TÉCNICO LISBOA

JAVA INSPECTOR

Grupo 4
Alberto Carvalho, 66933
David Duarte, 68505
José Cavalheiro, 64804

ORGANIZAÇÃO DO CÓDIGO

- **Inspector:** responsável por orquestrar o comportamento das classes para suportar os comandos.

- **Inspected Object:** Classe que armazena um objecto inspeccionado e respectiva classe

ORGANIZAÇÃO DO CÓDIGO

- **InfoPrinter:** responsável pela impressão para o **System.Err** da informação relacionada com a inspecção dos objectos;

- **History Graph:** responsável pela navegação no grafo de objectos inspeccionados;

- **Types:** Enumerado que guarda informação acerca de tipos primitivos, respectivos wrappers e sabe como fazer conversões para tipos primitivos.

1. INSPECTOR

1.1 COMANDOS

INSPECT

i *name* [*value*]

- O **value** é opcional;
- O **value** indica quantos níveis queremos subir na hierarquia de classes para podermos inspeccionar *shadowed fields;*
- No caso padrão, se o field não for encontrado é procurado em cada superclasse.

MODIFY

m *name value*

- Suporta todos os tipos primitivos e qualquer tipo de objectos;

- Procura o field com nome **name**, na classe e nas superclasses;

- Tenta que o valor **value** seja convertido no tipo do field, antes de ser atribuído como valor do mesmo field.

CALL

c *name value$_0$ value$_1$ value$_2$ … value$_n$*

- Suporta todos os tipos primitivos e qualquer tipo de objectos;
- Procura o método mais adequado para ser executado;
  - Filtram-se os métodos;
  - Depois de obtido o método mais compatível, os argumentos a passar ao método são convertidos em função do tipo dos argumentos do método;
  - É feito **invoke**, inspecciona-se o objecto e adiciona-se ao grafo.

CALL

Filtragem dos métodos

- Os métodos são filtrados pelo **nome,** pelo **número** de argumentos e pela **compatibilidade** do tipo dos argumentos do método em relação aos argumentos passados como input.

CALL

Filtragem dos métodos

- **Nome:** m5
- **Argumentos:** "2", "5.2"

Nome diferente

Número de argumentos diferente

```
private void m3(int i);
```

Nome igual

```
private void m5(float f1, float f2);
```

Número de argumentos igual: 2

Ambos os tipos compatíveis

CALL

Filtragem dos métodos

- Atribui uma classificação a cada método e escolhe aquele que tiver valor mais baixo.
  - Int → 1; Float → 2; Double → 3; Etc.
- Exemplo:
  - m1(int, int, int) → classificação: 111
  - m1(int, int, float) → classificação: 112
  - m1(float, int, int) → classificação: 211
  - **⇒ m1(int, int, int) é escolhido**

SAVE E GET

s *name*

- Guarda o objecto que está a ser inspeccionado actualmente, associando-lhe o nome **name.**

g *name*

- Retorna o objecto gravado com o nome **name.**

2. INSPECTED OBJECT

INSPECTED OBJECT

- Necessidade de classificar tipos primitivos como tais e não com o respectivo wrapper;

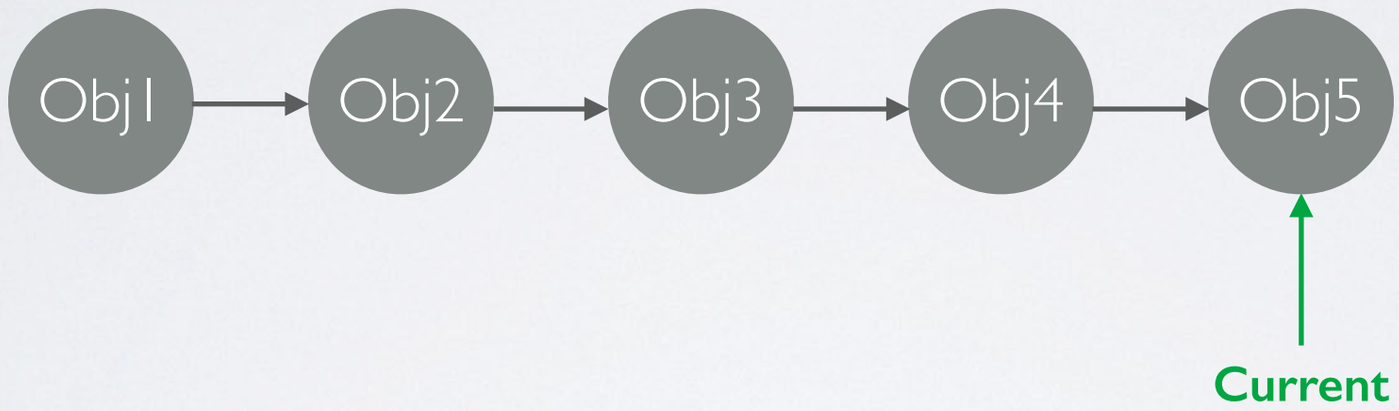
- Fundamental nos casos em que são guardados e recuperados objectos do tipo primitivo;

- O caso particular do comando Call;

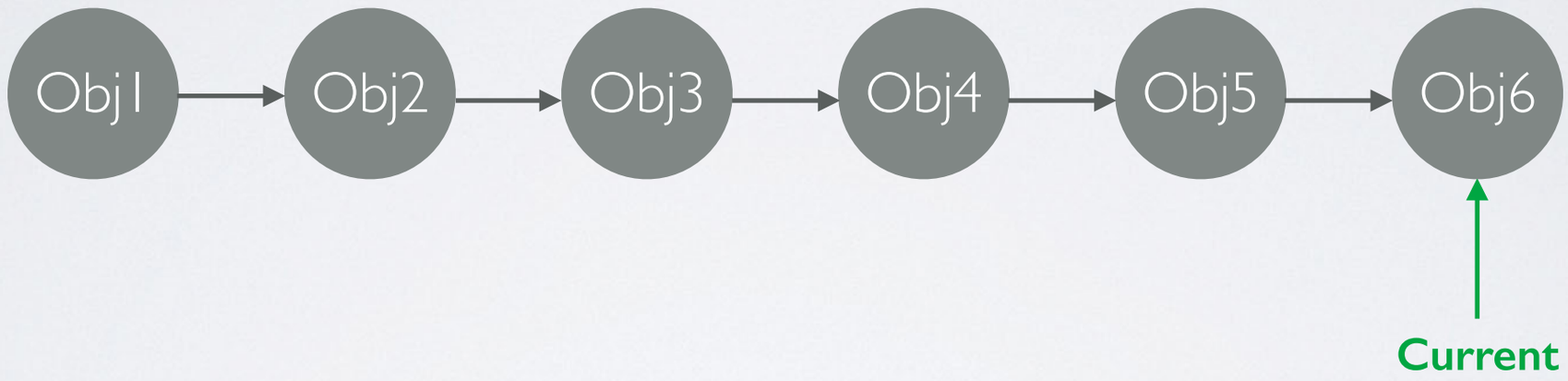
- O caso da classe InfoPrinter.

3. HISTORY GRAPH

HISTORY GRAPH

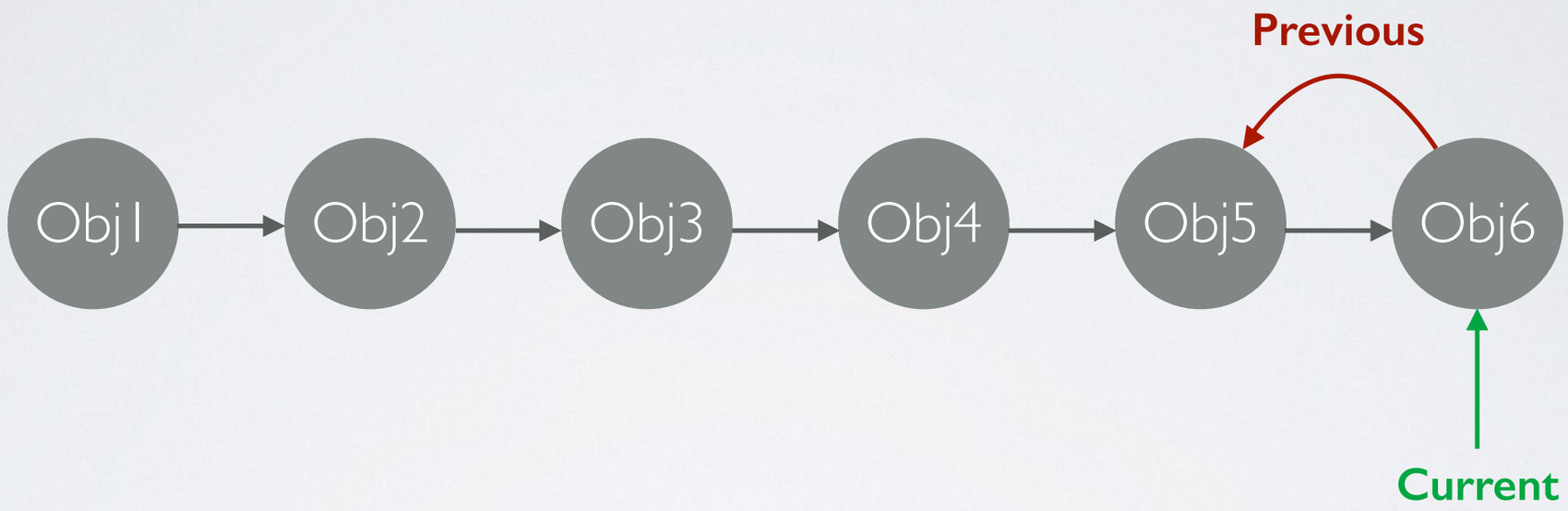


HISTORY GRAPH



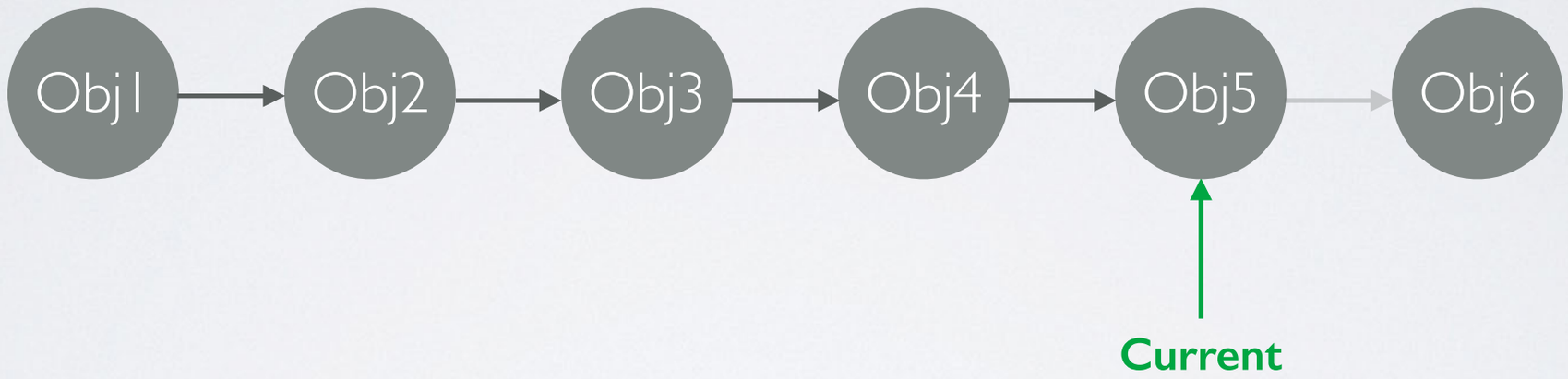

```
> i obj6
>
```

HISTORY GRAPH



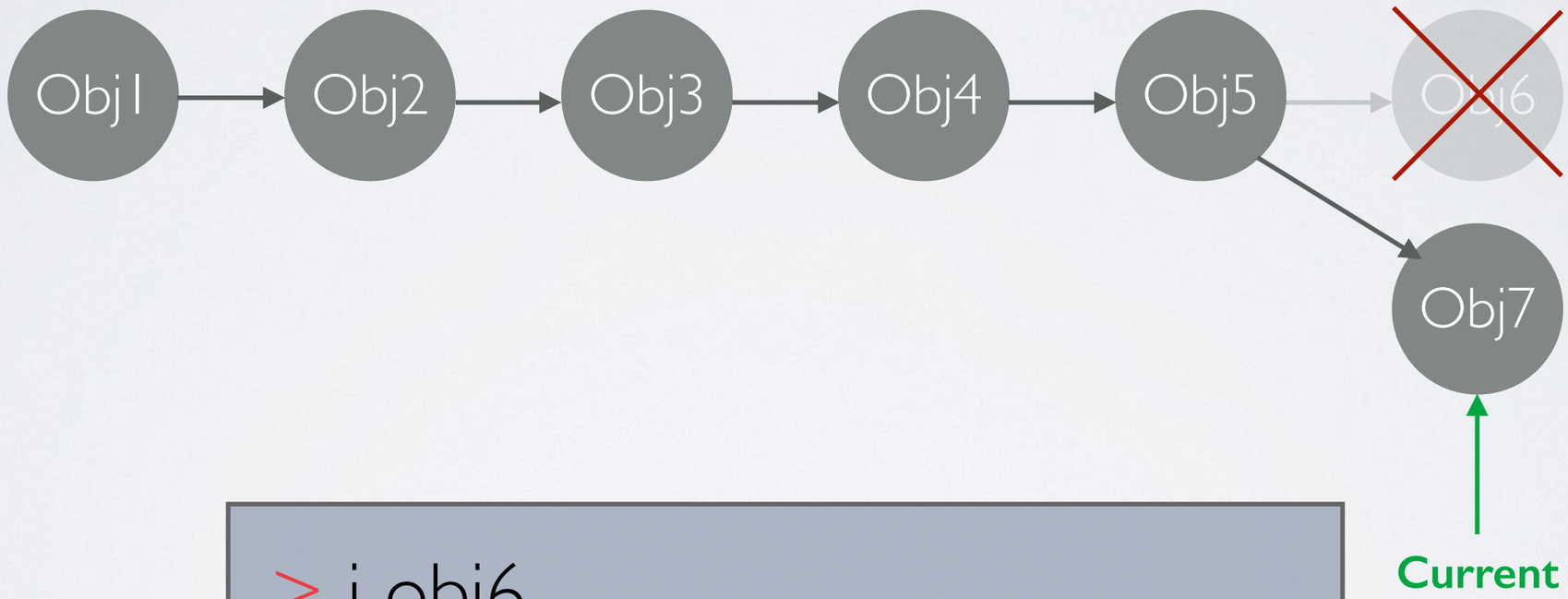

```
> i obj6
> p
>
```

HISTORY GRAPH




```
> i obj6
> p
>
```

HISTORY GRAPH

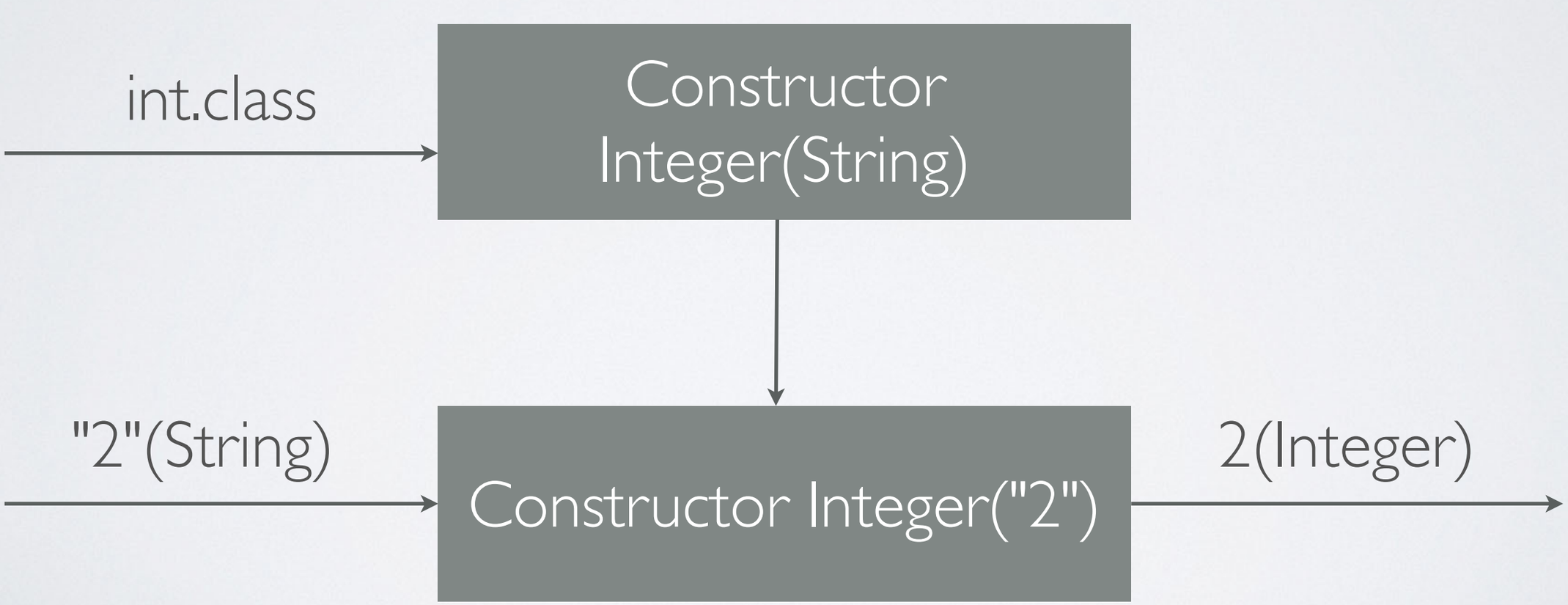



```
> i obj6
> p
> i obj7
```

4.TYPES

TYPES

- Mapeia cada tipo primitivo com o constructor do wrapper respectivo



TYPES

- Solução alternativa:

```
...
if (... == int.class) {
    parseInt
} else if(... == float.class) {
    parseFloat
}
...
```

TYPES

- É responsável por algumas conversões:
  - " ... " → String;
  - '...' → Char;
  - #abc → Objecto guardado com nome abc.

OUTROS ASPECTOS

IF-ELSE ENCADEADOS

- Usando *Reflection* invocamos os métodos respectivos através do método **invoke**.

```
private void i(String[ ] input);
```

- Definimos os nomes dos métodos para corresponderem ao nome dos comandos.

```
this.getClass().getDeclaredMethod("i", String[ ].class).
    invoke(this, new Object[ ] { new String[ ] {"obj6"}});
```

OBJECT INSPECIONADO:

um qualquer tipo primitivo

- Impedimos que se executem sobre estes os comandos i, m e c.

- Mesmo princípio no caso do objecto inspecionado ser null.

MÉTODOS COM ARGUMENTO DO TIPO OBJECT

Problema

- No caso do comando **c equals ...**, o método equals recebe como argumento um object;

- Seguindo a nossa solução descrita até aqui, tentaríamos converter o argumento para… object.

MÉTODOS COM ARGUMENTO DO TIPO OBJECT

Solução

- Tentamos converter a string passada como argumento para cada um dos tipos primitivos, e se ainda assim não der, tentamos usá-la enquanto string.