

Programação Orientada Pelos Objectos

Herança de Interfaces Classes Abstractas

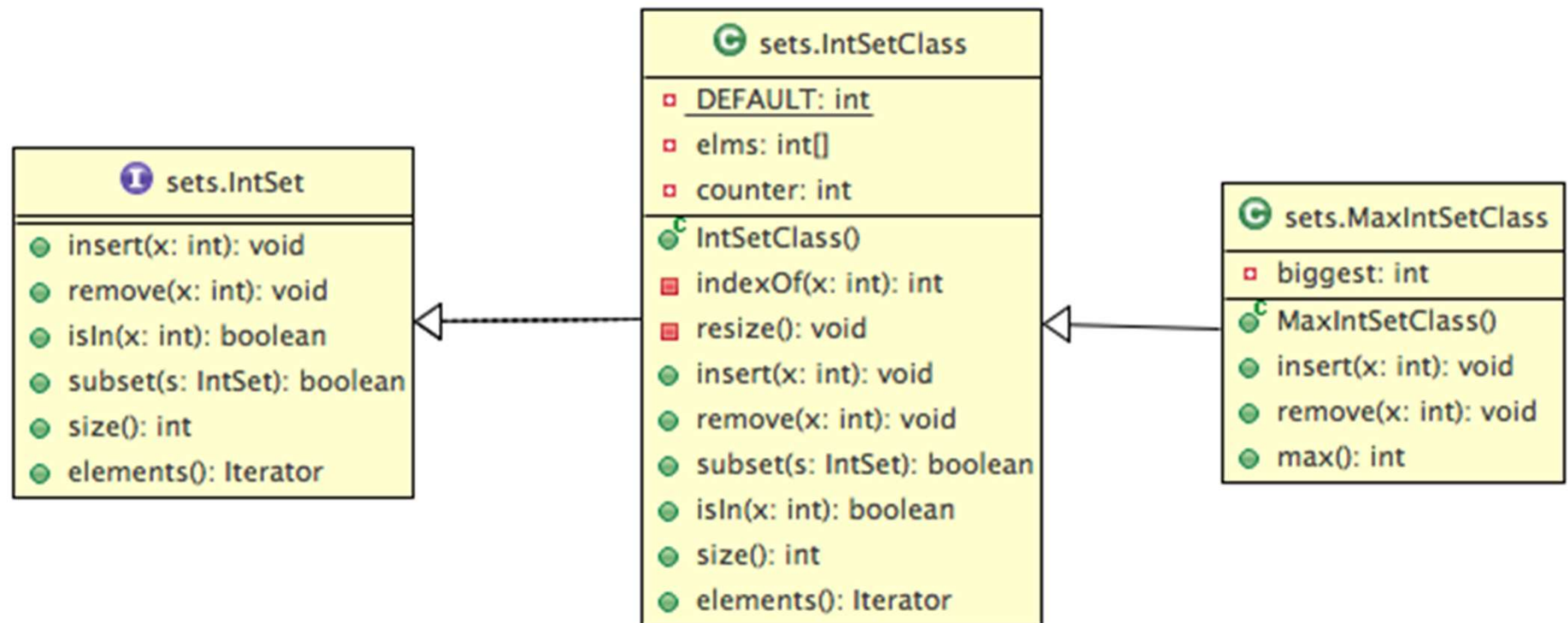


2

Herança de interfaces

Relembrar a hierarquia de tipos...

3



Como usar os métodos das subclasses?

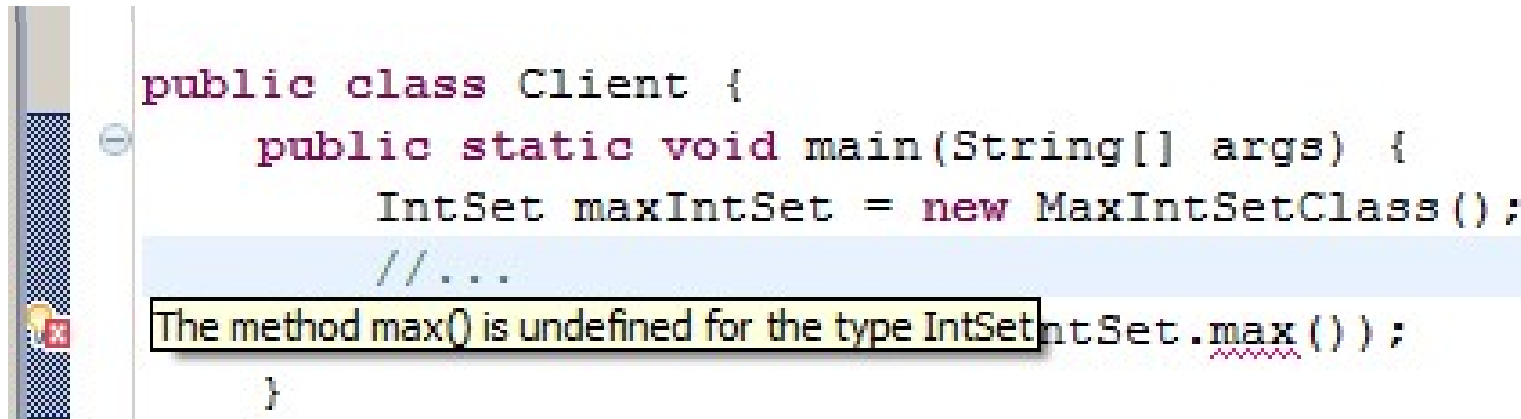
4

- Imaginem que uma nova classe **Client** pretendia usar **MaxIntSetClass...**

Como usar os métodos das subclasses?

5

- Imaginem que uma nova classe **Client** pretendia usar **MaxIntSetClass**...



```
public class Client {  
    public static void main(String[] args) {  
        IntSet maxIntSet = new MaxIntSetClass();  
        //...  
        IntSet.max();  
    }  
}
```

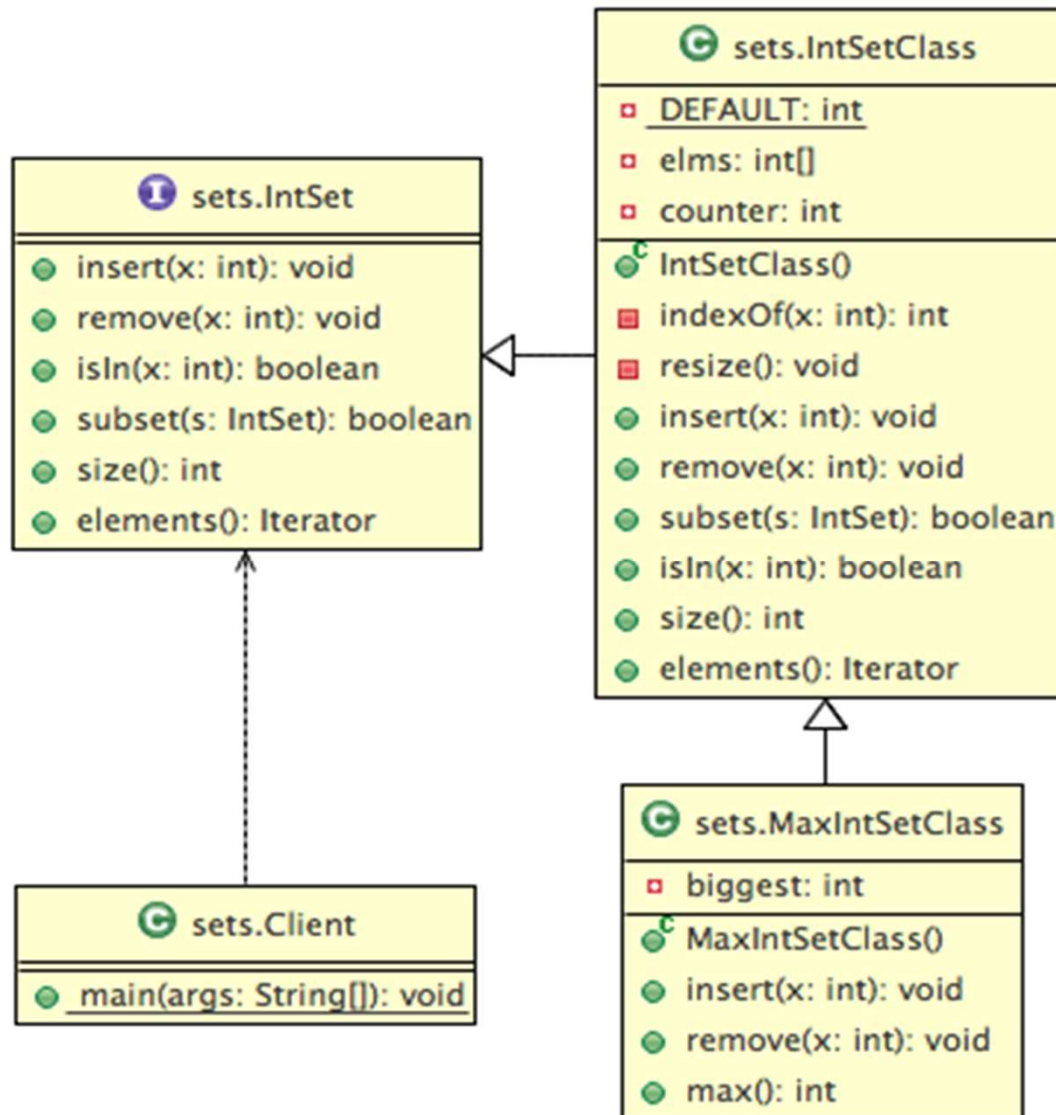
The method max() is undefined for the type IntSet

- Devemos usar a interface na declaração.
 - Mas a interface **IntSet** não declara a nova operação **max**! Ooopsie...

Como usar os métodos das subclasses?

6

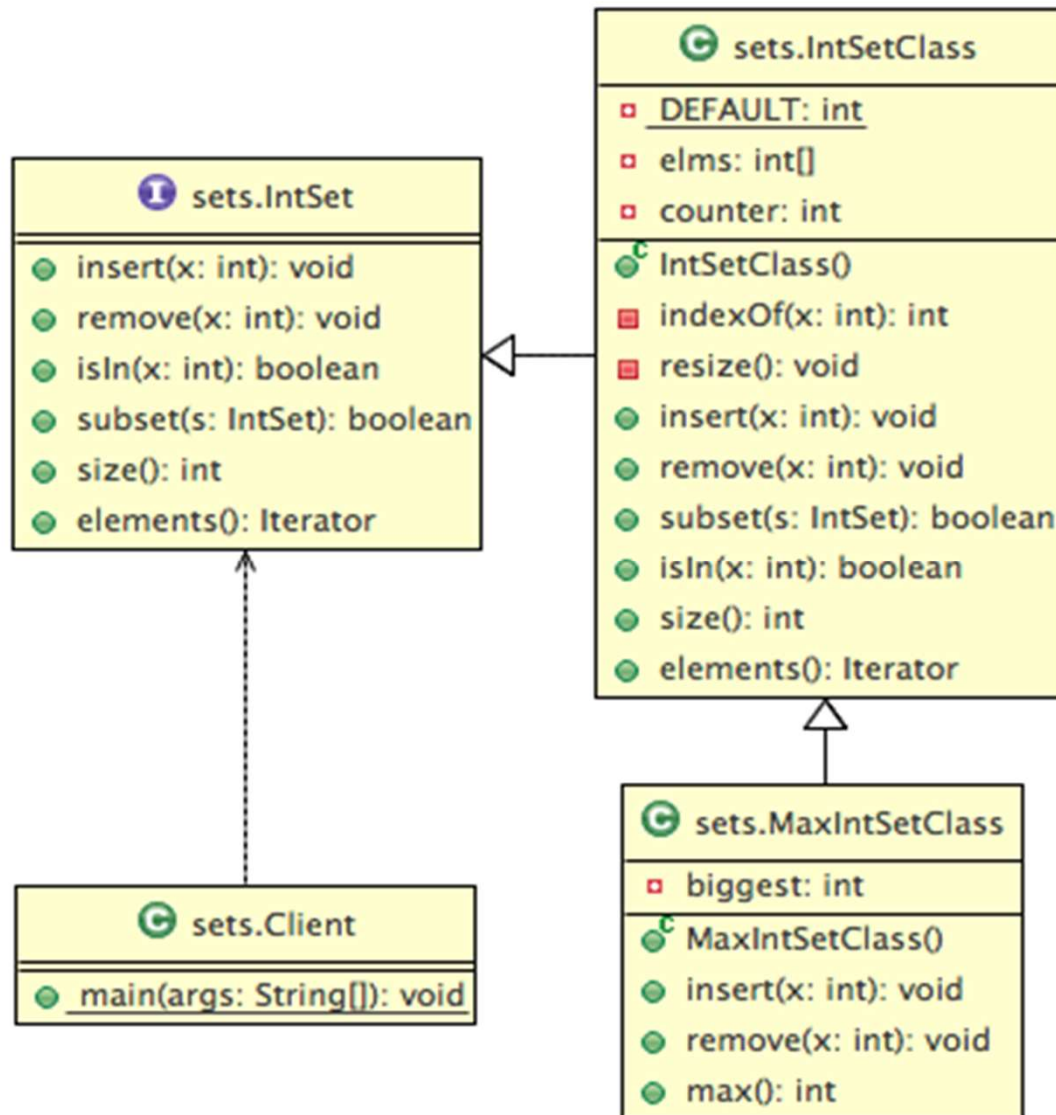
```
IntSet maxIntSet = new MaxIntSetClass();
```



Como usar os métodos das subclasses?

7

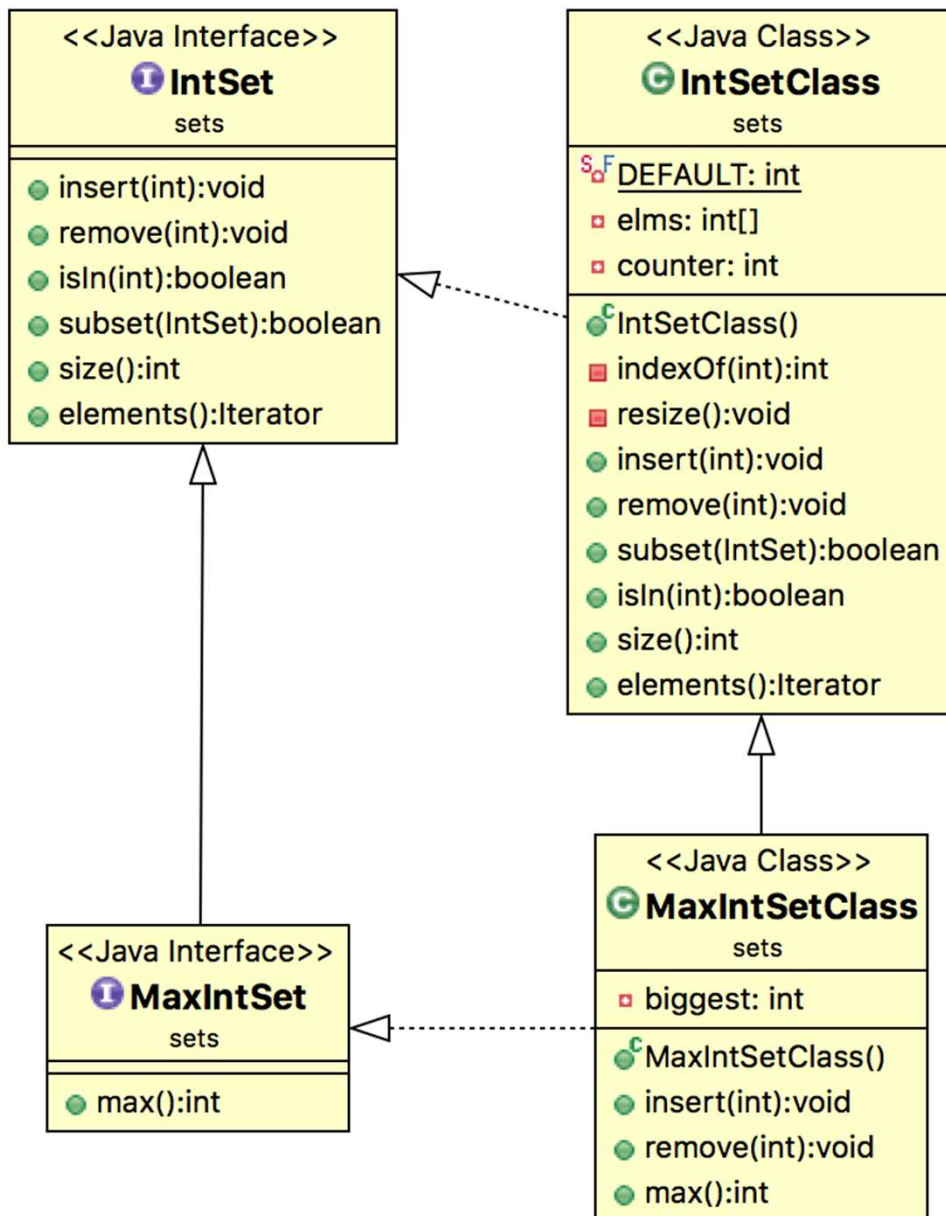
```
IntSet maxIntSet = new MaxIntSetClass();
```



- As interfaces existentes não incluem as novas operações das subclasses
- Para a nova operação devemos definir **novas interfaces**
- Por vezes, as novas interfaces estão **conceptualmente relacionadas** com as já existentes

Herança de interfaces

8



- Para estas situações, criamos uma subinterface da interface original, que declara as novas operações

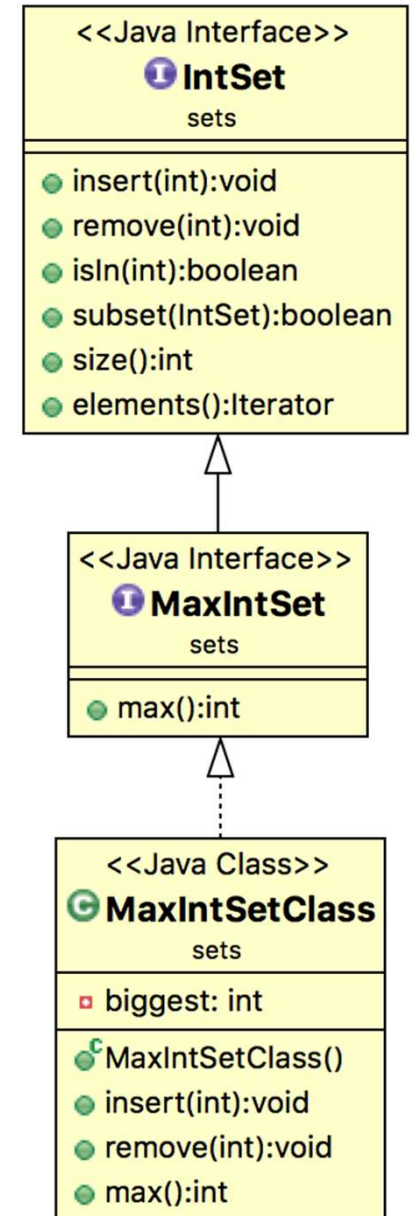
Herança de interfaces

9

```
public interface MaxIntSet extends IntSet {  
    /**  
     * Retorna o máximo da coleção de inteiros.  
     * Assume-se que esta operação só pode ser  
     * chamada se o conjunto não for vazio.  
     * @pre - this.size() > 0  
     * @return - o máximo da coleção.  
     */  
    int max();  
}
```

```
public class MaxIntSetClass extends IntSetClass  
    implements MaxIntSet {  
    public int max() { ... }  
    //...  
}
```

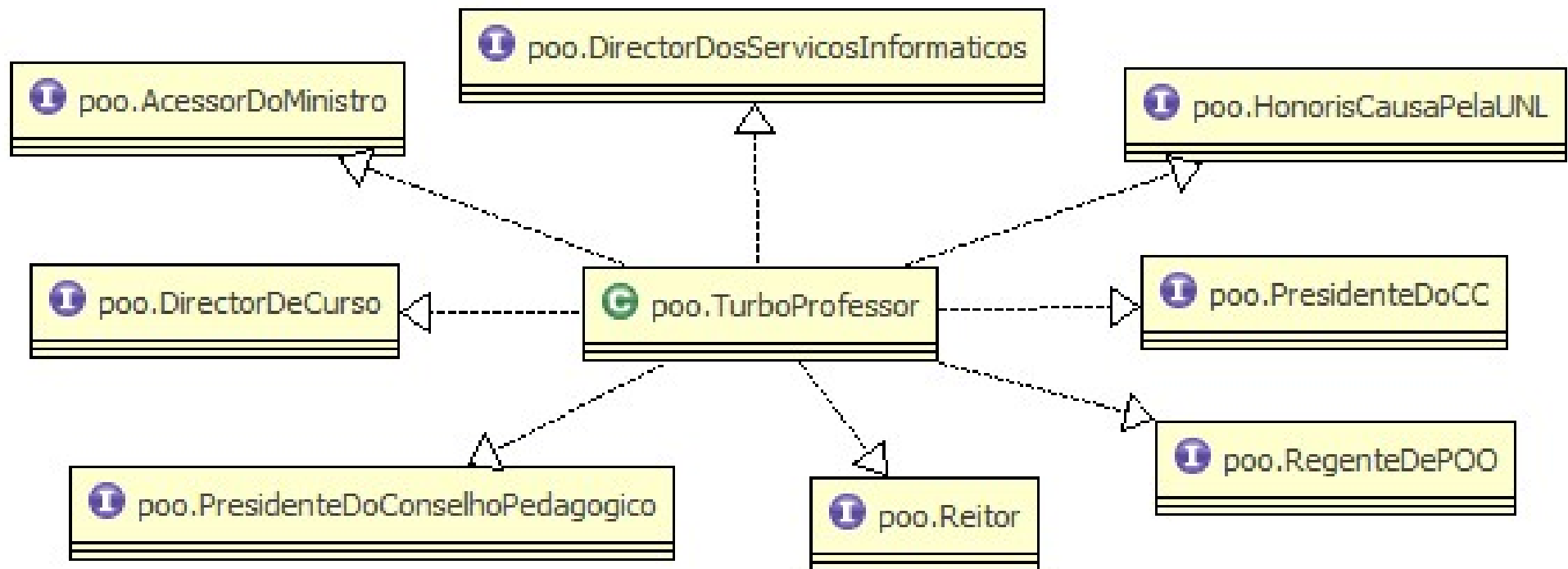
↓
Subclass e
utiliza interface



Implementação de múltiplas interfaces

10

- Uma classe pode implementar qualquer número de interfaces. O Java não estabelece limites.

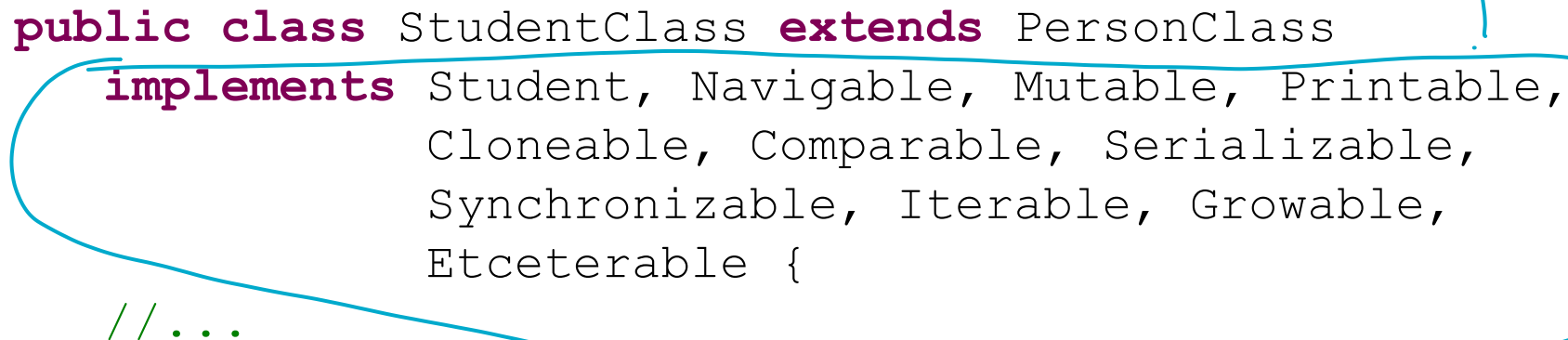


Implementação de múltiplas interfaces

11

- A cláusula **extends** é colocada **antes** da cláusula **implements**.
- As diversas interfaces surgem numa **única cláusula implements**, separadas por vírgulas

```
public class StudentClass extends PersonClass
    implements Student, Navigable, Mutable, Printable,
        Cloneable, Comparable, Serializable,
        Synchronizable, Iterable, Growable,
        Etceterable {
    //...
```



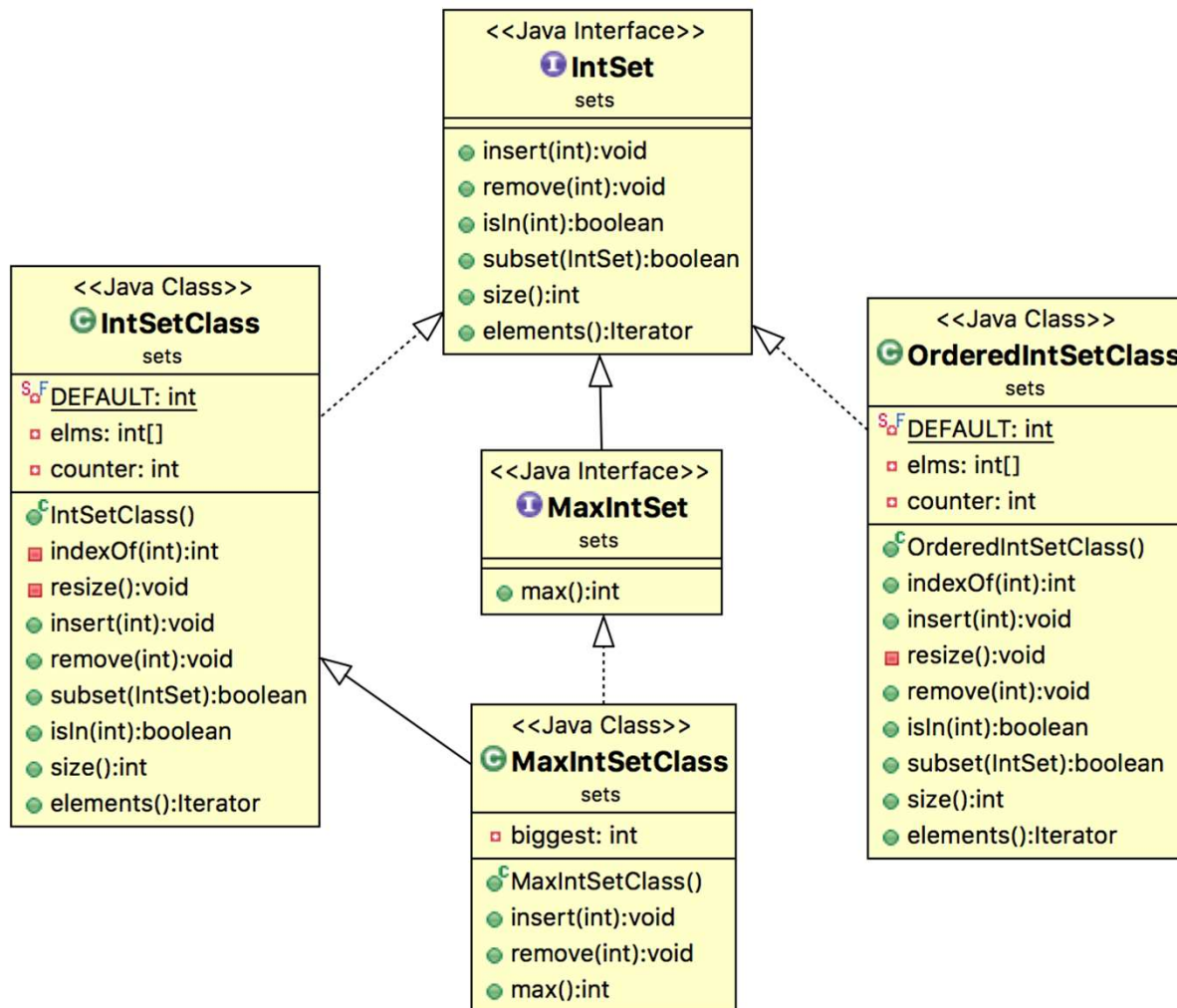
12

Classes abstractas

Repetição de código



13



- Ambas as implementações **IntSetClass** e **OrderedIntSetClass** suportam a interface **IntSet**
 - Mesma lista de constantes
 - Mesma lista de variáveis de instância
 - Mesma lista de operações
 - Algumas delas, com implementações iguais!



IntSetClass vs. OrderedIntSet

14

IntSetClass

```
public class IntSetClass
    implements IntSet {
    private static final int DEFAULT = 10;
    private int[] elms;
    private int counter;

    public IntSetClass() {
        elms = new int[DEFAULT];
        counter = 0;
    }

    //indexOf, insert, remove, isIn diferentes
```

OrderedIntSetClass

```
public class OrderedIntSetClass
    implements IntSet {
    private static final int DEFAULT = 10;
    private int[] elms;
    private int counter;

    public OrderedIntSet() {
        elms = new int[DEFAULT];
        counter = 0;
    }

    // indexOf, insert, remove, isIn diferentes
```

IntSetClass vs. OrderedIntSet

15

IntSetClass

```
public boolean subset(IntSet s) {
    if (s.size() < this.size())
        return false;
    for (int i = 0; i < counter; i++)
        if (!s.isIn(elms[i]))
            return false;
    return true;
}

public int size() {
    return counter;
}

public Iterator elements() {
    return
        new IteratorClass(elms, counter);
}
} // Fim da classe IntSetClass
```

OrderedIntSetClass

```
public boolean subset(IntSet s) {
    if (s.size() < this.size())
        return false;
    for (int i = 0; i < counter; i++)
        if (!s.isIn(elms[i]))
            return false;
    return true;
}

public int size() {
    return counter;
}

public Iterator elements() {
    return
        new IteratorClass(elms, counter);
}
} // Fim da classe OrderedIntSetClass
```

Código repetido? repetido? repetido?



16

- E se descobrirmos um bug?
 - Vamos corrigir o código em **TODAS** as cópias
- E se quisermos simplesmente acrescentar algo novo?
 - Vamos acrescentar a nova funcionalidade em **TODAS** as cópias
- Lei de Murphy:
 - "Whatever can go wrong, will go wrong."

Razões de não fazer código repetido

Factorização de código

17

- Herança permite escrever código **factorizado (sem redundâncias)**
 - Se várias classes têm código **repetido**, provavelmente são implementações de casos particulares de um conceito mais geral
 - Devemos identificar esse conceito e materializá-lo numa classe que concentre o código comum
 - As classes originais passam a incorporar o código factorizado através de herança

Vantagens da factorização

18

- Factorização contribui para a **extensibilidade** do código
 - O código diz-se **extensível** se for possível acrescentar-lhe novas funcionalidades sem ter de alterar o código já existente
- Aumenta o nível de **generalidade** das abstracções usadas
- Torna o código mais compacto e bem organizado
- Facilita a correcção de erros
- Reduz a possibilidade de introdução de incoerências nos programas

Classes e métodos abstractos

19

- Quando factorizamos código podem surgir classes tão gerais que não faz sentido serem directamente instanciadas
 - Tais classes dizem-se **abstractas** e caracterizam-se por:
 - Implementar apenas **parcialmente** um tipo
 - Poder conter **métodos sem corpo**, ou seja **métodos abstractos**
 - Não poder ser instanciadas
- Por vezes usa-se a metáfora “*classes com buracos*”
- Em Java essas classes declaram-se com a palavra reservada **abstract**
- Métodos abstractos declaram-se igualmente com **abstract**

Classes concretas vs. Classes abstractas

20

- Classes **concretas** implementam completamente o tipo
- Classes **abstractas** implementam-no apenas parcialmente
 - Não é possível criar instâncias de uma classe abstracta
 - Os métodos não implementados dizem-se abstractos
 - Cabe às subclasses implementar esses métodos

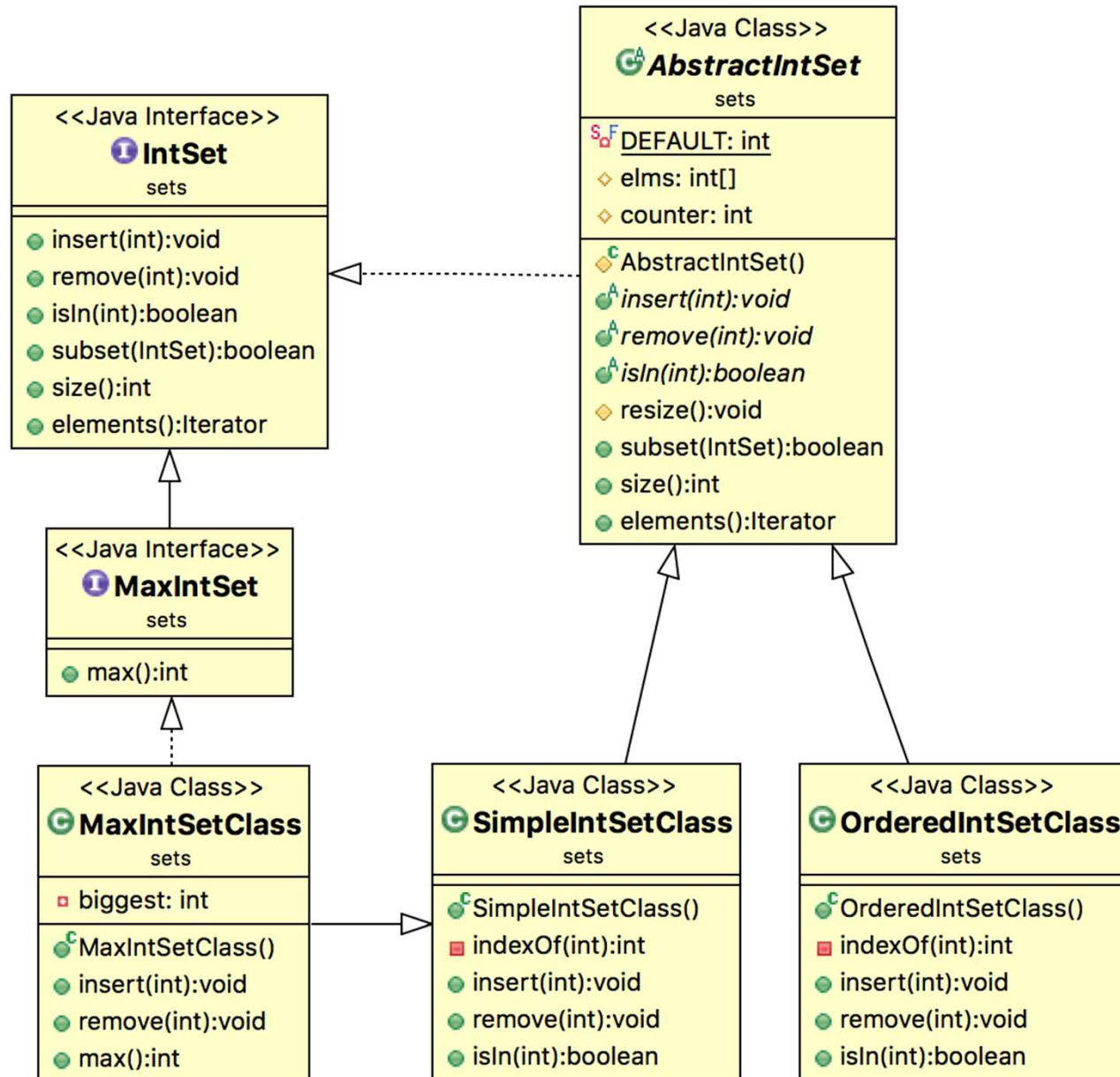
Classes abstractas

21

- Uma classe abstracta pode ter:
 - Variáveis de instância
 - Nesse caso, a classe abstracta deve ter construtores, para inicializar as variáveis de instância
 - Estes construtores **não podem** ser usados para criar objectos – podemos declarar como `protected`
 - São usados pelas subclasses da classe abstracta
 - Métodos abstractos e não abstractos
 - As implementações dos métodos não abstractos frequentemente tiram partido dos métodos abstractos
 - A superclasse define a parte genérica da implementação
 - As subclasses definem os detalhes em falta

Diagrama de classes, agora usando a classe abstracta **AbstractIntSet**

22



A classe AbstractIntSet

23

```
public abstract class AbstractIntSet implements IntSet {  
    /**  
     * Dimensão, por omissão, do vector onde guardamos os  
     * elementos do conjunto.  
     */  
    private static final int DEFAULT = 10;  
  
    /**  
     * Vector acompanhado onde guardamos os elementos do conjunto  
     * de inteiros. Visível nas subclasses.  
     */  
    protected int[] elms;  
  
    /**  
     * Dimensão do conjunto. Protegida, porque todas as  
     * implementações concretas de um conjunto vão ter de  
     * manter esta variável. Visível nas subclasses.  
     */  
    protected int counter;
```


A classe AbstractIntSet

24

```
public abstract class AbstractIntSet implements IntSet {
```

Ao restringir a visibilidade aumentamos o encapsulamento

```
private static final int DEFAULT = 10;
```

```
/**
```

```
 * Vector acompanhado onde guardamos os elementos do conjunto  
 * de inteiros. Visível nas subclasses.
```

```
 */
```

```
protected int[] elms;
```

```
/**
```

```
 * Dimensão do conjunto. Protegida, porque todas as  
 * implementações concretas de um conjunto vão ter de  
 * manter esta variável. Visível nas subclasses.
```

```
 */
```

```
protected int counter;
```

A classe AbstractIntSet

25

```
protected AbstractIntSet() {  
    elms = new int[DEFAULT];  
    counter = 0;  
}  
  
protected void resize() {  
    int[] tmp = new int[elms.length*2];  
    for (int i = 0; i < counter; i++)  
        tmp[i] = elms[i];  
    elms = tmp;  
}  
  
public abstract void insert(int x);  
  
public abstract void remove(int x);  
  
public abstract boolean isIn(int x);
```

Classe com "buracos"

- Compete às subclasses "preencher" os *slots* deixados em aberto
- É possível omitir a declaração dos métodos abstractos que já estejam declarados numa interface que seja implementada por esta classe abstracta (neste caso, na interface IntSet)

A classe AbstractIntSet

26

```
public boolean subset(IntSet s) {
    if (s.size() < this.size()) return false;
    for (int i = 0; i < counter; i++)
        if (!s.isIn(elms[i]))
            return false;
    return true;
}

public int size() {
    return counter;
}

public Iterator elements(){
    return new IteratorClass(elms, counter);
}
} // Fim da classe abstracta AbstractIntSet
```

A classe SimpleIntSetClass

27

```
public class SimpleIntSetClass extends AbstractIntSet {  
    /**  
     * Construtor de <code>SimpleIntSet</code>  
     */  
    public SimpleIntSetClass() {  
        super();  
    }  
  
    private int indexOf(int x) {  
        int i = 0;  
        while (i < counter) {  
            if (elms[i]==x)  
                return i;  
            i++;  
        }  
        return -1;  
    }  
}
```

→ Chama o construtor da superclasse

A classe SimpleIntSetClass

28

```
public void insert(int x) {
    if (counter == elms.length) {
        resize();
    }
    elms[counter++] = x;
}

public void remove(int x) {
    int index = indexOf(x);
    counter--;
    elms[index] = elms[counter];
}

public boolean isIn(int x) {
    return (indexOf(x) != -1);
}
} // Fim da classe SimpleIntSet
```


A classe OrderedIntSetClass

29

```
public class OrderedIntSetClass extends AbstractIntSet {

    public OrderedIntSetClass() {
        super();
    }

    private int indexOf(int n) {
        int low = 0;
        int high = counter-1;
        int mid = -1;
        while (low <= high) {
            mid = (low+high)/2;
            if (elms[mid] == n) return mid;
            else if (n < elms[mid]) high = mid-1;
            else low = mid+1;
        }
        return low;
    }
}
```

A classe OrderedIntSetClass

30

```
public void insert(int x) {  
    int pos = indexOf(x);  
    if (counter == elms.length)  
        resize();  
    for (int i = counter; i > pos; i--)  
        elms[i] = elms[i-1];  
    elms[pos] = x;  
    counter++;  
}
```

A classe OrderedIntSetClass

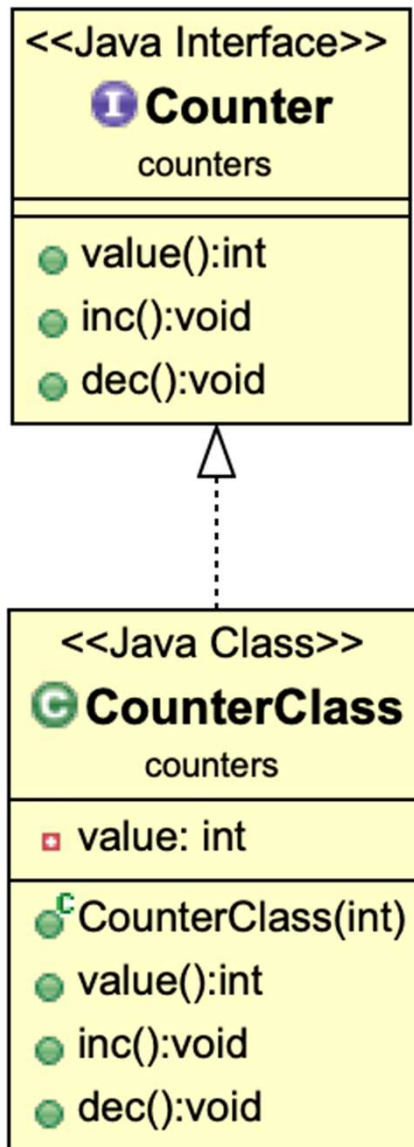
31

```
public void remove(int x) {
    int i = indexOf(x);
    while (i < counter-1) {
        elms[i] = elms[i+1];
        i++;
    }
    counter--;
}

public boolean isIn(int x) {
    int i = indexOf(x);
    if (counter == i)
        return false;
    return elms[i] == x;
}
} // Fim da classe OrderedIntSetClass
```

Exercício sobre Herança

32



- `value()`
 - Devolve o valor do contador;
- `inc()`
 - Incrementa o contador;
- `dec()`
 - Decrementa o contador.

- Contador que pode ser ligado e desligado;
- Similar a um pedómetro.

33

Resumindo...

Factorização

34

Na matemática

- Técnica que consiste em colocar em evidência os factores comuns em expressões algébricas
- Facilita os cálculos
- Ajuda a resolver situações mais complexas

Na programação

- Técnica que consiste em colocar em evidência os membros iguais de uma ou mais classes numa superclasse
- Em vez de as classes repetirem a definição dos membros iguais, centralizamos essa definição na superclasse e obtemo-la na subclasse por herança

A superclasse onde se factoriza o código, em geral, é abstracta

35

- No sentido técnico
 - Implementa parcialmente um tipo
- No sentido lógico
 - Resulta da factorização dos aspectos comuns das subclasses, ignorando os aspectos que não são comuns

A classe abstracta deve representar um conceito “natural”

36

- A classe abstracta não deve corresponder a uma ficção desligada da realidade, pois isso obcureceria a lógica do programa