

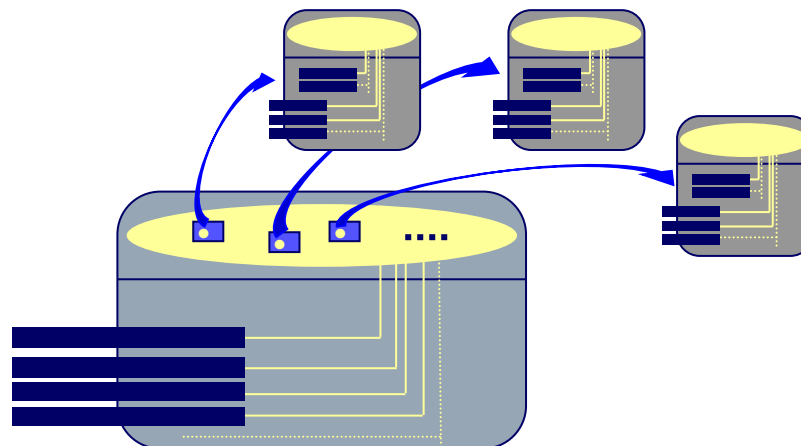
PG II

Programação Orientada aos Objectos em Java

Implementação de Contentores:

- Introdução
- Estruturas de Dados
- Definição de uma colecção genérica
- Implementação de um Stack (pilha)
- Herança
- Implementação de um StackInt
- Diagrama final
- Redefinição de métodos

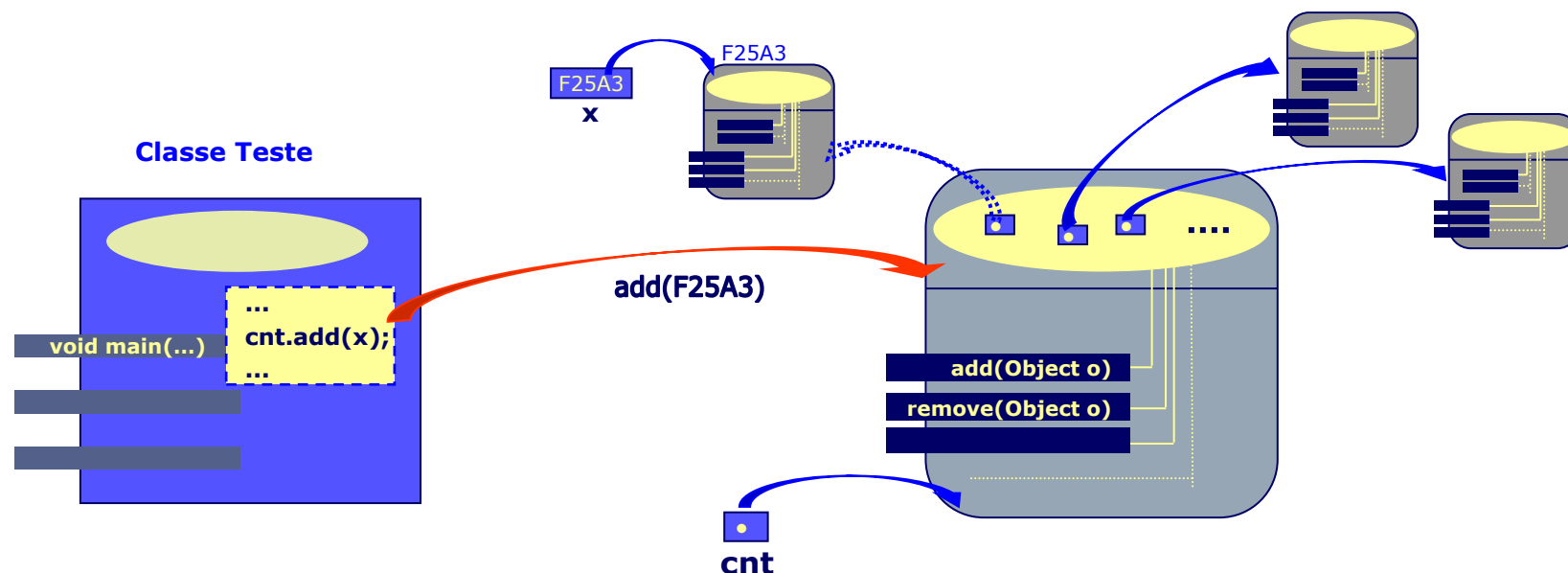
- É vulgarmente designado de **contentor** (ou **coleção**), um objecto que agrupa múltiplos **elementos** na sua **estrutura**.
- Os elementos armazenados são objectos, pelo que, aquilo que é efectivamente guardado num contentor são as referências para esses objectos.



- Principais diferenças funcionais para um *array*:
 - Não têm uma dimensão fixa, ou seja, crescem em função do número de elementos armazenados;
 - Não armazenam directamente elementos de tipos primitivos.

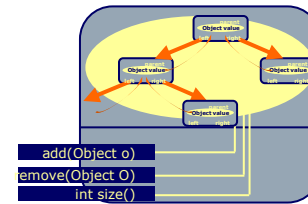
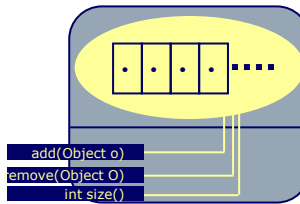
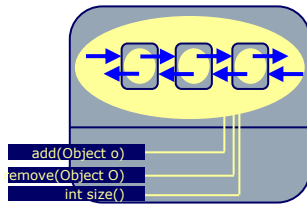
... Introdução

- Nas implementações de contentores é usual a estrutura estar “escondida” do exterior, sendo os seus elementos manipulados através de um conjunto de métodos específicos para esse efeito.



- A implementação interna dos métodos disponibilizados, dão o comportamento específico a cada contentor.
- Os contentores distinguem-se entre si pelo tipo de **restrições/regras** que impõem na forma como armazenam o seus elementos; exemplo: aceitar ou não elementos repetidos, manter os elementos ordenados, aceder aos elementos através do índice (como num array), etc.

- A **estrutura de dados** onde ficam alojados os elementos dentro do contentor vai ser um dos temas de estudo deste capítulo.

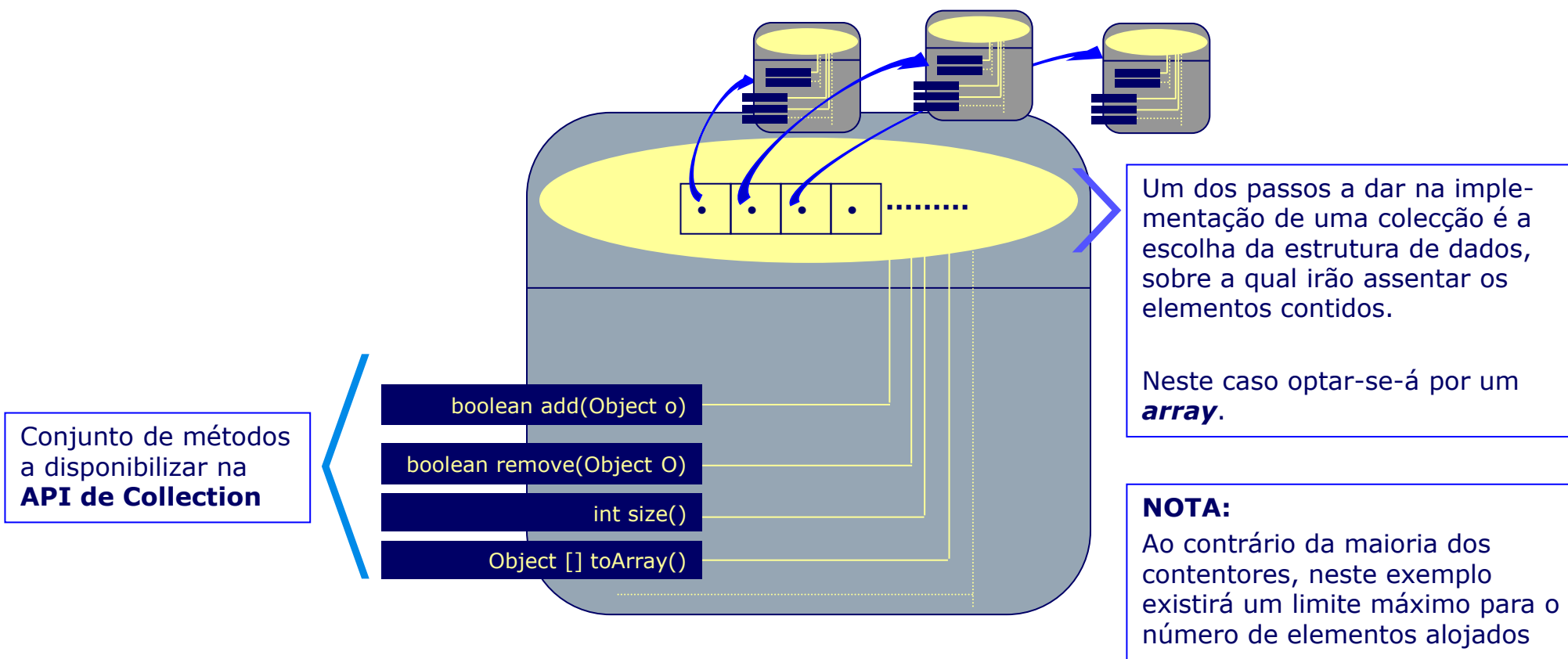


- A compreensão detalhada do funcionamento dos contentores é de importância relevante, na medida em que a maioria das aplicações que se constroem, assentam sobre este tipo de objectos.
- Além disto, o conhecimento da implementação dos contentores é uma forma de transmitir um conjunto de novos conceitos de OOP, tais como **Interfaces** e **Polimorfismo**.
- Os **algoritmos** inerentes à implementação de contentores bem como os conceitos de OOP que estão adjacentes, são fundamentais para consolidar o **conhecimento de OOP**.
- Estes conceitos e ainda a implementação das **estruturas internas** dos contentores, sobre as quais assentam os elementos contidos, fazem parte deste importante tema de estudo da Programação, que são as **Estruturas de Dados**.

Definição de uma colecção genérica

Seja então considerada uma colecção genérica, que obedeça às seguintes regras:

- Os elementos não apresentam ordenação;
- Os elementos podem ser repetidos;
- Pode ter elementos a *null*.



... Definição de uma colecção genérica

```
package aula05;
public class Collection {

    private static final int DEFAULT = 100;
    private Object [] elements;
    private int size=0;

    public Collection(int dim) {elements = new Object[dim];}
    public Collection() {this(DEFAULT);}

    public int size(){return size;}
    public Object [] toArray(){
        Object [] a = new Object[size];
        for(int i=0; i<size; i++) a[i] = elements[i];
        return a;
    }

    public boolean add (Object o){
        if (size() == elements.length) return false;
        System.arraycopy(elements,0,elements,1,size());
        elements[0] = o;
        size++;
        return true;
    }

    public boolean remove(Object o) {
        if (o == null) return false;
        for (int i=0; i<size(); i++)
            if (o.equals(elements[i])) {
                elements[i] = null;
                System.arraycopy(elements, i+1, elements, i, size()-i-1);
                elements[--size] = null;
                return true;
            }
        return false;
    }
}
```

Tamanho *default*, do contentor quando instanciado sem indicação de um tamanho específico.

Estrutura de dados do contentor

Contabiliza o número de elementos alojado no contentor

Constrói um contentor vazio com capacidade para o número de elementos passado no parâmetro dim.

Constrói um contentor vazio com capacidade para 100 elementos.

Retorna o número de elementos armazenados no contentor.

Retorna um array com os elementos alojados neste contentor.

Adiciona o elemento especificado ao contentor. Retorna true se o parâmetro for adicionado com sucesso, e false se o contentor estiver cheio.

Remove do contentor o elemento referido como parâmetro. Retorna false se o parâmetro for null ou não existir no contentor. Retorna true se remover o elemento com sucesso.

... Definição de uma colecção genérica

Além dos métodos específicos desta colecção deverão ainda ser **redefinidos** os seguintes métodos:

```
/**
 * Devolve uma representação em String deste contentor, contendo a
 * representação em String de cada um dos seus elementos.
 * @return Representação em String deste contentor.
 */
public String toString () {
    String var_str = "[";
    for(int i=0; i<elements.length; i++)
        if(elements[i] != null)
            var_str += "(" + elements[i].toString() + ")";
        else
            var_str += "()";
    var_str += "]";
    return var_str;
}

/**
 * Compara a igualdade entre o objecto parametro e este contentor.
 * Devolve true, se e só se, o objecto <code>o</code> for uma <code>
 * Collection</code>, se os contentores tiverem o mesmo número de
 * elementos e se todos eles forem iguais.
 * @param o Objecto a ser comparado com este contentor.
 * @return true se o objecto parâmetro for igual a este contentor.
 */
public boolean equals(Object o){
    Collection c = (Collection) o;
    for (int i=0; i<elements.length; i++)
        if (!(elements[i].equals(c.elements[i])))
            return false;
    return true;
}
```

Class Collection - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address D:\work\aula05\Collection.html Go

toString

```
public java.lang.String toString()
```

Devolve uma representação em String deste contentor, contendo a representação em String de cada um dos seus elementos.

Overrides:

toString in class java.lang.Object

Returns:

Representação em String deste contentor.

equals

```
public boolean equals(java.lang.Object o)
```

Compara a igualdade entre o objecto parametro e este contentor. Devolve true, se e só se, o objecto o for uma Collection, se os contentores tiverem o mesmo número de elementos e se todos eles forem iguais.

Overrides:

equals in class java.lang.Object

Parameters:

o - Objecto a ser comparado com este contentor.

Returns:

true se o objecto parâmetro for igual a este contentor.

... Definição de uma colecção genérica

Teste à classe `Collection`:

```
package aula05;
import pg2.io.IO;

public class Alunos {
    public static void main (String args[]){
        Collection stock = new Collection(args.length);
        IO.cout.writeln("\nContentor vazio = " + stock.toString() + "\n");

        //Testa inserção
        IO.cout.writeln("Testa a inserir:");
        for (int i=0; i<args.length; i++) {
            if (!stock.add(args[i])) break;
        }
        IO.cout.writeln("Contentor cheio = " + stock.toString() + "\n");

        //Testa remoção
        IO.cout.writeln("\nTesta a remover:");
        int r;
        while (stock.size() != 0){
            r = (int) (Math.random()*args.length);
            boolean res = stock.remove(args[r]);
            IO.cout.writeln("Remove o " + args[r] + " = " +
                           res + " : " + stock.toString());
        }
    }
}
```

```
Command Prompt

D:\work>java aula05.Alunos 26838 25855 27586 27552

Contentor vazio = [<><><><>]

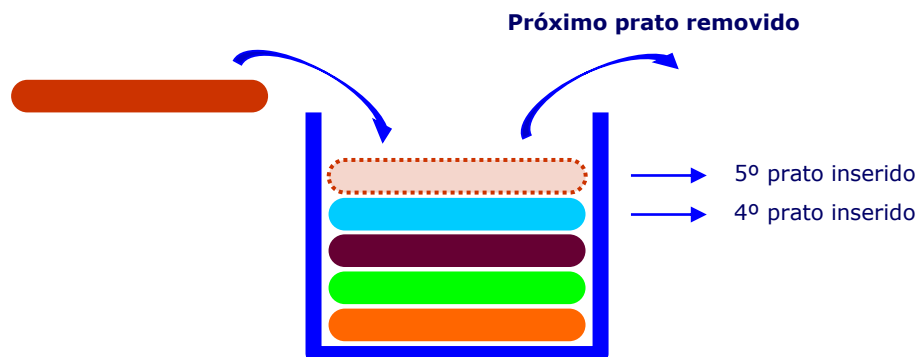
Testa a inserir:
Contentor cheio = [<27552><27586><25855><26838>]

Testa a remover:
Remove o 27586 = true : [<27552><25855><26838><>]
Remove o 27586 = false : [<27552><25855><26838><>]
Remove o 27552 = true : [<25855><26838><><>]
Remove o 27552 = false : [<25855><26838><><>]
Remove o 27552 = false : [<25855><26838><><>]
Remove o 25855 = true : [<26838><><><>]
Remove o 25855 = false : [<26838><><><>]
Remove o 27552 = false : [<26838><><><>]
Remove o 26838 = true : [<><><><>]

D:\work>_
```

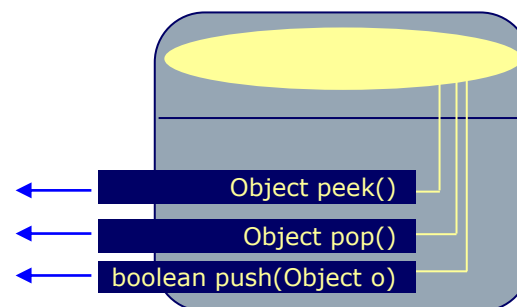

Implementação de um Stack (pilha)

Um contentor do tipo pilha armazena os seus elementos pela ordem de inserção e remove-os pela ordem inversa. Quer isto dizer que os elementos vão-se sobrepondo à medida que são inseridos, ficando sempre no topo da pilha o próximo elemento a ser removido. Seja considerada a pilha apresentada no exemplo seguinte:



Um contentor que implementa este algoritmo, designado de LIFO (*Last in first out*), deve disponibilizar na sua interface o seguinte conjunto de métodos:

Devolve uma referência para o objecto que está no topo da pilha sem o remover.
Remove o objecto que está no topo da pilha e retorna-o como valor deste método.
Adiciona o objecto passado como parâmetro no topo da pilha.



... Implementação de um Stack (pilha) - Herança

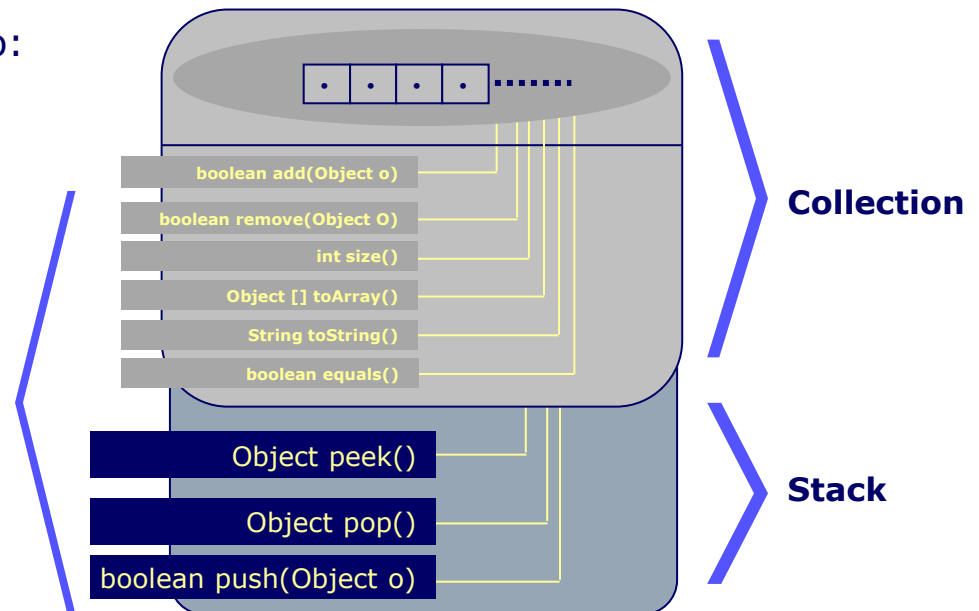
Qual a melhor forma de se implementar esta pilha?

- A **Classe Collection**, já tem a **estrutura** e parte do **comportamento**, que se pretende implementado numa pilha;
- A pilha tem um funcionamento mais específico que uma Collection, ou de outra forma, pode ser vista como uma **especialização** da Classe Collection;
- O ideal seria que de alguma forma a **Classe Stack baseasse a sua implementação na Classe Collection, estendendo** a sua estrutura e comportamento.

Isso é possível através da seguinte declaração:

```
public class Stack extends Collection {
    ...
}
```

A API da classe Stack vai ser
= API de Collection + peek() + pop() + push()



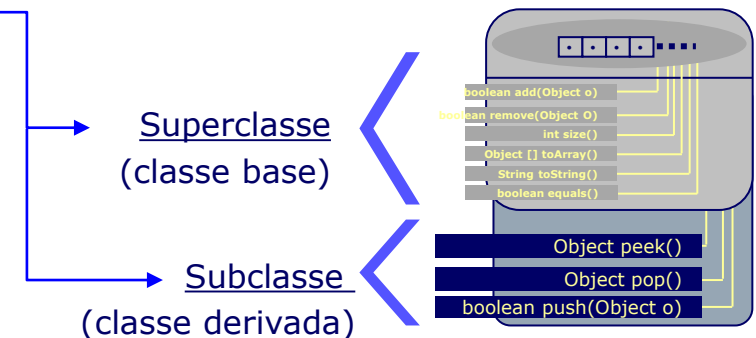
... Implementação de um Stack (pilha) - Herança

Este mecanismo designa-se de **Herança** (ou derivação) e tem as seguintes características:

- A classe **stack** tem **acesso** a todas as variáveis e métodos de instância que **não** sejam declarados como **private** na Classe Collection;
- Na classe **stack** podemos definir novas variáveis e métodos, que serão adicionados à estrutura e comportamento herdado;
- A classe **stack** pode redefinir as variáveis e métodos herdados (mediante algumas regras).
- Neste mecanismo são usadas as seguintes designações:

Seguindo esta arquitectura, para a implementação da classe **stack** apenas terão que ser definidos os seguintes métodos:

```
public class Stack extends Collection {
    public Object peek() {...}
    public Object pop() {...}
    public Object push() {...}
}
```



Ao serem implementado estes métodos coloca-se de imediato o seguinte problema:

- Se as variáveis **private** não estão acessíveis como é que pode ser manipulada na classe **stack** a estrutura de dados, mais concretamente o **array elements**?
- Por um lado é pretendido que a estrutura de dados esteja escondida do utilizador, não devendo por isso ser **public**, por outro esta deve estar acessível à classe derivada.

A forma de concretizar estes dois objectivos é declarando a estrutura da classe **Collection** como **protected**:

```
public class Collection {
    protected Object [] elements;
    protected int size=0;
    ...
}
```

... Implementação de um Stack (pilha) – peek e pop

Implementação da classe **Stack** com os métodos peek() e pop():

```
package aula05;

/**
 * A classe <code>Stack</code> representa uma pilha de objectos LIFO
 * (last-in-first-out). Esta classe estende <code>Collection</code>
 * com mais 3 operações, que fazem com que este contentor seja
 * tratado como uma pilha.
 */
public class Stack extends Collection {
    /**
     * Devolve uma referência para o objecto que está no topo da
     * pilha sem o remover.
     * @return O objecto no topo da pilha (1º elemento do objecto
     * <code>Collection</code>)
     */
    public Object peek(){
        return elements[0];
    }
    /**
     * Remove o objecto que está no topo da pilha e retorna-o como
     * valor deste método.
     * @return O objecto no topo da pilha (1º elemento do objecto
     * <code>Collection</code>)
     */
    public Object pop(){
        Object aux = elements[0];
        System.arraycopy(elements, 1, elements, 0, size-1);
        elements[--size] = null;
        return aux;
    }
}
```

Class Stack - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address D:\work\aula05\Stack.html Go

aula05

Class Stack

java.lang.Object

|-- [aula05.Collection](#)

 |-- **aula05.Stack**

public class **Stack**
extends [Collection](#)

A classe Stack representa uma pilha de objectos LIFO (last-in-first-out). Esta classe estende Collection com mais 3 operações, que fazem com que este contentor seja tratado como uma pilha.

Fields inherited from class [aula05.Collection](#)

[elements](#), [size](#)

Constructor Summary

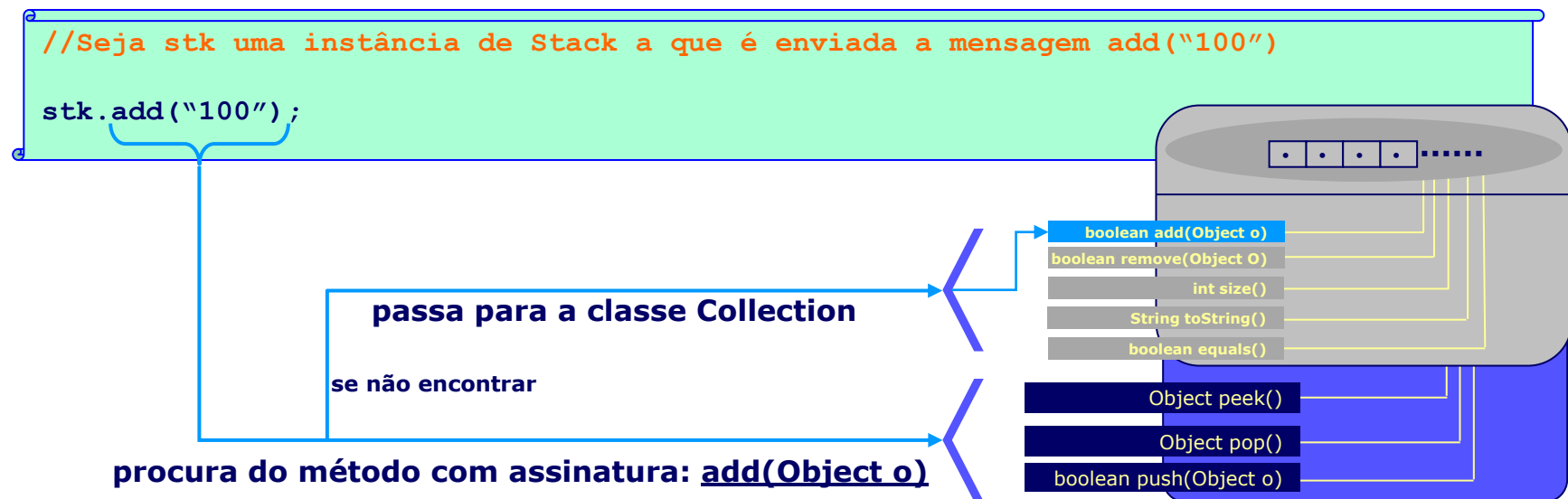
[Stack](#)()

... Implementação de um Stack (pilha)

Ainda sem estar implementado o método `push` em `Stack`, é possível adicionar elementos a uma instância desta classe através do método `add`, herdado de `Collection`.

Seja analisado o mecanismo de pesquisa de um método, que foi invocado a uma instância da classe `Stack` :

1. *Lookup* na API de `Stack`, de um **método com assinatura** igual à **mensagem** enviada;
2. Caso não seja encontrado, então a busca prossegue para a classe hierarquicamente acima, neste caso **`Collection`**, e assim sucessivamente.



... Implementação de um Stack (pilha) - super

O método `push` deverá inserir o novo objecto na posição correspondente à do primeiro elemento a ser devolvido/removido pelo método `peek` ou `pop`. Neste caso os elementos estão a ser retirados da posição 0 do `array` `elements`, pelo que deverão também ser inseridos nesta posição.

A implementação do método `push` é assim a mesma que já existe no método `add` de `Collection`, logo a sua definição será a seguinte:

```
/**
 * Adiciona o objecto passado como parâmetro no topo da pilha. Este método tem
 * exactamente o mesmo comportamento do método: <code>add(Object o)</code>.
 * @param o Objecto a ser colocado no topo desta pilha
 * @return true se o objecto <code>o</code> for inserido com sucesso
 */
public boolean push(Object o){
    return super.add(o);
}
```

push

```
public boolean push(java.lang.Object o)
```

Adiciona o objecto passado como parâmetro no topo da pilha. Este método tem exactamente o mesmo comportamento do método: `add(Object o)`.

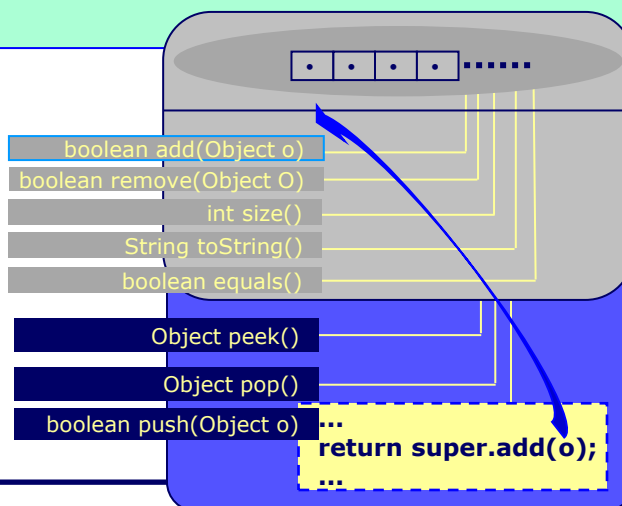
Parameters:

o - Objecto a ser colocado no topo desta pilha

Returns:

true Se o objecto o for inserido com sucesso

referência **super**



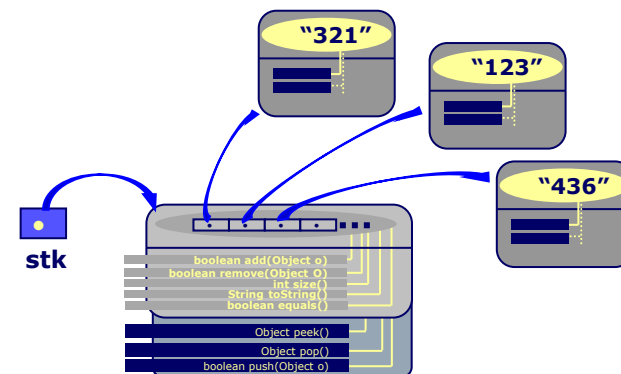
... Implementação de um Stack (pilha) - Construtor

Quais são os construtores da classe `Stack`?

- Tal como em qualquer classe, caso não sejam implementados construtores, então está implícito o construtor sem parâmetros: `Stack()`.

Neste caso é então possível executar as seguintes instruções:

```
...
Stack stk = new Stack();
stk.add("321");
stk.add("123");
stk.add("436");
...
```



Para que este resultado seja possível, então `elements` foi iniciado com uma instância de um array de objectos. Significa que o construtor implícito em `Stack` fez a inicialização da estrutura herdada, ou seja, fez a inicialização que está implementada no construtor de `Collection`.

Isto acontece porque:

- em qualquer construtor que não é invocado **explicitamente** o construtor da classe base (neste caso `Collection`), então é **implícitamente** executado `super()`, construtor sem parâmetros da classe base.

Desta forma é garantida a inicialização da estrutura herdada.

... Implementação de um Stack (pilha)

À semelhança da classe `Collection`, para que seja possível instanciar da mesma forma a classe `Stack`, então deverão ser definidos os dois construtores seguintes:

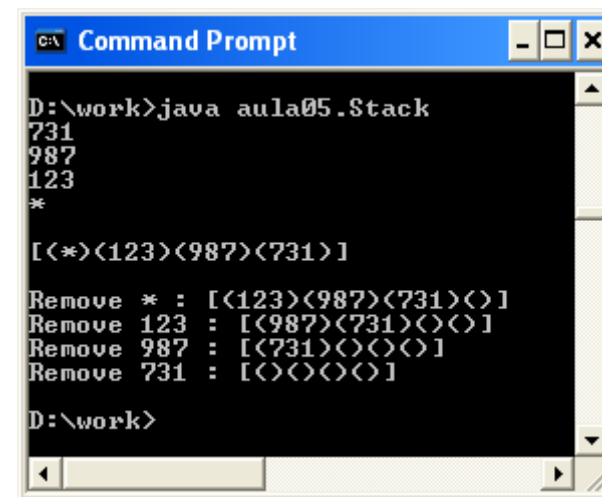
```
public Stack(){} //Está implícita a invocação de super()
public Stack(int dim){super(dim);}
```

Teste à classe `Stack`:

```
public static void main (String argv []) {
    Stack opr = new Stack(4);
    //Testa inserção
    boolean full;
    do {
        full = !opr.push(IO.cin.readLine());
    }
    while (! (opr.peek().equals("+") &&
        ! (opr.peek().equals("*") &&
        !full));

    IO.cout.writeln("\n" + opr.toString() + "\n");

    //Testa remoção
    int i = 0;
    while (opr.size() != 0)
        IO.cout.writeln("Remove " + opr.pop() + " : " + opr);
}
```

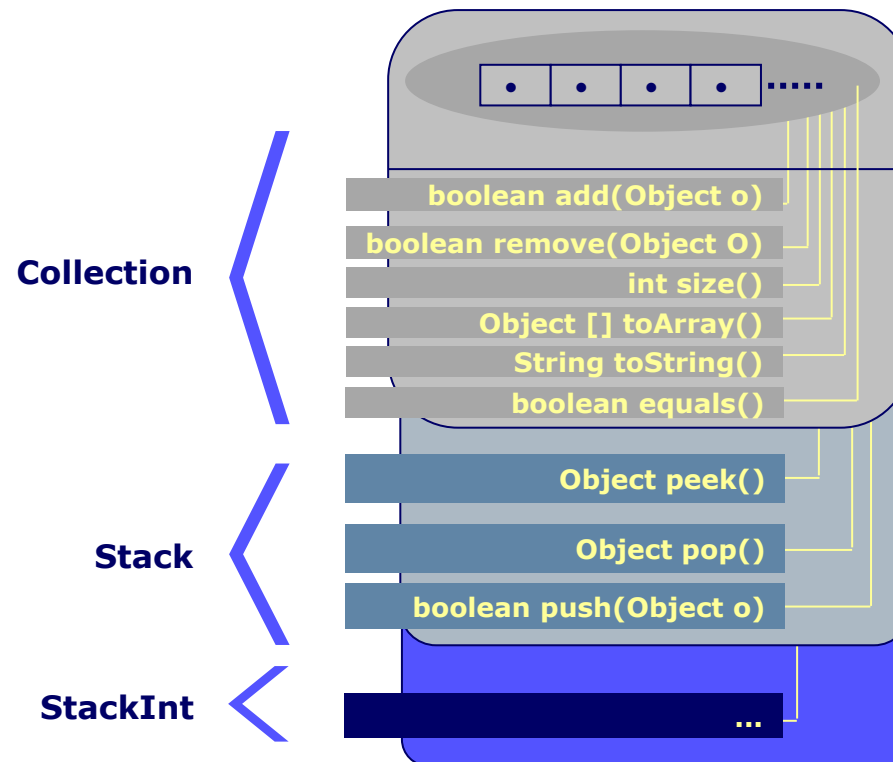


```
C:\ Command Prompt
D:\work>java aula05.Stack
731
987
123
*
[(*)(123)(987)(731)]
Remove * : [(123)(987)(731)()]
Remove 123 : [(987)(731)()]
Remove 987 : [(731)()]
Remove 731 : [()]
D:\work>
```


Mais uma subclasse

Criar mais um nível de subclasse, que seja uma especialização da **Stack** para inteiros: **StackInt**

A herança é **transitiva**, ou seja a **subclasse**, herda da **superclasse** e esta da sua. Assim, **StackInt** vai herdar a estrutura e comportamento (dentro de determinadas restrições) de **Stack**, que por sua vez já tinha herdado de **Collection**.



- A especialização **StackInt**, terá todas as características de uma **Stack**, com a diferença que só aceitará objectos que possam ser representados através de uma instância da classe **Integer**.
- Para implementar esta característica será redefinido (*override*) na classe **StackInt** os métodos **add()** e **push()**;

```
//Métodos de Instancia
public boolean add (Object o){
    if (!(StackInt.isInteger(o.██████████()))) return false;
    return super.add(Integer.valueOf(o.toString()));
}

public boolean push(Object o) {return this.add(o);}
```

- Vamos também ter que implementar um método auxiliar, que nos diga se um determinado objecto pode ou não ser representado através de uma instância da classe Integer.

**Não confundir sobreposição ("overriding")
com sobrecarga ("overloading")**

... StackInt



A definição final da Classe **StackInt** ficará então do seguinte modo:

```
package aula05;

public class StackInt extends Stack {

    //Método de Classe
    private static boolean isInteger(Object o) {
        String str = o.toString();
        if (str == null) return false;
        if (str.length() == 0) return false;
        for (int i = 0; i < str.length(); i++)
            if (!Character.isDigit(str.charAt(i)))
                return false;
        return true;
    }

    //Construtores
    public StackInt() {}
    public StackInt(int dim) {super(dim);}

    //Métodos de Instancia
    public boolean add(Object o) {
        if(!StackInt.isInteger(o)) return false;
        return super.add(Integer.valueOf(o.toString()));
    }

    public boolean push(Object o) {
        return add(o);
    }
}
```

Teste à Classe **StackInt**:

```
...
public static void main(String [] args){
    StackInt opr = new StackInt(10);
    String str;
    //Testa inserção
    do{
        str = IO.cin.readLine();
    }
    while (opr.push(str));
    IO.cout.writeln("\n" + opr.toString() + "\n");

    //Testa remoção
    int i = 0;
    while (opr.size() != 0)
        IO.cout.writeln("Remove " + opr.pop() + " : " + opr);
}
```

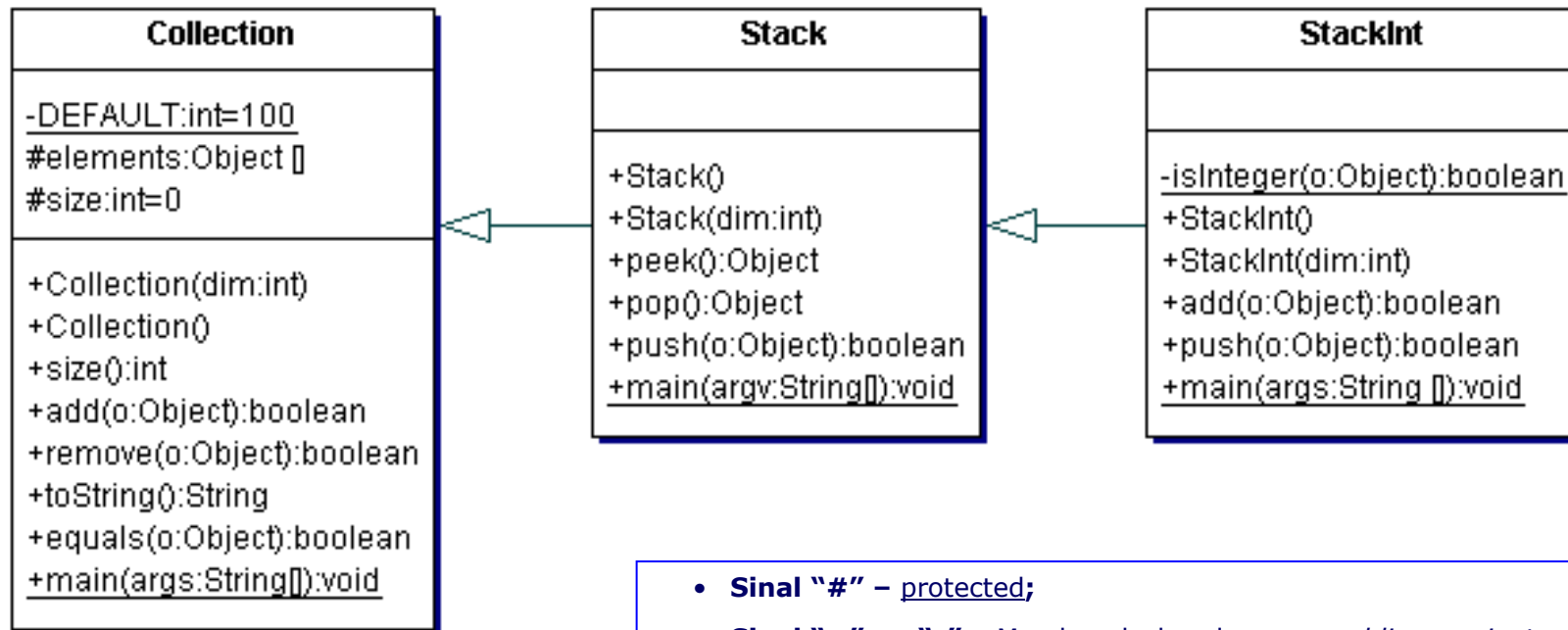
Command Prompt

```
D:\work>java aula05.StackInt
987
465
312
xxx

[<312><465><987><><><><><><>]
Remove 312 : [<465><987><><><><><><><>]
Remove 465 : [<987><><><><><><><><>]
Remove 987 : [<><><><><><><><><>]
D:\work>_
```

Diagrama final

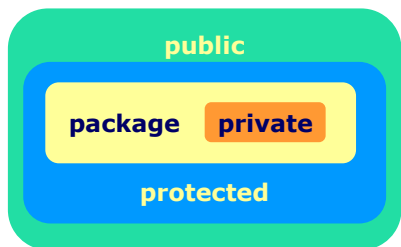
Em UML o símbolo de derivação é uma seta fechada e o seu sentido é da classe derivada para a classe base. Assim, pode ser apresentado como diagrama de classes destes contentores o seguinte esquema:



- **Sinal "#"** – protected;
- **Sinal "+" ou "-"** – Membro declarado como *public* ou *private*, respectivamente;
- **Sublinhado** – Indicação de *static* (de classe);
- **...:<tipo>** - Tipo de uma variável ou tipo de retorno de um método/função.

Redefinição de métodos

Falta introduzir um nível de acesso aos membros de uma classe, que é o **package**. Tal como o nome indica os atributos e métodos declarados com esta permissão, só estão acessíveis em classes definidas no mesmo **package**. Este é o nível de acesso que está implícito quando um atributo ou método é definido sem o prefixo **private**, **public** ou **protected**.



- **public**
- **protected**
- **package** (sem especificação)
- **private**

- sem restrições
- dentro do package e em classes derivadas
- dentro do package
- só dentro da classe

Regras de redefinição:

- Os métodos declarados como **final**, não poderão ser redefinidos pelas subclasses que os herdarem;
- Uma classe declarada como **final**, não poderá ser "estendida" a subclasses;
- A redefinição dos níveis de acesso num método, nunca poderá diminuir o grau de acessibilidade definido na superclasse: