PROGRAMAÇÃO ORIENTADA PELOS OBJECTOS

Aula 8

Factorização de código: Classes Abstractas

Conjuntos de inteiros

Voltemos, então, ao nosso problema motivador:

- Queremos ter várias classes a representar conjuntos de inteiros
- Em particular, vamos ter conjuntos simples e conjuntos onde se pode calcular o máximo de forma muito expedita
- Vamos tirar partido da herança, para minimizar o código a desenvolver

O nosso programa deve permitir

- Criar um conjunto de inteiros
  - Simples, ou com a funcionalidade extra de encontrar rapidamente o máximo
- Acrescentar números inteiros a um conjunto
- Remover um número inteiro do conjunto
- Testar se um número pertence ao conjunto
- Testar se um conjunto está contido noutro conjunto
- Listar os elementos de um conjunto

Vamos definir uma família de conjuntos de inteiros

- `IntSet` **fornece um conjunto adequado de métodos**
  - Conjuntos alteráveis, não limitados, de inteiros
  - `public void insert(int x)`
    - Acrescenta o inteiro `x` ao conjunto
  - `public void remove(int x)`
    - Remove o inteiro `x` do conjunto
  - `public boolean isIn(int x)`
    - Se `x` pertence ao conjunto, retorna `true`, caso contrário, retorna `false`
  - `public boolean subset(IntSet s)`
    - Se this é sub-conjunto de `s` retorna `true`, caso contrário, retorna `false`
  - `public Iterator elements()`
    - Retorna um iterador de inteiros, para suportar as listagens

FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE NOVA DE LISBOA

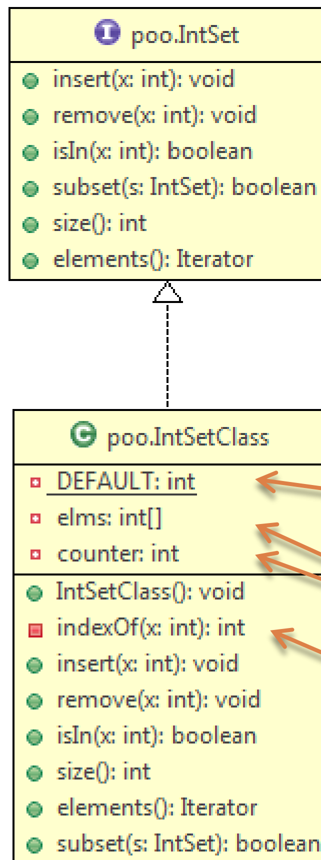
A interface IntSet


```java
// Comentários omitidos por economia de espaço no slide :-(
public interface IntSet {
  public void insert(int x);
  public void remove(int x);
  public boolean isIn(int x);
  public boolean subset(IntSet s);
  public int size();
  public Iterator elements();
}
```

poo.IntSet

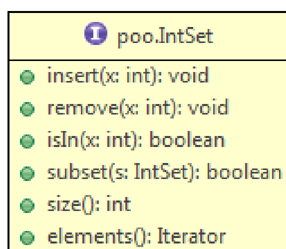
- insert(x: int): void
- remove(x: int): void
- isIn(x: int): boolean
- subset(s: IntSet): boolean
- size(): int
- elements(): Iterator

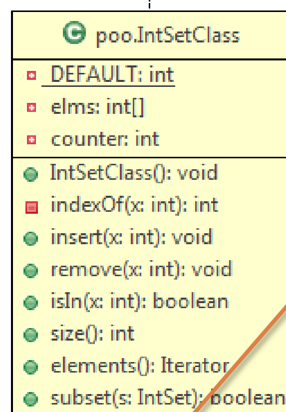
A classe IntSetClass

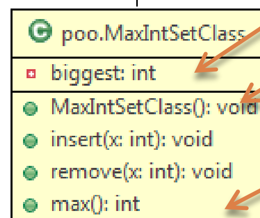


- Vamos definir uma classe que implemente a interface `IntSet` de modo a criar Conjuntos de inteiros simples
  - Esquema de implementação da interface a que já estamos habituados ☺
    - Acrescentamos uma constante, com o tamanho por omissão
    - Duas variáveis, com um vector acompanhado
    - Um método auxiliar, privado, já nosso velho conhecido, para descobrir a posição do elemento

Uma família de conjuntos de inteiros





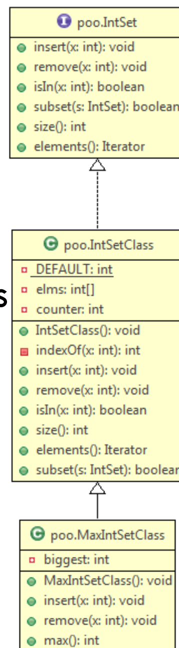


- `MaxIntSet` é uma sub-classe de `IntSetClass`
- Por ser sub-classe de `IntSetClass`, também implementa a interface `IntSet`:
  - Símbolo de herança
  - Comportamento semelhante ao de `IntSetClass`, mas com um método extra `max` que retorna o maior elemento do conjunto
    - Variável `biggest` acrescentada
    - Novo construtor
    - `insert` e `remove` redefinidos
    - Método `max` acrescentado

Especificação de `IntSetClass`


```java
public class IntSetClass implements IntSet {



  public IntSetClass(){}

  public void insert(int x)  {...}
  public void remove(int x)  {...}
  public boolean isIn(int x)  {...}
  public boolean subset(IntSet s)  {...}
  public int size(){...}
  public Iterator elements() {...}
}
```

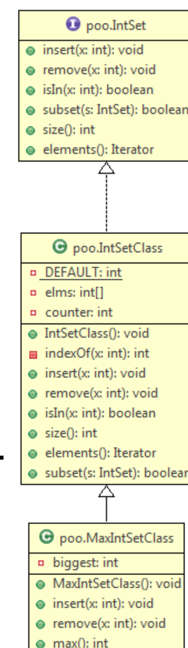
Não são usados membros protegidos, neste exemplo. Isto significa que as sub-classes de IntSetClass apenas lhe podem aceder através da sua interface pública. O nível de acesso é aceitável, porque o iterador permite visitar todos os elementos da colecção.



Implementação de IntSetClass

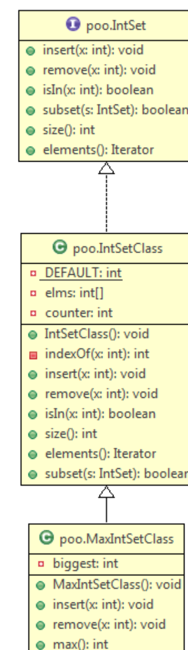

```
public class IntSetClass implements IntSet {
  private static final int DEFAULT = 10;
  private int[] elms;
  private int counter;

  public IntSetClass(){...}
  private int indexOf(int x)  {...}
  public void insert(int x)  {...}
  public void remove(int x)  {...}
  public boolean isIn(int x)  {...}
  public boolean subset(IntSet s)  {...}
  public int size(){...}
  public Iterator elements()  {...}
}
```

Esta constante, as duas variáveis e o método `indexOf` são privados. São inacessíveis fora desta classe, mesmo para as sub-classes.



Implementação de 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;
  }

  private int indexOf(int x) {
    int index = -1;
    int i = 0;
    while (i < counter && index == -1) {
      if (elms[i]==x)
        index = i;
        i++;
    }
    return index;
  }
```

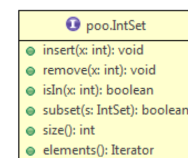


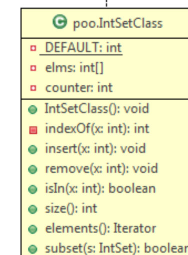
Implementação de `IntSetClass`

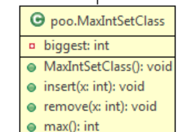

```
public void insert(int x) {
  if (indexOf(x) == -1) {
    if (counter == elms.length) {
      int[] tmp = new int[elms.length*2];
      for (int i = 0; i < counter; i++)
        tmp[i] = elms[i];
      elms = tmp;
    }
    elms[counter++] = x;
  }
}

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

Só inserimos o elemento se ele não existir no conjunto

Só removemos o elemento se ele existir no conjunto







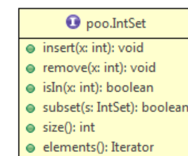
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

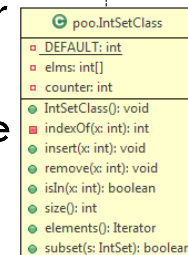
Implementação de IntSetClass

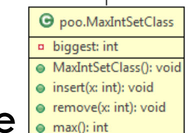

```
public boolean subset(IntSet s) {
  if (s == null) return false;
  else if (s.size() < this.size()) return false;
  else {
    for (int i = 0; i < counter; i++)
      if (!s.isIn(elms[i]))
        return false;
    return true;
  }
}
public boolean isIn(int x) {
  return (indexOf(x) != -1);
}
public int size(){
  return counter;
}
public Iterator elements() {
  return new IteratorClass(elms, counter);
}
}
```

Para que **this** possa estar contido em **s**, é necessário que **s** não seja **null** e que **this** tenha pelo menos tantos elementos como **s**.

Por outro lado, se pelo menos um dos elementos de **this** não pertencer a **s**, **this** não está contido em **s**.

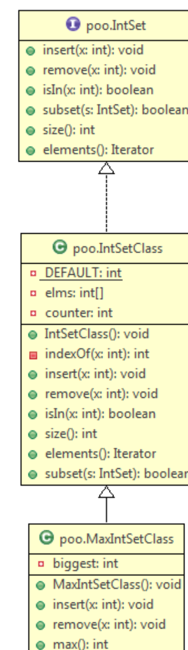






Especificação de MaxIntSetClass


```
public class MaxIntSetClass extends IntSetClass {
  private int biggest;
  public MaxIntSet() {...}
  public void insert(int x) {...}
  public void remove(int x) {...}
  public int max() {...}
}
```



- Repare que:
  - A constante **DEFAULT**, por ser de classe, não é herdada
  - As duas variáveis de instância da super-classe, **elms** e **counter**, são herdadas mas não podem ser acedidas directamente por ser privadas
  - O construtor da super-classe não é herdado
  - O método **indexOf** da super-classe não é herdado, por ser privado
  - Temos uma nova variável **biggest**
  - Temos um novo construtor **MaxIntSet** e um novo método max
  - Temos dois métodos redefinidos **insert** e **remove**
  - **Esta classe também implementa a interface IntSet**

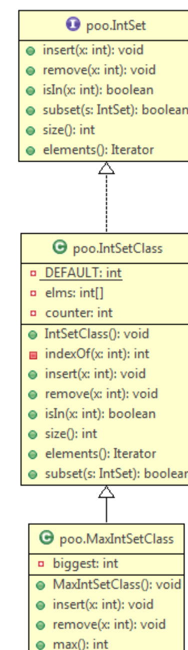
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Implementação de `MaxIntSetClass`


```java
public class MaxIntSetClass extends IntSetClass {
  private int biggest;

  public MaxIntSetClass() {
    super();
    biggest = 0;
  }

  public void insert(int x) {
    if (size() == 0 || x > biggest)
      biggest = x;

    super.insert(x);
  }
```

Como em todos os construtores devemos inicializar todas as variáveis de instância. Para garantir que as variáveis inacessíveis da super-classe também são inicializadas, usa-se a chamada ao construtor da super-classe!

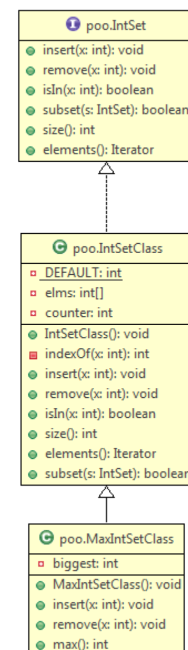
O insert começa por tratar do caso especial levantado pela necessidade de actualizar a variável biggest. No resto, seria igual ao da super-classe, portanto, delega nela a implementação.



Implementação de `MaxIntSetClass`

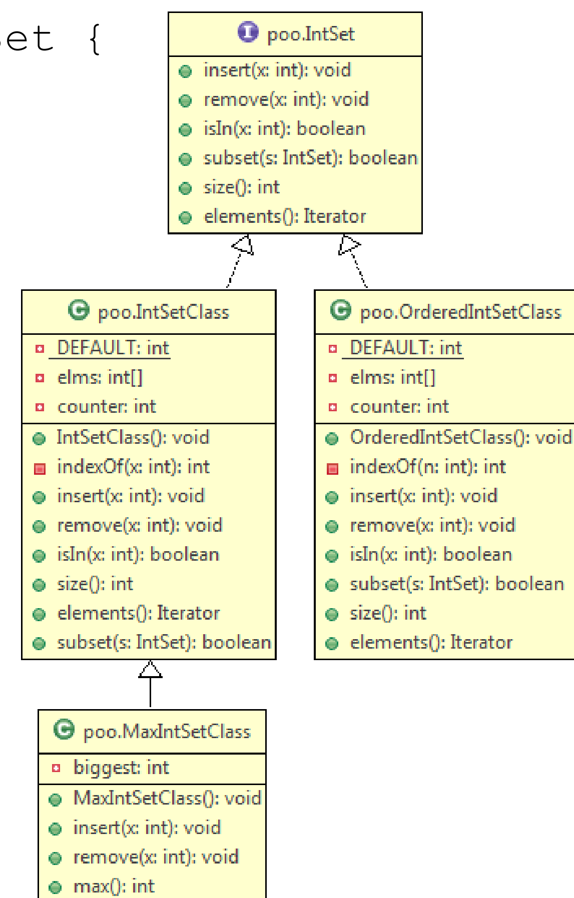

```
public void remove(int x) {
   super.remove(x);

   if (size()>0) && (x == biggest) {
      Iterator it = elements();
      biggest = it.next();
      int tmp;
      while (it.hasNext()) {
         tmp = it.next();
         if (tmp > biggest)
           biggest = tmp;
      }
   }
}

public int max() { return biggest; }
}
```

Começa-se por fazer a remoção tal como definida na super-classe. Depois, acrescenta-se o que é específico (neste caso, a actualização da variável `biggest`).



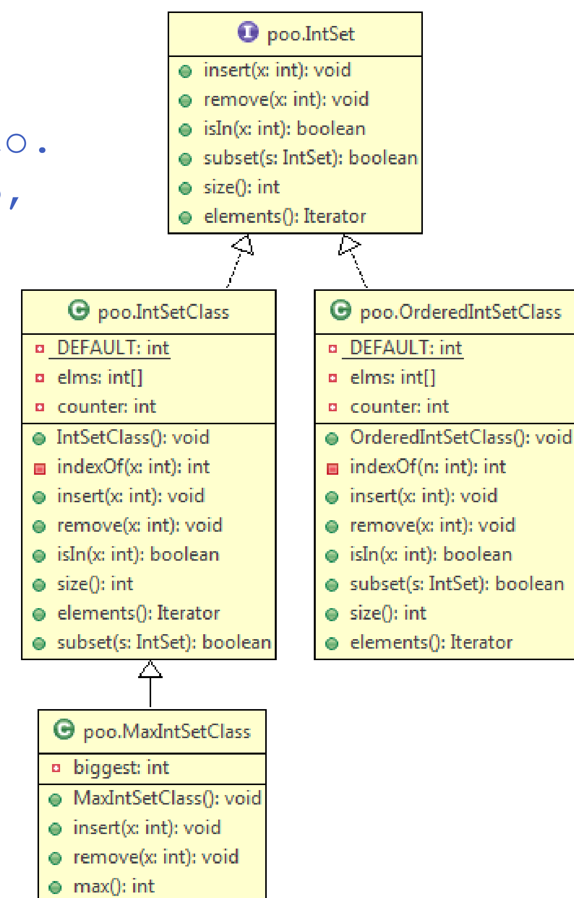
Implementação de OrderedIntSetClass


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

  /**
   * Inicializa o vector acompanhado,
   * de modo a representar
   * um conjunto vazio de inteiros.
   */
  public OrderedIntSetClass(){
    elms = new int[DEFAULT];
    counter = 0;
  }
```



Implementação de OrderedIntSetClass

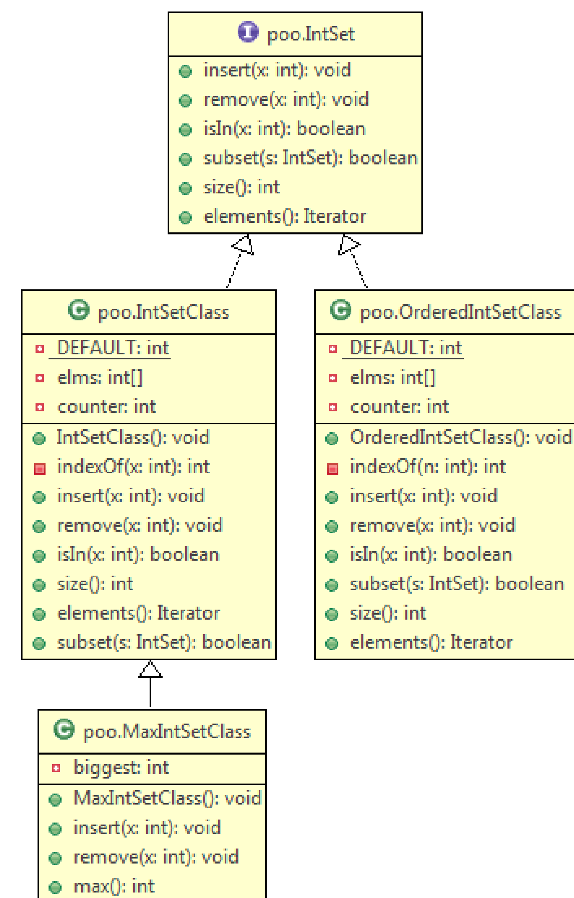

```java
/**
 * Devolve o índice do elemento <code>n</code>.
 * @param n - o elemento a pesquisar no conjunto.
 * @return - o índice com a posição do elemento,
 * ou <code>-1</code>, se não existir.
 */
private int indexOf(int n) {
   boolean found = false;
   int low = 0;
   int high = counter-1;
   int mid = -1;
   while (!found && low <= high) {
      mid = (low+high)/2;
      if (elms[mid] == n) found = true;
      else if (n < elms[mid]) high = mid-1;
      else low = mid+1;
   }
   if (found)
      return mid;
   return -1;
}
```



FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE NOVA DE LISBOA

Implementação de OrderedIntSetClass

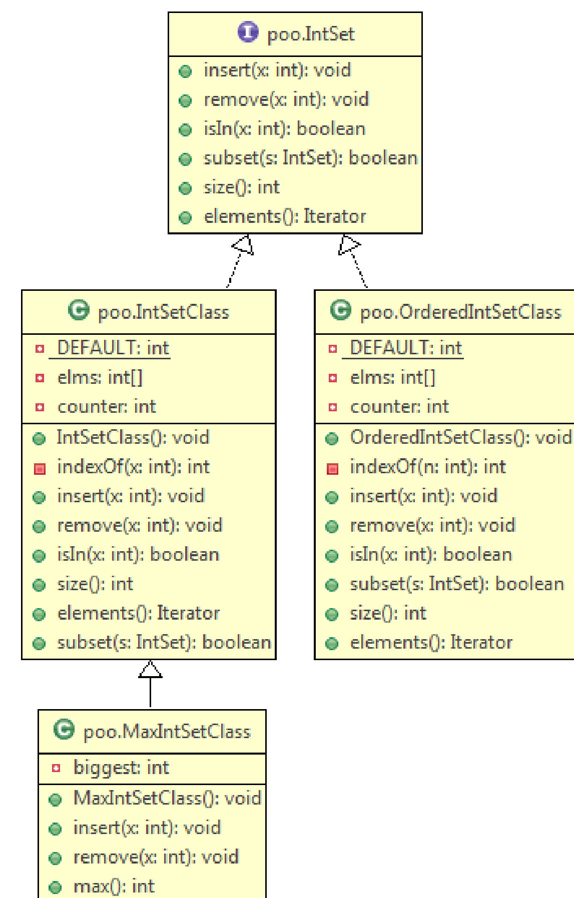

```java
public void insert(int x) {
  int pos = 0;
  boolean lower = true;
  while ((pos < counter) && lower) {
    lower = elms[pos] < x;
    if (lower)
      pos++;
  }
  if (elms[pos] != x) {
    if (counter == elms.length) {
      int[] tmp = new int[elms.length*2];
      for (int i = 0; i < counter; i++)
        tmp[i] = elms[i];
      elms = tmp;
    }
    for (int i = counter; i > pos; i--)
      elms[i] = elms[i-1];
    elms[pos] = x;
    counter++;
  }
}
```



FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

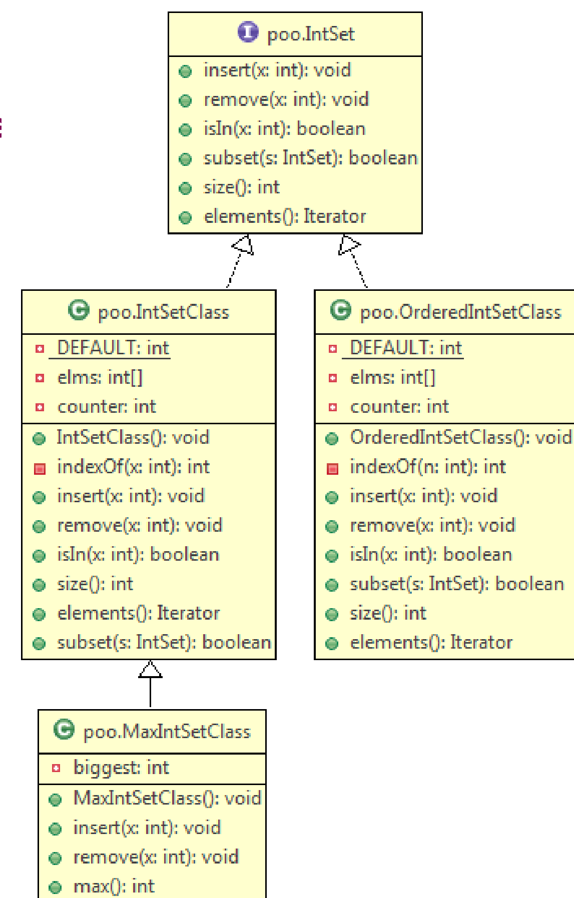
Implementação de OrderedIntSetClass


```java
public void remove(int x) {
   int index = indexOf(x);
   if (index != -1) { // inteiro existente
      int i = index;
      while (i < counter-1) {
         elms[i] = elms[i+1];
         i++;
      }
      counter--;
   }
}

public boolean isIn(int x) {
   return indexOf(x) != -1;
}
```

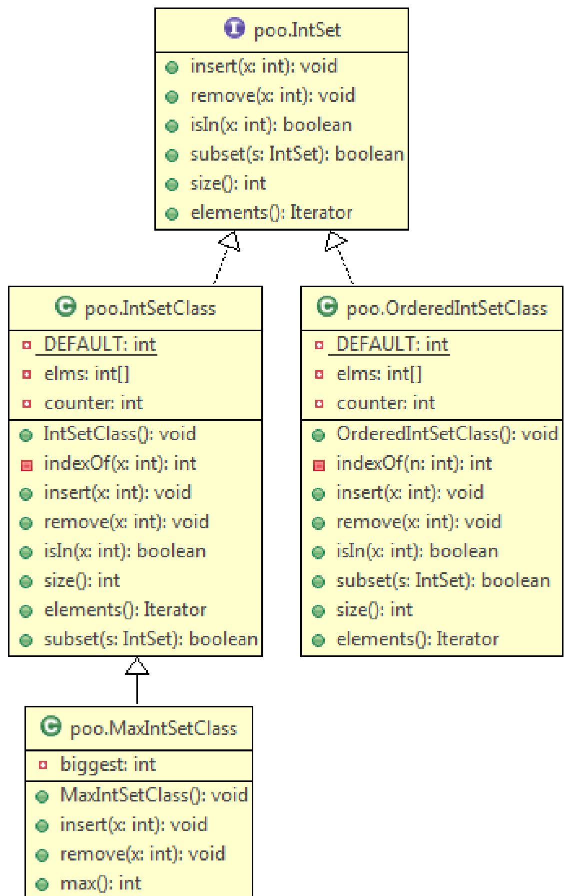


Implementação de OrderedIntSetClass


```
public boolean subset(IntSet s) {
  if (s == null) return false;
  else if (s.size() < this.size()) return false
  else {
    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 OrderedIntSet
```



Hierarquia completa



- 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!

FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE NOVA DE LISBOA

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 diferentes

  public boolean isIn(int x) {
    return indexOf(x) != -1;
  }
```

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 diferentes

  public boolean isIn(int x) {
    return indexOf(x) != -1;
  }
```


FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

IntSetClass vs. OrderedIntSet

IntSetClass

```
public boolean subset(IntSet s) {
  if (s == null) return false;
  else if (s.size() < this.size())
    return false;
  else {
    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 == null) return false;
  else if (s.size() < this.size())
    return false;
  else {
    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
```


FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Código repetido?

- E se descobrirmos um bug?
  - Vamos corrigir o código em TODAS as cópias
  - A "lei de Murphy" garante que nos vamos esquecer de corrigir um destes bugs replicados, mais cedo ou mais tarde
  - Mesmo se não nos esquecermos, corrigir o mesmo *bug* n vezes sai mais caro do que corrigir apenas uma vez
- E se quisermos simplesmente acrescentar algo novo?
  - Vamos acrescentar a nova funcionalidade em TODAS as cópias
  - A "lei de Murphy" garante que nos vamos esquecer de acrescentar a nova funcionalidade numa destas classes replicadas, mais cedo ou mais tarde
  - Mesmo que não nos esquecêssemos, acrescentar uma funcionalidade n vezes sai mais caro do que apenas uma vez

Classes abstractas

Factorização de código

- O sistema de herança permite escrever código **factorizado**, ou seja, código **sem redundância**
  - Quando várias classes apresentam código replicado, provavelmente são implementações de casos particulares de um conceito mais geral
  - Devemos identificar esse conceito e materializá-lo numa classe que concentre em si o código comum, eliminando assim a redundância
  - As classes originais passam a incorporar o código factorizado através de herança

Vantagens da factorização

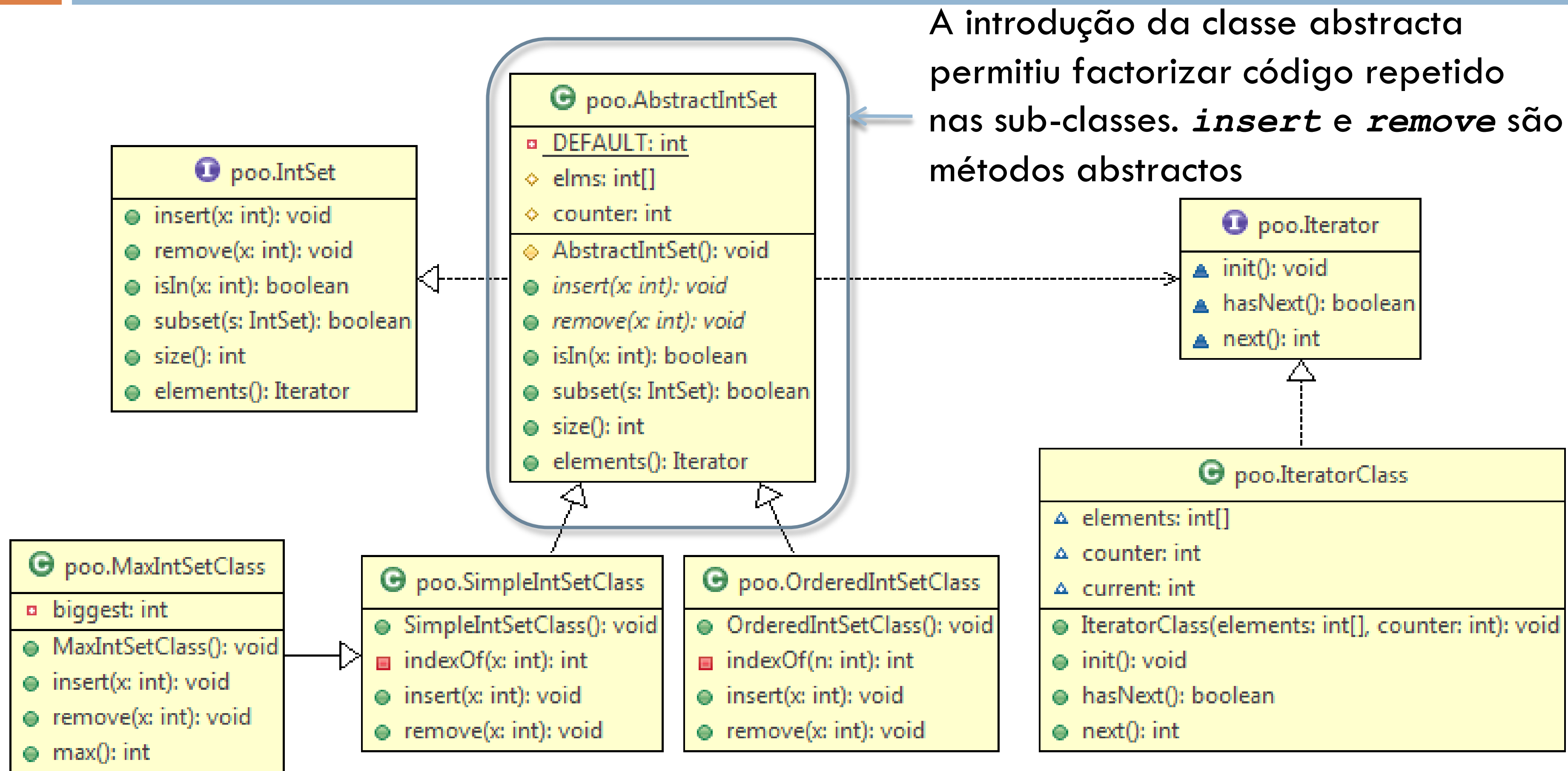
- Factorização contribui para a **extensibilidade** do código
  - O código diz-se **extensível** se for possível acrescentar-lhe novas funcionalidades sem ter de alterar as já existentes
- Aumenta o nível de **generalidade** das abstracções usadas
- Torna o código mais compacto e bem organizado
- Facilita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

Classes abstractas (e métodos abstractos)

- Por vezes, quando factorizamos código, surgem classes tão gerais que não faz sentido que sejam directamente instanciadas
  - Essas classes são denominadas classes abstractas que se caracterizam por:
    - Implementar apenas parcialmente um tipo
    - Não poder ser instanciadas
    - Poder conter métodos sem corpo, ou seja **métodos abstractos**
- Em Java, essas classes declaram-se com a palavra reservada **`abstract`**
- Em Java, os métodos abstractos também se declaram com a palavra reservada **`abstract`**

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

A introdução da classe abstracta permitiu factorizar código repetido nas sub-classes. ***insert*** e ***remove*** são métodos abstractos



Classes abstractas

- Uma classe abstracta pode ter:
  - Algumas variáveis de instância, partilhadas por todas as sub-classes
    - Nesse caso, a classe abstracta deve ter um ou mais construtores, para inicializar as variáveis de instância de modo adequado
      - Estes construtores não podem ser usados por classes clientes
      - Mas podem ser usados pelas sub-classes da classe abstracta, para inicializar a parte da representação definida pelas variáveis de instância da classe abstracta
  - Métodos abstractos e não abstractos
    - As implementações dos métodos não abstractos tiram frequentemente partido dos métodos abstractos
      - Assim, a super-classe pode definir a parte genérica da implementação
      - Cabe então às sub-classes definir os detalhes em falta

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;
```

Os métodos abstractos insert e remove terão de lidar de modo eficiente com estas variáveis!

A classe **`AbstractIntSet`**


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

public abstract void insert(int x);

public abstract void remove(int x);

public boolean isIn(int x) {
  Iterator it = elements();
  while (it.hasNext()) {
    if (it.next() == x)
      return true;
  }
  return false;
}
```

Estes métodos são abstractos. Terão de ser implementados nas sub-classes concretas de **`AbstractIntSet`**.

FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE NOVA DE LISBOA

A classe AbstractIntSet


```
public boolean subset(IntSet s) {
  if (s == null) 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 index = -1;
    int i = 0;
    while (i < counter && index == -1) {
      if (elms[i] == x)
        index = i;
      i++;
    }
    return index;
  }
```

Para construir objectos da classe concreta, começamos por construir a parte deles definida na super-classe abstracta.

A classe SimpleIntSetClass


```
  public void insert(int x) {
    if (indexOf(x) == -1) {
      // Se necessário, aumenta a capacidade do vector
      if (counter == elms.length) {
        int[] tmp = new int[elms.length*2];
        for (int i = 0; i < counter; i++)
          tmp[i] = elms[i];
        elms = tmp;
      }
      elms[counter++] = x;
    }
  }
  public void remove(int x) {
    int index = indexOf(x);
    if (index != -1) {
      for (int i = index; i < counter-1; i++)
        elms[i] = elms[i+1];
      counter--;
    }
  }
} // Fim da classe
```

A classe **OrderedIntSetClass**


```
public class OrderedIntSetClass extends AbstractIntSet {

  public OrderedIntSetClass(){
    super();
  }
```

Para construir objectos da classe concreta, começamos por construir a parte deles definida na super-classe abstracta.

A classe OrderedIntSetClass


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

Pesquisa binária

FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE NOVA DE LISBOA

A classe **OrderedIntSetClass**


```
public void insert(int x) {
  int pos = 0;
  boolean lower = true;
  while ((pos < counter) && lower) {
    lower = elms[pos] < x;
    if (lower)
      pos++;
  }
  if (elms[pos] != x) {// novo inteiro, temos de o inserir.
    if (counter == elms.length) {
      int[] tmp = new int[elms.length*2];
      for (int i = 0; i < counter; i++)
        tmp[i] = elms[i];
        elms = tmp;
    }
    for(int i = counter; i > pos; i--)
      elms[i] = elms[i-1];
    elms[pos] = x;
    counter++;
  }
}
```

Começa por procurar a posição em que o novo elemento deve aparecer. Se o elemento for realmente novo, será depois acrescentado ao conjunto.

A classe **OrderedIntSetClass**


```
  public void remove(int x) {
    int index = indexOf(x);
    if (index != -1) { // inteiro existente
      int i=index;
      while (i < counter-1) {
        elms[i] = elms[i+1];
        i++;
      }
      counter--;
    }
  }
} // Fim da classe OrderedIntSetClass
```