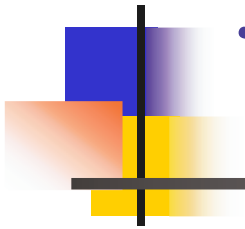
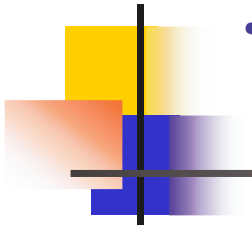


Tipos de Dados Abstractos

Algoritmos e Estuturas de Dados

Inverno 2006

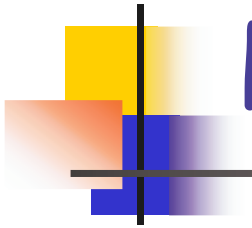
Cátia Vaz



Tipos Abstractos

- Um **tipo abstracto** é:
  - um tipo genérico de dados, dos quais não se conhece os valores
  - uma *interface* que define os acessos e as operações sobre os dados
  - O conjunto das *propriedades* algébricas da interface, que delimitam as possíveis implementações.

- Um programa que usa um tipo abstracto é um ***cliente***. O cliente não tem acesso à implementação.
- Um programa que especifique o tipo de dados é uma **implementação**.



Pilha - *Stack*

- Operações abstractas:
  - inicialização;
  - colocação de um elemento na pilha;
  - remoção de um elemento (o último colocado); LIFO
  - indicar se pilha está vazia.

- **interface para uma pilha** que contenha inteiros:

```
public interface intStack{
    public int empty();
    public void push(int i);
    public int pop();
}
```

Usando a mesma interface, podem ser definidas várias implementações!

Implementação do tipo pilha usando um *array*

```
public class intStackArray implements intStack{
    private int[] s;
    private int n;
    public intStackArray(int capacity) {
          s = new int[capacity];
    }
    public boolean isEmpty(){
          return n == 0;
          }
    public void push(int item) {
          if(n<s.length-1){
          s[n++] = item;}
    } //assume-se que antes de invocar o método pop é testado se a pilha está vazia
    public int pop() {
          int item = s[n-1];
          s[n-1] = 0;n--;
          return item;
    }}
```

Vantagens: simplicidade.
Desvantagens: tamanho máximo limitado à partida.

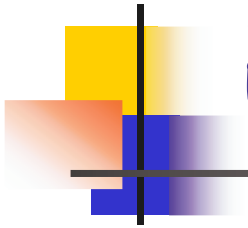
Implementação do tipo pilha usando um **lista ligada**

```
public class intStackList implements intStack{
    private Node head;
    private class Node {
          private int item;
          private Node next;
          private Node(int item){
              this.item=item; }
    }
    public boolean isEmpty() {return head == null; }

    public void push(int item) {
          Node novo = new Node(item);
          novo.next=head;
          head =novo;}
//assume-se que antes de invocar o método pop é testado se a pilha está vazia
    public int pop() {
          int item = head.item;
          head = head.next;
          return item;
    }}
```

Vantagens:
- aumenta e diminui consoante se inserem ou removem novos elementos na pilha.

Desvantagens:
- ocupa mais memória.
- acesso mais lento.

Cátia Vaz



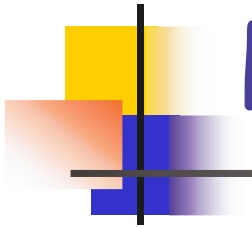
Exercício

- Implemente uma aplicação que leia do standard input uma expressão aritmética em notação posfixa e calcule o valor da expressão. Considere que:
  - a expressão apenas envolve os operadores binários + e x;
  - os operadores e operandos são delimitados por um espaço.
  - Exemplo:
    - 5 9 8 + 4 6 * * 7 + *

    é equivalente a:

    (5 * (((9 + 8) * (4 * 6)) + 7))

    isto é, o seu valor é 2075



Pilha - *Stack*

- Operações abstractas:
  - inicialização;
  - colocação de um elemento na pilha;
  - remoção de um elemento (o último colocado); LIFO
  - indicar se pilha está vazia.

- **interface para uma pilha** genérica:

```
public interface GenStack<Item>{
    public Item empty();
    public void push(Item i);
    public Item pop();
}
```

Usando a mesma interface, podem ser definidas várias implementações!

Pilha genérica: implementação usando uma lista ligada

```
public class Stack<Item> implements GenStack<Item>{
    private Node head;
    private class Node {
          private Item item;
          private Node next;
          private Node(Item item){this.item=item; }
    }
    public boolean isEmpty() { return head == null; }

    public void push(Item item) {
          Node novo = new Node(item);
          novo.next=head;
          head =novo;}
//assume-se que antes de invocar o método pop é testado se a pilha está vazia
    public Item pop() {
          Item item = head.item;
          head = head.next;
          return item;}
}
```

Cátia Vaz

Pilha genérica: implementação usando um *array*

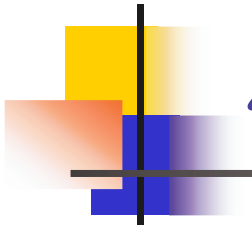
```
//Não compila!!
public class ArrayStack<Item> {
    private Item[] a;
    private int N;
    public ArrayStack(int capacity) {
          a = new Item[capacity];
    }
          //! a criação de um array genérico não é permitida em Java

    public boolean isEmpty() { return N == 0; }
    public void push(Item item) {
          if(n<s.length-1){
          a[n++] = item;}
    }
    //assume-se que antes de invocar o método pop é testado se a pilha está
    vazia
    public Item pop() {
          return a[--N];
    }
}
```

Cátia Vaz

Pilha genérica: implementação usando um *array*

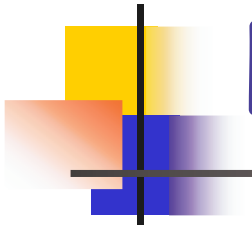
```
public class ArrayStack<Item> {
    private Object[] a;  //solução!
    private int N;

    public ArrayStack(int capacity) {
          a = new Object[capacity];  //solução!
    }
    public boolean isEmpty(){
          return N == 0; }

    public void push(Item item) {
          if(n<s.length-1){
          a[n++] = item;}
    }
//assume-se que antes de invocar o método pop é testado se a pilha está vazia
    public Item pop() {
          return (Item) a[--N];  //solução!
    }
}
```

Cátia Vaz



Autoboxing

- Implementação da pilha genérica:
  - Perite objectos, não tipos primitivos.
- Tipo *Wrapper:*
  - Cada tipo primitivo tem um tipo de objecto *wrapper*;!.
  - Ex: **`Integer`** é o tipo *wrapper* para **`int`**.
- Autoboxing: conversão automática entre um tipo primitivo e o seu *wrapper*.



Fila-*Queue*

- Operações abstractas:
  - inicialização;
  - colocação de um elemento na fila;
  - remoção de um elemento (FIFO);
  - indicar se fila está vazia.

- **interface para uma fila** que contenha inteiros:

```
public interface intQueue{
    public int empty();
    public void put(int i);
    public int get();
}
```

Usando a mesma interface, podem ser definidas várias implementações!

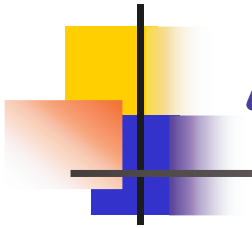
Implementação do tipo fila usando um *array*

```
public class intQueueArray implements intQueue{
   private int[] q;
   private int nLivres,head,tail,maxN;

   public intQueueArray(int maxN){
        q=new int[maxN];
        nLivres=maxN; head=0; tail=0;this.maxN=maxN;}

   public boolean empty(){
        return (head%maxN==tail);}

   public void put(int i){
        if(nLivres!=0){
                q[tail++]=i; tail=tail%maxN;nLivres--;}
   }
//assume-se que antes de invocar o método get é testado se a fila está vazia
   public int get(){
        head=head%maxN;nLivres++; return q[head++];}
}
```

Cátia Vaz

Implementação do tipo fila usando uma lista ligada

Exercício!



ADTs

- ADTs permitem programação modular:
  - separam o programa em módulos mais pequenos;
  - clientes diferentes podem partilhar a mesma ADT.
- ADTs permitem o encapsulamento:
  - mantêm-se os módulos independentes;
  - podem-se substituir as várias classes que implementam a mesma interface, sem alterar o cliente.
- Aspectos do desenho de uma ADT:
  - especificação formal do problema;
  - a implementação tem tendência a tornar-se obsoleta.