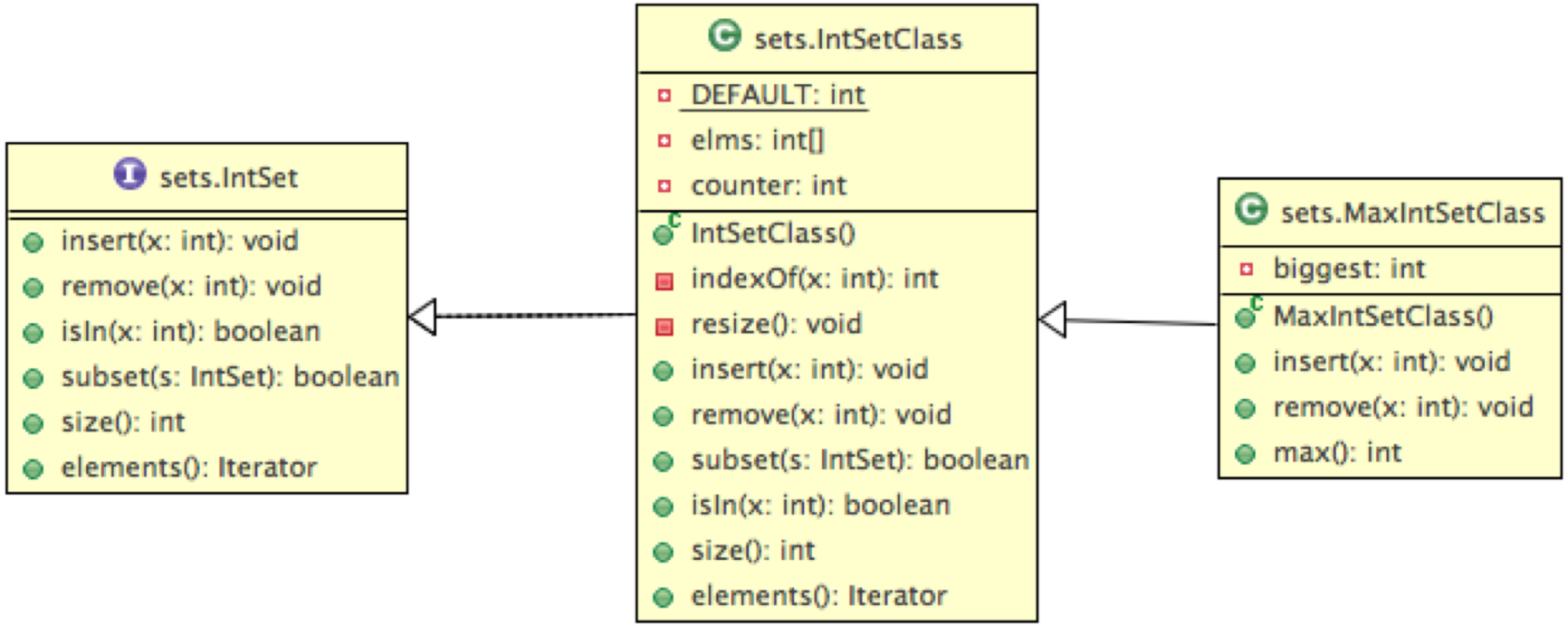
PROGRAMAÇÃO ORIENTADA PELOS OBJECTOS

Herança de Interfaces – Classes Abstractas

2

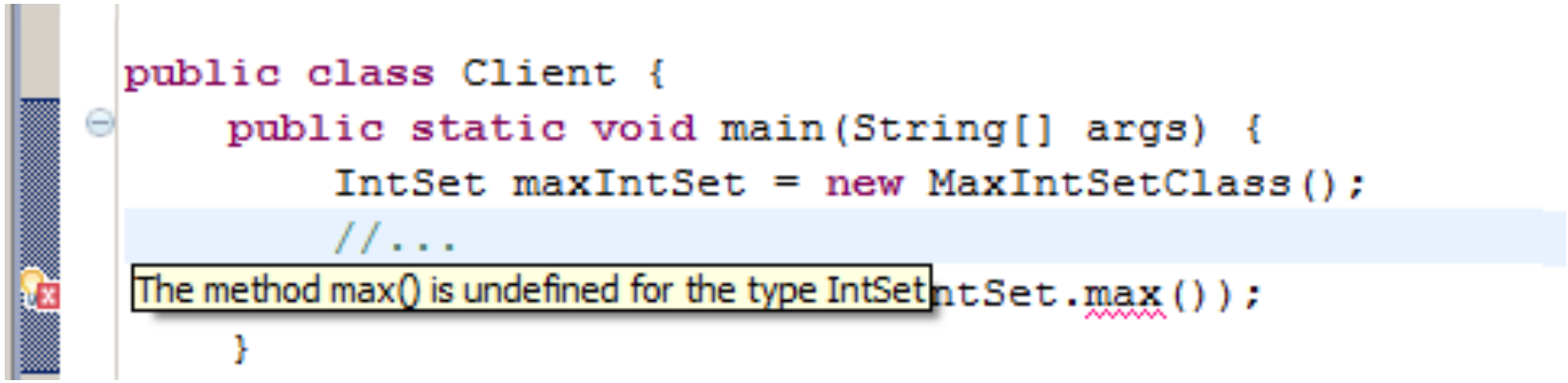
Herança de interfaces

Relembrar a hierarquia de tipos...



Como usar os métodos das sub-classes?

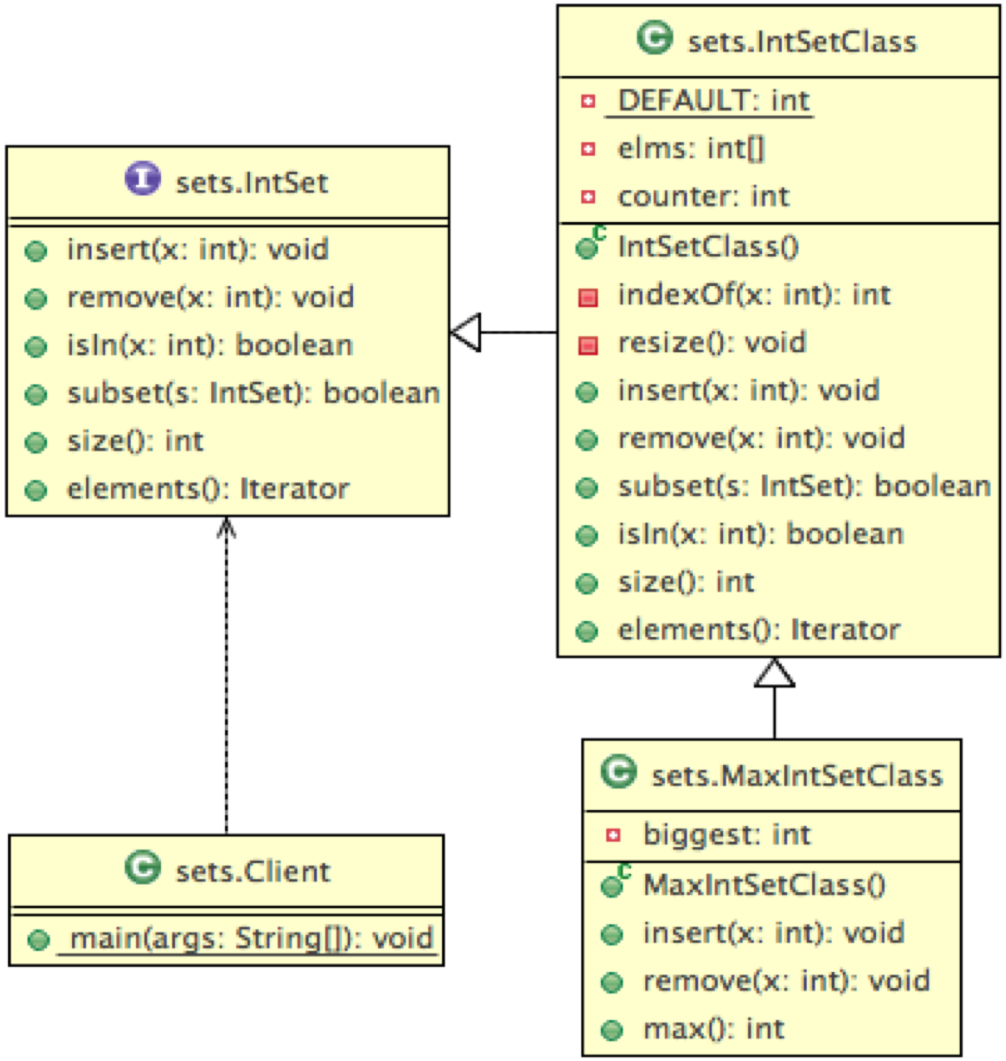
- Imaginem que uma nova classe **`Client`** pretendia usar **`MaxIntSetClass`**...



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

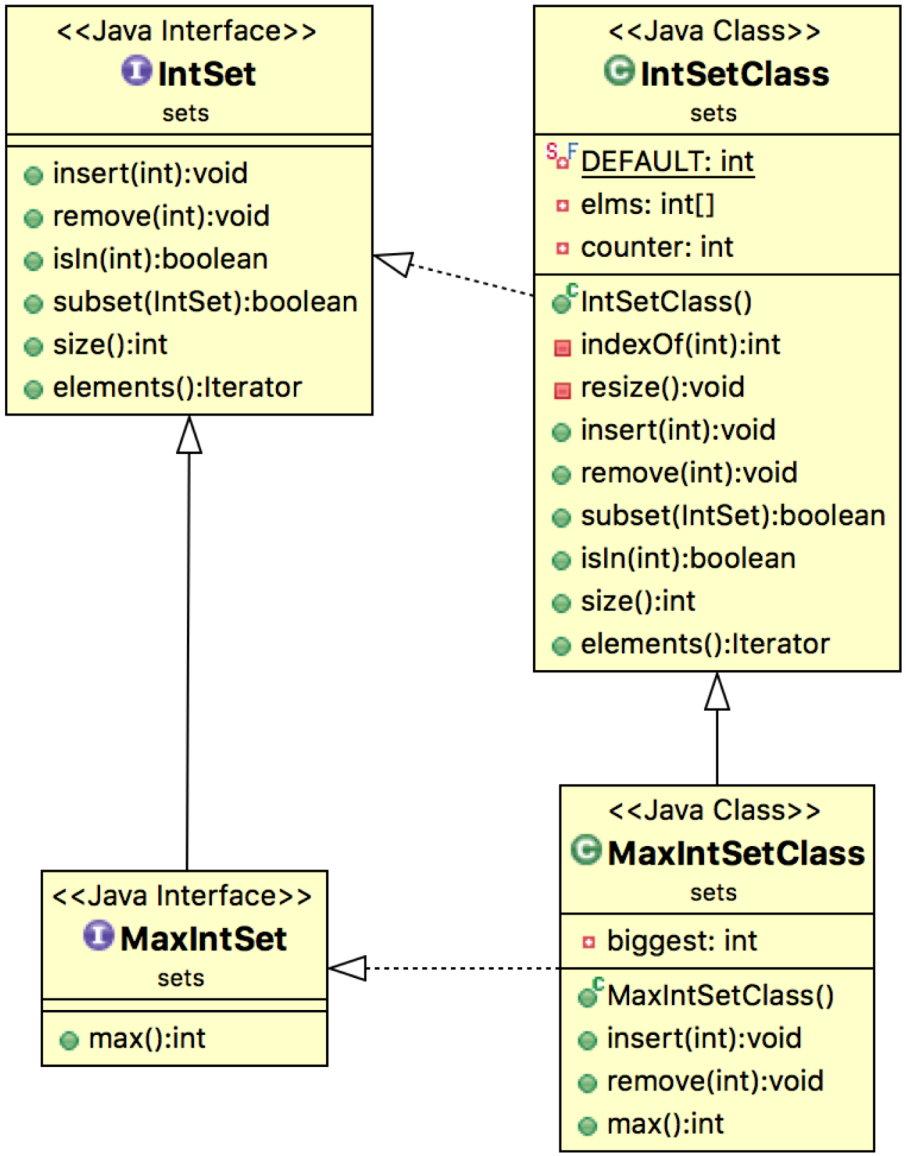
Como usar os métodos das sub-classes?


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



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

Herança de interfaces

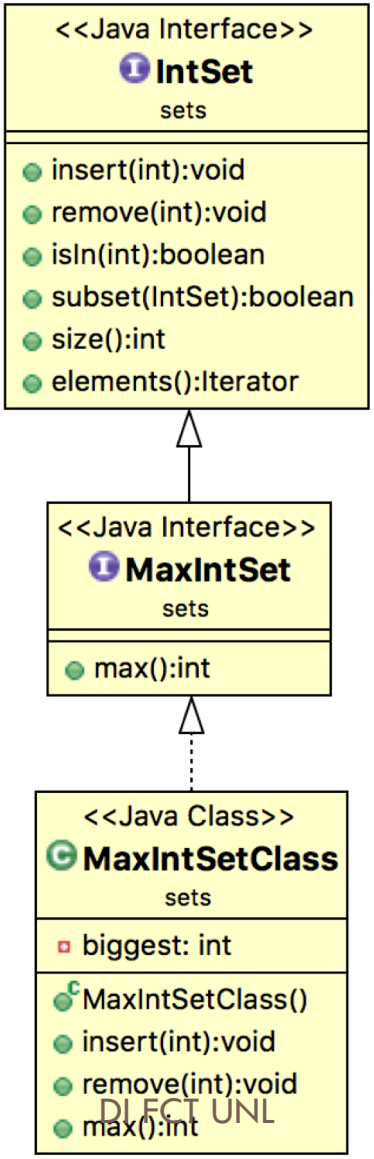


- Para estas situações, criamos uma sub-interface da interface original, que declara as novas operações

Herança de interfaces

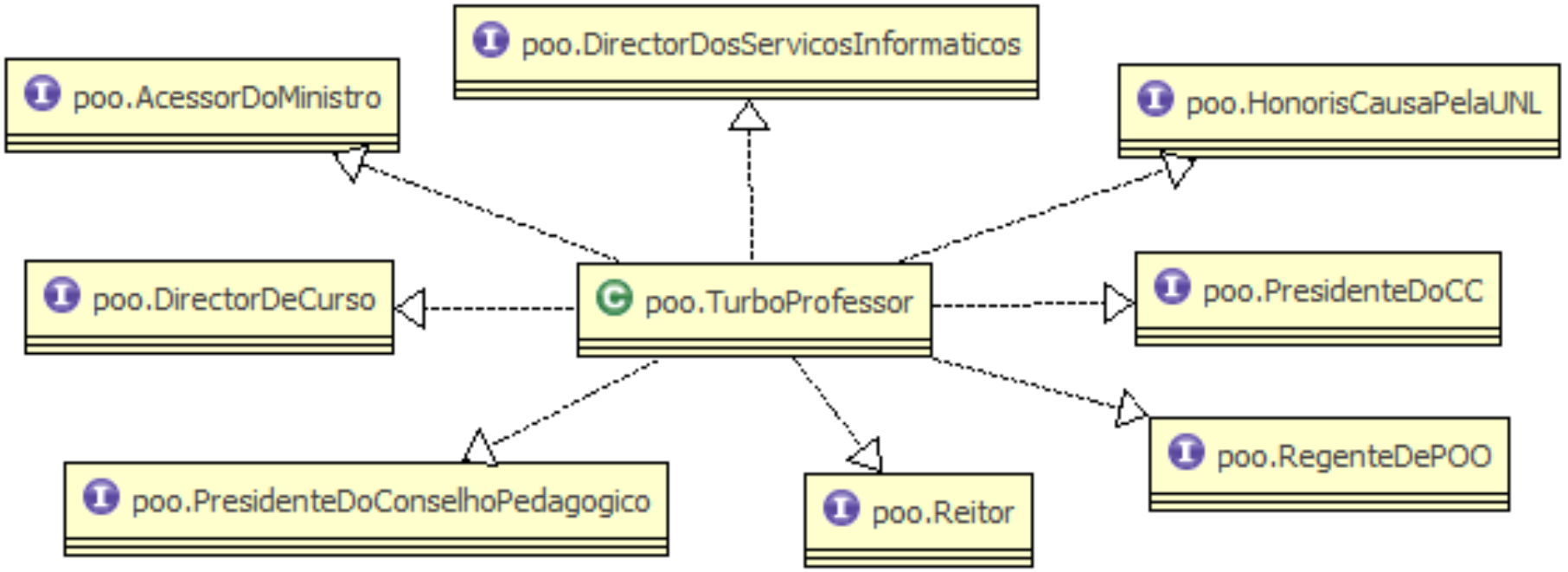

```
public interface MaxIntSet extends IntSet {
   /**
    * Retorna o máximo da colecção de inteiros.
    * Assume-se que esta operação só pode ser
    * chamada se o conjunto não for vazio.
    * @pre - this.size() > 0
    * @return - o máximo da colecção.
    */
 public int max();
}

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



Implementação de múltiplas interfaces

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



Implementação de múltiplas interfaces

- A cláusula **`extends`** é colocada **antes** da cláusula **`implements`**.

```
public class Student extends Person implements IStudent,
Navigable, Mutable, Printable, Cloneable, Comparable,
Serializable, Synchronizable, Iterable, Growable, EtCetera {
   //...
```

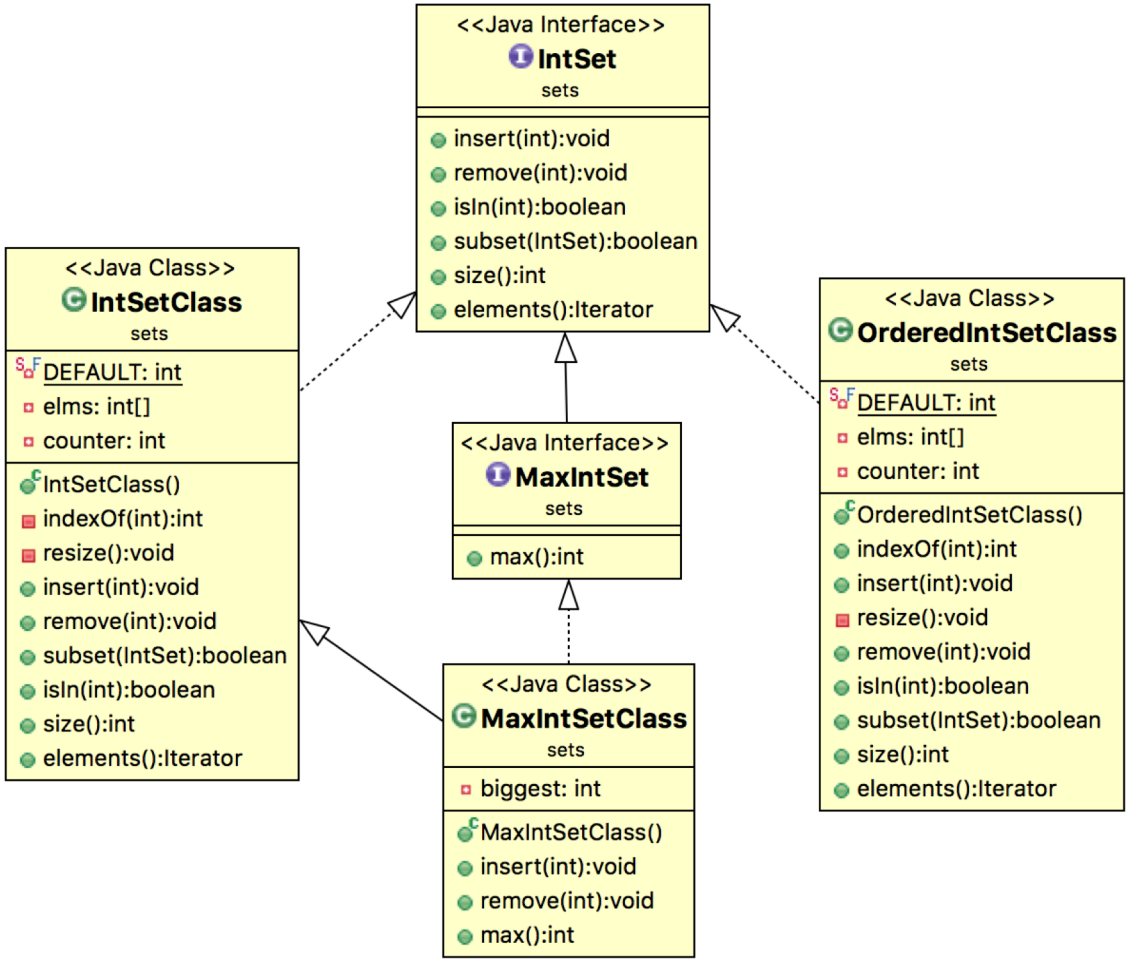
  - As diversas interfaces surgem numa única cláusula **`implements`**, separadas por vírgulas

Herança multipla?

11

Classes abstractas

Repetição de código



- 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

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

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? rep

- 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."

Factorização de código

- 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

- Factorização contribui para a **extensibilidade** do código
  - O código diz-se **extensível** se for possível acrescentar-lhe novas funcionalidades sem ter de alterar as já existentes
- 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

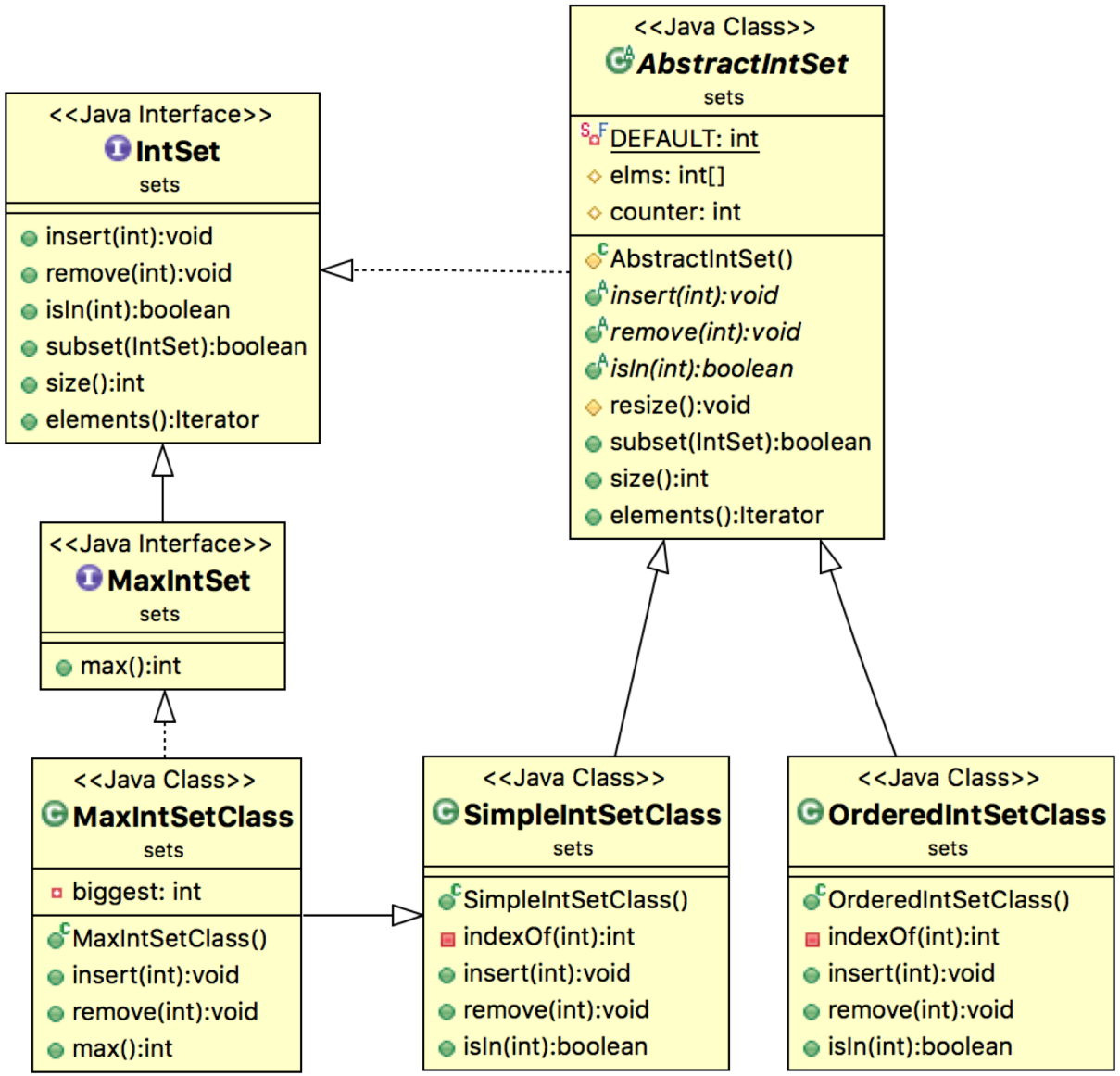
Classes e métodos abstractos

- 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

- 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 sub-classes implementar esses métodos

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



A classe `AbstractIntSet`


```
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.
   */
  protected int[] elms;

  /**
   * Dimensão do conjunto. Protegida, porque todas as
   * implementações concretas de um conjunto vão ter de
   * manter esta variável.
   */
  protected int counter;
```

A classe `AbstractIntSet`


```
protected AbstractIntSet() {
  elms = new int[DEFAULT];
  counter = 0;
}
```

```
public abstract void insert(int x);

public abstract void remove(int x);

public abstract boolean isIn(int x);
```

Classe com buracos

- Compete às sub-classes "preencher" os *slots* deixados em aberto

A classe AbstractIntSet


```
  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


```
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;
  }
```

A classe SimpleIntSetClass


```
  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`


```
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`


```
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`


```
  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
```