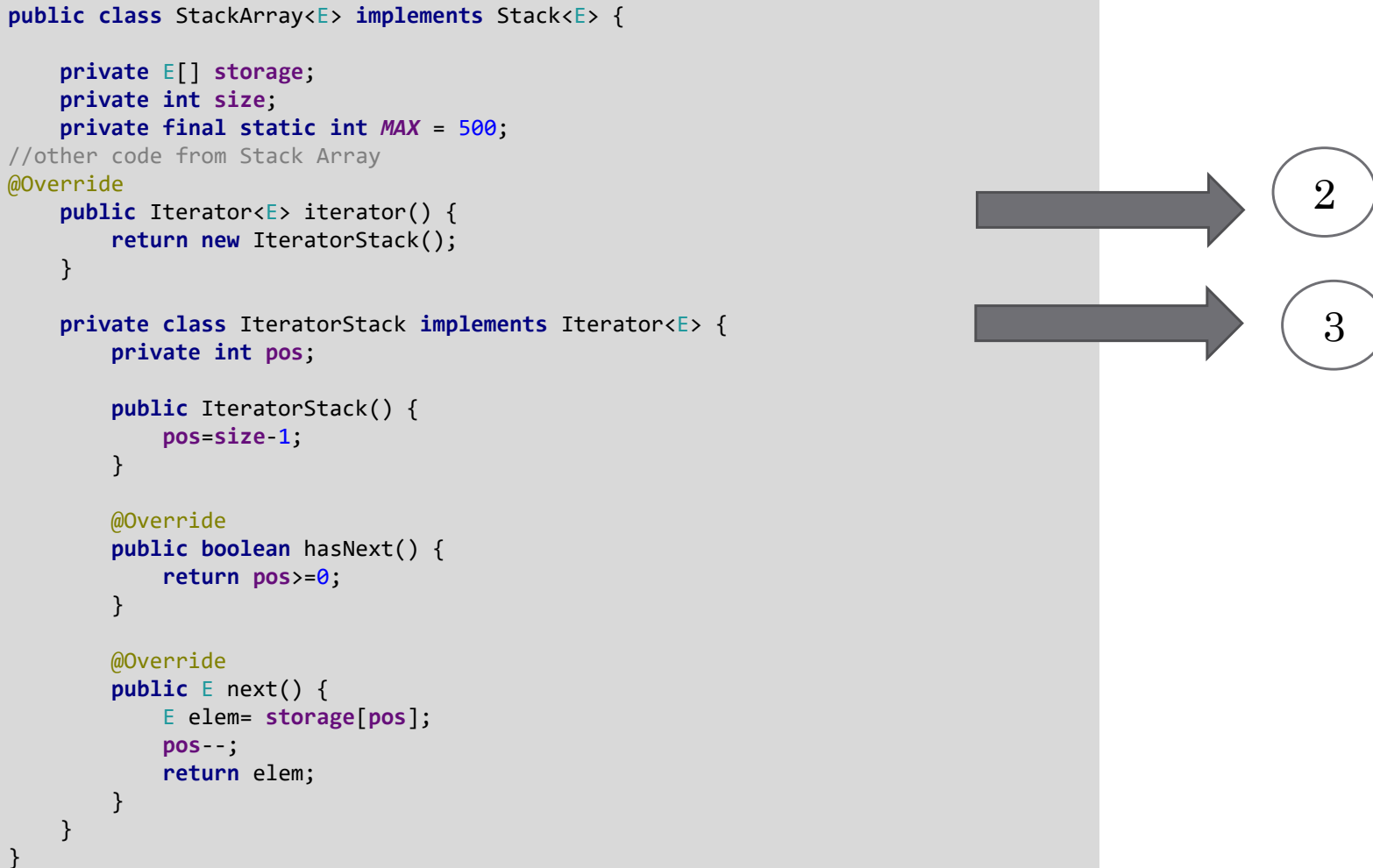
    private E[] storage;
    private int size;
    private final static int MAX = 500;
    //other code from Stack Array
    @Override
    public Iterator<E> iterator() {
        return new IteratorStack();
    }

    private class IteratorStack implements Iterator<E> {
        private int pos;

        public IteratorStack() {
            pos=size-1;
        }

        @Override
        public boolean hasNext() {
            return pos>=0;
        }

        @Override
        public E next() {
            E elem= storage[pos];
            pos--;
            return elem;
        }
    }
}
```



# Iterator| Exercícios(1)

---



- Crie um projeto a partir do template:

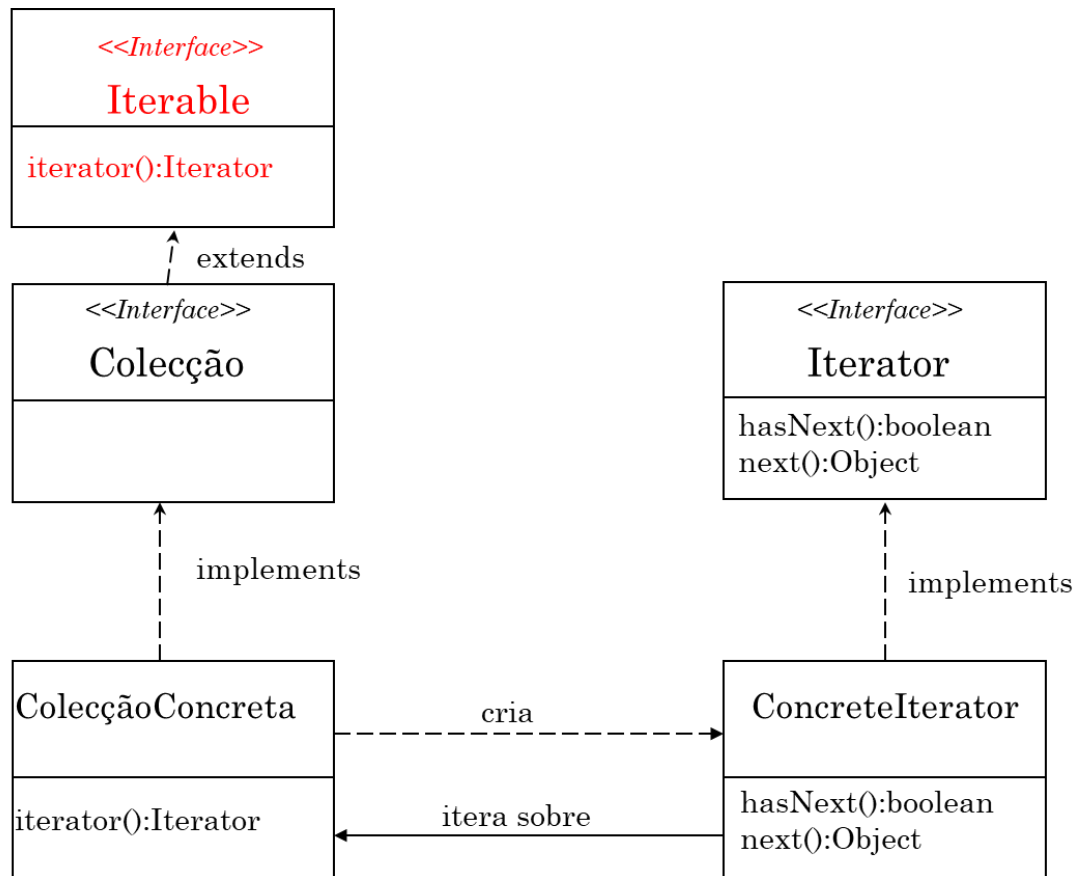
[https://github.com/pa-estsetubal-ips-pt/IteratorExamples\\_Template.git](https://github.com/pa-estsetubal-ips-pt/IteratorExamples_Template.git)

- Reveja a implementação do Padrão Iterator para a classe StackArray
- Implemente o método iterator na classe StackLinked, de forma a torna-la Iterável.
- Torne a classe Tree Iterável, implementando o iterador de forma a disponibilizar o elementos da árvore na sequência pre-order.



## Iterator Pattern (usando as classes e interfaces do JAVA)

- Um padrão é uma solução padronizada para um problema comum
- O padrão Iterator cria uma forma padronizada de iterar sequencialmente os elementos de um contentor



# Participantes do Padrão

- **Iterator**
  - Define uma interface para aceder e percorrer os elementos
- **ConcreteIterator**
  - Implementa a interface Iterator
  - Mantém a informação da posição actual de iteração
- **Coleção**
  - Define uma interface para criar um objeto Iterator
- **ColeçãoConcreta**
  - Implementa a interface que cria o Iterator para retornar o IteradorConcreto apropriado

## Revendo a receita -> solução padronizada

1. Torne a interface do tipo que pretende tornar iterável a estender de Iterable.
2. Na colecção que pretenda implementar o iterador, implemente o método `Iterator<E>iterator()`, que devolve uma instancia do iterador da classe `ConcreteIterator`.
3. Crie uma classe interna privada `ConcreteIterator` que implementa a interface `Iterator`.
4. Implemente os métodos `next()` e `hasNext()`, na classe `ConcreteIterator`.

# Iterator| Exercícios Adicionais

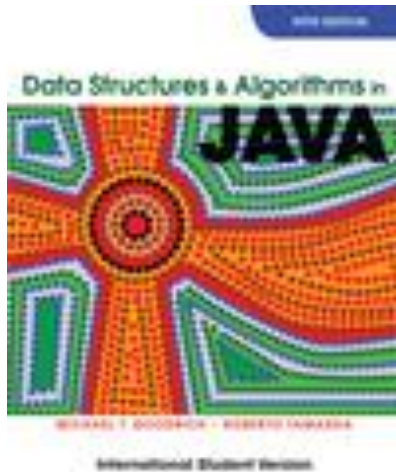


Implemente para o TAD Stack um iterador que percorra os elementos da base para o topo.

- Defina na interface Stack o método `Iterator<E> iteratorInverse()`
- Implemente o método para o `StackArray` e `StackLinked`
- Teste este novo iterador no main, usando os métodos `hasNext` e `next`.

```
Iterator<Integer> it= stack.iteratorInverse();  
while(it.hasNext())  
    System.out.print(it.next() + " ");
```

# Estudar e Rever



**Pagina 254-260**

- [https://www.tutorialspoint.com/design\\_pattern/iterator\\_pattern.htm](https://www.tutorialspoint.com/design_pattern/iterator_pattern.htm)
- <https://refactoring.guru/design-patterns/iterator>



# Programação Avançada

## **Strategy Pattern**

Programação Avançada – 2020-21

Bruno Silva, Patrícia Macedo

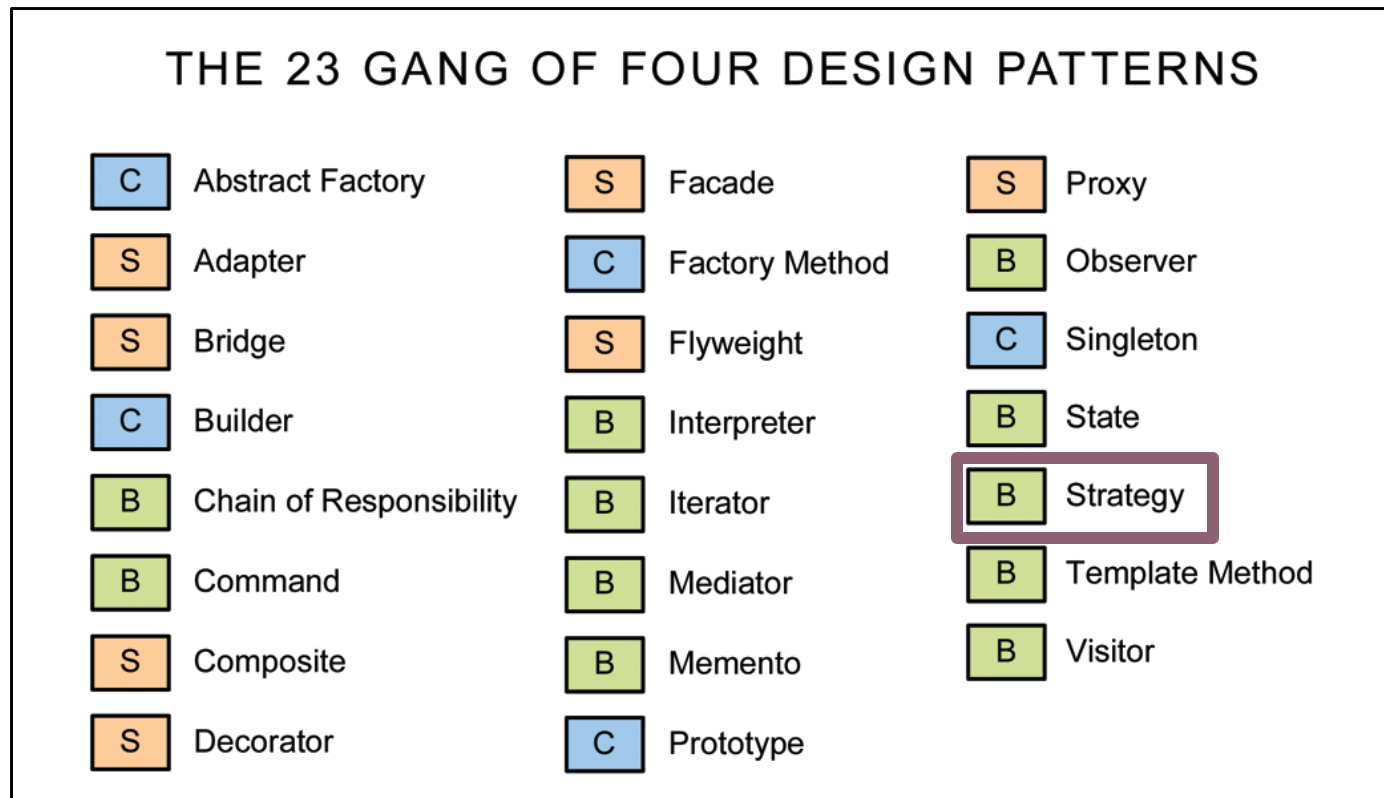
# Sumário



- Pattern Strategy
- Exmplo de Aplicação

# Strategy Pattern

- Padrão comportamental
- Tal como o nome sugere permite que um objecto **troque de estratégia** de comportamento em tempo de execução.





# Strategy (Motivação)

- **Problema Genérico**

- Existe uma classe que o seu **comportamento varia** dependendo das circunstâncias ou do seu tipo.
- Para tal recorremos a uma estrutura **switch case** e em função do tipo, executamos um conjunto de instruções código diferente.
- Como garantir a extensibilidade e manutenibilidade?

**Nota:**

O uso de enumerados para representar tipos diferentes de classes, vai contra as boas práticas da programação orientada a objetos e é de evitar:

- Mais difícil de manter
- Mais difícil de estender

```
public class X {  
  
    public enum TYPE {TYPE1,TYPE2,TYPE3};  
    private TYPE type;  
    private int id;  
    //...  
  
    public X(int id, TYPE type) {  
        this.id = id;  
        this.type=type;  
        //....  
    }  
  
    public void metodo1()    {  
        switch (type) {  
            case TYPE1:  
                //code1  
                break;  
            case TYPE2:  
                //code2  
                break;  
            case TYPE3:  
                //code3  
                break;  
        }  
    }  
}
```

# Strategy (Motivação)

## Problema concreto

- Um grupo é constituído por programadores e pretende-se calcular o índice de potencial de um grupo
- A formula de calculo, varia em função do perfil pretendido para o grupo
- É possível o grupo mudar de perfil, ao longo do tempo.
- Como garantir que temos uma aplicação onde é fácil
  - **adicionar/remover novos tipos de perfis**
  - **adicionar novos métodos que variam em função do perfil do grupo**

```
public class Group {  
  
    public enum TYPE {DIVERSITY, SENIOR, MULTYSKILLS}  
    private TYPE type;  
    private String name;  
    private Map<Integer, Programmer> personList;  
  
    public Group(String name, Group.TYPE type) {  
        this.name = name;  
        this.type = type;  
        personList = new HashMap<>();  
    }  
    public float calculateGlobalIndex() {  
        int value = 0;  
        float res = 0.0f;  
        switch (type) {  
  
            case DIVERSITY:  
                /* res= X/Y  
                X= programmers with more than 5 years of experience  
                Y= programmers with less than 5 years of experience */  
  
            case SENIOR:  
                /* res= X/Y  
                X= programmers with more than 10 years of experience  
                Y= group size*/  
  
            case MULTYSKILLS:  
                /* res= X/Y  
                X= programmers with specialist in more than 5 languages  
                Y= group size*/  
  
        }  
        return res;  
    }  
}
```

# Strategy (Motivação)

## Problema concreto

- Pretende-se calcular o índice global de potencial de um grupo
- A formula de calculo, varia em função do perfil do grupo
- É possível o perfil associado a um grupo mudar ao longo da vida do grupo
- Como garantir que temos uma aplicação onde é fácil
  - A. adicionar/remover novos tipos de perfis**
  - B. adicionar novos métodos que variam em função do perfil do grupo**

## Solução 1

- A. Adicionar mais um valor ao enumerado
- B. Adicionar mais um case no método `float calculateGlobalIndex()`


## Solução 2 - polimorfismo

- A. Definir a class `Group` como abstrata e o método `float calculateGlobalIndex()` como abstrato
- B. Implementar três subclasses de `Group` uma para cada perfil do grupo

# Strategy (Motivação)

## Solução 1

- A. Adicionar mais um valor ao enumerado
- B. Adicionar mais um case no método float `calculateGlobalIndex()`



```
public enum TYPE {DIVERSITY, SENIOR, MULTYSKILS, SPECIALIZED};

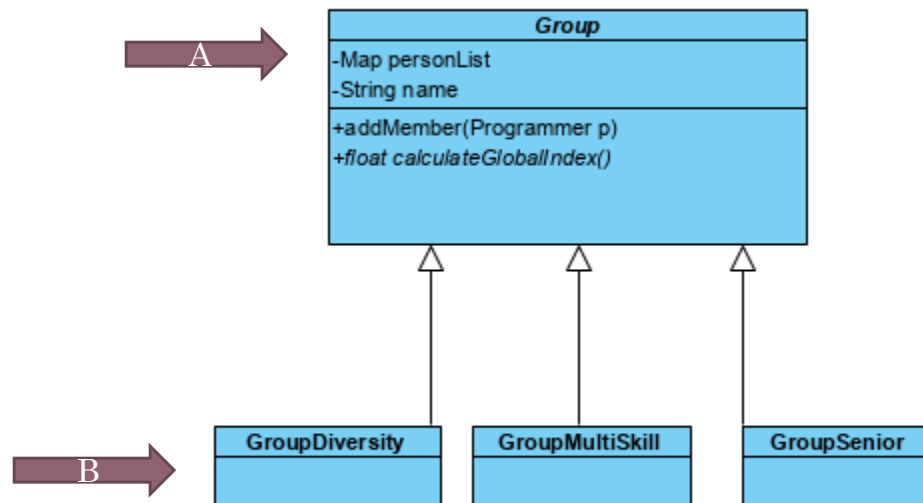
public float calculateGlobalIndex() {
    switch (type) {

        case DIVERSITY:
            //code 1
            break;
        case SENIOR:
            // code 2
            break;
        case MULTYSKILS:
            //code 3
            break;
        case SPECIALIZED:
            //code 4
            break;
    }
    return res;
}
```

# Strategy (Motivação)

## Solução 2 – polimorfismo

- A. Definir a class Group como abstrata e o método `float calculateGlobalIndex()` como abstrato.
- B. Implementar três subclasses de Group uma para cada perfil do grupo



## Strategy (Motivação)

No caso do Polimorfismo Se o grupo quisesse mudar de perfil em tempo de execução, teria que se instanciar um novo grupo, perdendo-se os dados do anterior.

```
Group gr1 = new GroupDiversity("PA-23");  
gr1.addMember(p1,p2,p3,p4);  
  
System.out.printf("\nGrupo %s , GlobalIndex- %f", gr1.toString(),gr1.calculateGlobalIndex());  
  
gr1 = new GroupMultiSkills("PA-23");   
  
System.out.printf("\nGrupo %s , GlobalIndex- %f", gr1.toString(),gr1.calculateGlobalIndex());  
  
gr1 = new GroupStrategy("PA-23");   
  
System.out.printf("\nGrupo %s , GlobalIndex- %f", gr1.toString(),gr1.calculateGlobalIndex());
```

Será que existe Outra Solução melhor?

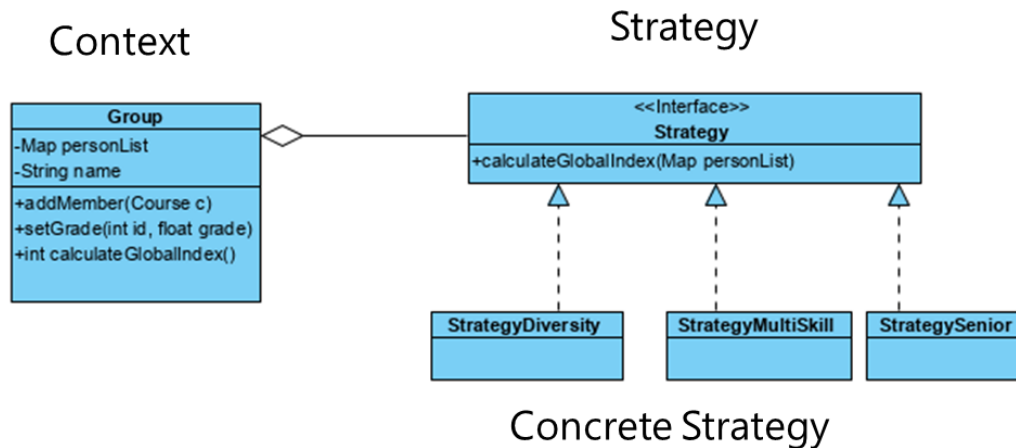
- Sim !! **Aplicar o padrão Strategy**

# Solução : Strategy

**Strategy:** declara uma **interface** comum a todos os algoritmos suportados.

**Context:** usa essa interface para chamar o algoritmo definido por uma estratégia concreta (ConcreteStrategy).

**ConcreteStrategy:** Classes **que** implementam o algoritmo segundo uma estratégia específica.



# Strategy e Concrete Strategy

```
public interface Strategy {  
  
    public float calculateGlobalIndex(Map<Integer, Programmer> personList);  
  
}
```

```
public class StrategyDiversity implements Strategy {  
    @Override  
    public float calculateGlobalIndex(Map<Integer, Programmer> personList){  
        int countYoung=0, countOld=0;  
        for (Programmer programmer : personList.values()) {  
  
            if(programmer.getYearsOfExperience()>5) countOld++;  
            if(programmer.getYearsOfExperience()<=5) countYoung++;  
        }  
        return countYoung*1.f/countOld;  
    }  
}
```

```
public class StrategySenior implements Strategy {  
    @Override  
    public float calculateGlobalIndex(Map<Integer, Programmer> personList){  
        int countOld=0;  
        for (Programmer programmer : personList.values()) {  
            if(programmer.getYearsOfExperience()>10) countOld++;  
        }  
        return countOld*1.f/personList.size();  
    }  
}
```



# Context

```
public class Group {

    private Strategy strategy;
    private String name;
    private Map<Integer, Programmer> personList;

    private static Random random = new Random();

    public Group(String name, Strategy strategy) {
        this.name = name;
        this.strategy=strategy;
        this.personList = new HashMap<>();
    }
    public void setStrategy(Strategy strategy) {
        this.strategy = strategy;
    }
    public float calculateGlobalIndex() {
        return strategy.calculateGlobalIndex(personList);
    }
}
```

1. Define um atributo do tipo **strategy** e é inicializado no construtor.
2. O perfil do grupo (estratégia) pode ser alterado
3. O método **delega** na **estratégia instanciada** o calculo do índice global

# Client – (Main)

```
public class MainStrategy {  
  
    public static void main(String[] args) {  
  
        Programmer p1= new Programmer(1, "Ana",5,2);  
        Programmer p2= new Programmer(2, "Rui",15,8);  
        Programmer p3= new Programmer(3, "Paula",22,9);  
        Programmer p4= new Programmer(4, "Luis",5,6);  
        Group gr1 = new Group("PA-23", new StrategyDiversity());  
        gr1.addMember(p1,p2,p3,p4);  
        System.out.printf("\nGrupo %s , GlobalIndex- %f", gr1.toString(),gr1.calculateGlobalIndex());  
        gr1.setStrategy(new StrategyMultiSkill());  
        System.out.printf("\nGrupo %s , GlobalIndex- %f", gr1.toString(),gr1.calculateGlobalIndex());  
        gr1.setStrategy(new StrategySenior());  
        System.out.printf("\nGrupo %s , GlobalIndex- %f", gr1.toString(),gr1.calculateGlobalIndex());  
  
    }  
}
```

- O grupo altera o seu perfil em tempo de execução

# Strategy| Exercícios(1)



## Crie um projeto a partir do template:

[https://github.com/pa-estsetubal-ips-pt/GroupSelection\\_Template.git](https://github.com/pa-estsetubal-ips-pt/GroupSelection_Template.git)

e analise o código disponibilizado, com as soluções completas apresentadas ao longo destes slides.

- Adicione um novo perfil : SPECIALIZED, em o índice é calculado da seguinte formula:

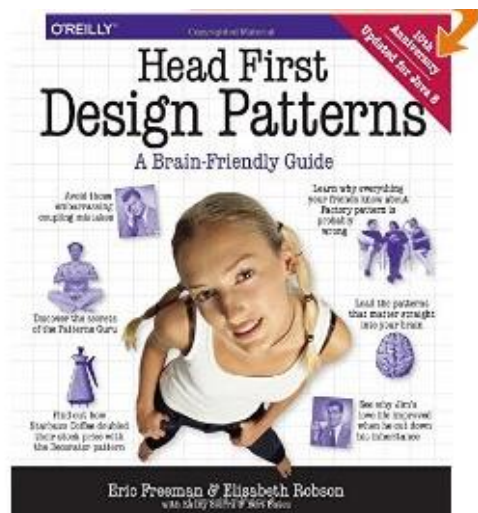
$$X / (Y - X)$$

X número de pessoas com mais de 5 anos de experiência e especializadas no máximo em 3 linguagens

Y número total de pessoas

- Adicione um novo método, para seleção do chefe do grupo que também difere consoante o perfil pretendido
  - SENIOR - o membro com mais anos
  - DIVERSITY - Random
  - MULTISKILLS - o membro com mais linguagens de programação, em caso de empate o mais novo
  - SPECIALIZED - dos membros com mais de 5 anos de experiencia, o que domine menos linguagens de programação. Implemente o código necessário para disponibilizar o método `Programmer selectLeader(Map<Integer,Programmer> personList)` que devolve o membro que será Lider

# Rever



Paginas: 10 - 24

- <https://refactoring.guru/design-patterns/strategy>
- <http://www.dofactory.com/net/strategy-design-pattern>