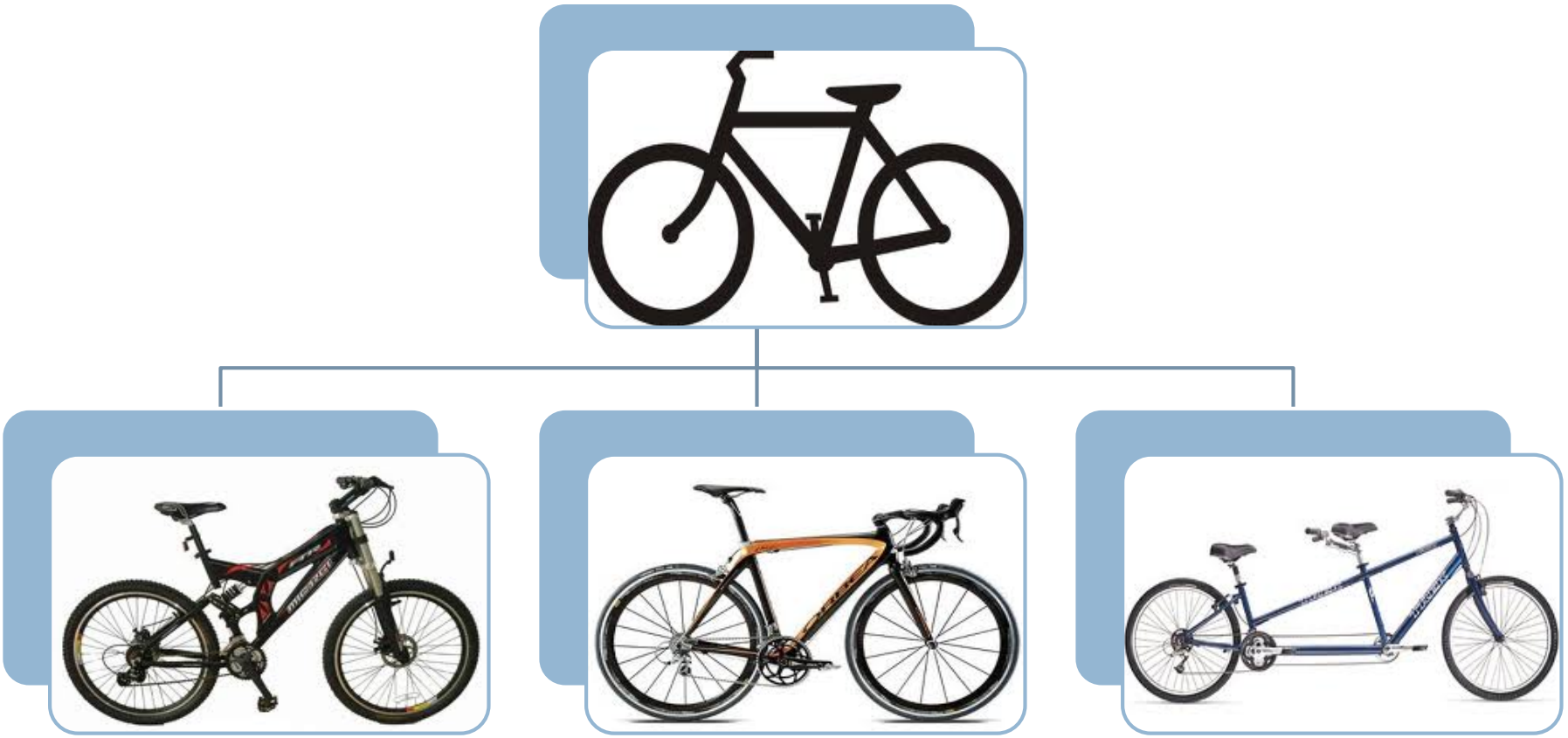
PROGRAMAÇÃO ORIENTADA PELOS OBJECTOS

Hierarquias de tipos

Famílias de entidades relacionadas

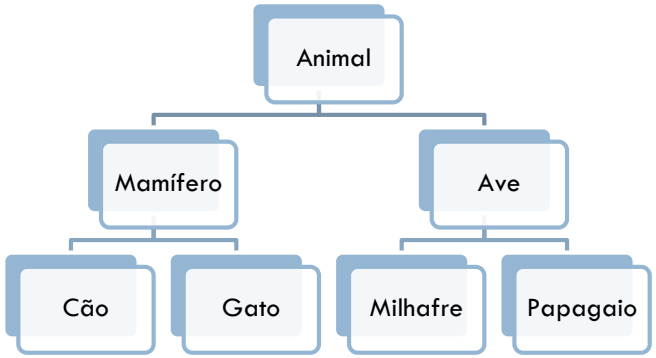


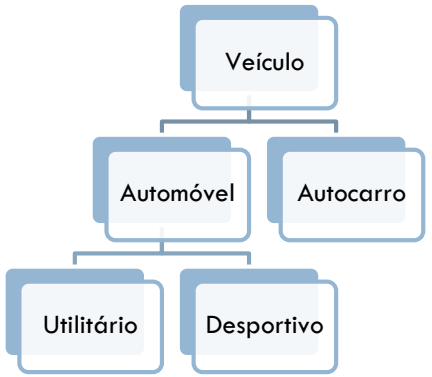
Famílias?

- Como tirar partido da abstracção de dados definindo famílias de tipos relacionados?
  - Os membros de uma família têm algum comportamento comum
    - Têm um conjunto semelhante de métodos
    - As chamadas a esses métodos resultam em comportamentos semelhantes
    - Os elementos de uma família podem divergir
      - Estendendo alguns dos comportamentos comuns da família
      - Acrescentando novos comportamentos

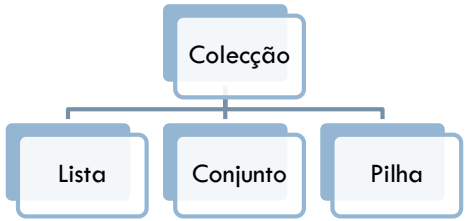
Famílias!

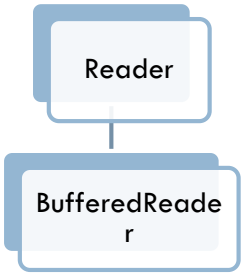
- A família pode corresponder a uma hierarquia do mundo real





- Ou no mundo da programação





Família de tipos

- Uma família de tipos é definida por uma hierarquia de tipos
  - No topo da hierarquia, está um tipo que define o comportamento comum a todos os membros da família
    - Isto pode incluir quer as assinaturas, quer o comportamento das operações que funcionam de modo comum para os membros da família
  - Os outros elementos da família são **sub-tipos** deste tipo que está no topo da hierarquia: o **super-tipo**
    - Por sua vez, estes sub-tipos podem ter, também eles, os seus respectivos sub-tipos
      - A hieraquia pode ter mais que dois níveis!

Famílias de tipos usadas de duas formas

- Famílias usadas para fornecer diferentes implementações de um tipo
  - Neste caso, os sub-tipos não adicionam novo comportamento, com excepção de que cada um deles terá o seu respectivo construtor
  - A classe implementando o sub-tipo implementa exactamente o comportamento definido pelo super-tipo
- Famílias cujos sub-tipos estendem o comportamento do seu super-tipo
  - Por exemplo, através do fornecimento de novos métodos
  - A hierarquia nestas famílias pode ser multi-nível
  - Na parte de baixo da hierarquia, podem existir várias implementações de um determinado sub-tipo

Famílias de tipos usadas de duas formas

- Famílias usadas para fornecer diferentes implementações de um tipo

- Famílias cujos sub-tipos estendem o comportamento do seu super-tipo

Hierarquia de tipos

- Define uma família de tipos consistindo num super-tipo e nos seus sub-tipos
- Algumas famílias de tipos são usadas para fornecer diferentes implementações de um tipo: os sub-tipos fornecem implementações diferentes do seu super-tipo
- Mais genericamente, os sub-tipos estendem o comportamento dos super-tipos, por exemplo acrescentando-lhes novos métodos
- P**rincípio da substituição**
  - Os sub-tipos comportam-se de acordo com a especificação dos seus super-tipos

Para que serve uma hierarquia de tipos?

- Permite "relaxar" a verificação de tipos em certos pontos do programa
  - atribuição de um objecto a uma variável
  - passagem de argumentos
  - a forma como as chamadas a métodos são tratadas
    - em particular, na **selecção do bloco de código exacto que é executado** em resposta às chamadas
- Certos pontos dum programa aceitam objectos de tipos diferentes dos declarados, desde que o novo tipo seja um sub-tipo do tipo declarado
- Serve de suporte a **polimorfia**: a capacidade dum objecto ser de múltiplos tipos simultaneamente

Atribuição

- Uma variável pode ser declarada como pertencendo a um tipo, mas na realidade referir-se a um objecto que é de um sub-tipo desse tipo

```
Animal a1 = new DogClass("Boby");
Animal a2 = new CatClass("Tareco");
```

- Ou seja, variáveis do tipo **`Animal`** podem, na realidade referir-se a objectos do tipo **`DogClass`**, **`CatClass`**, ou qualquer outro sub-tipo de **`Animal`**
  - Para distinguir entre ambos, chamemos ao tipo declarado o tipo **aparente**, e ao usado na instanciação o tipo **real** (em Inglês, *apparent* e *actual type*, respectivamente)

Verificação de tipos pelo compilador

- O compilador verifica os tipos com base na informação para ele disponível
  - Usa sempre os tipos *aparentes*, não os *reais*, para determinar que chamadas a métodos são legais
  - O objectivo da verificação é garantir que o objecto tem **mesmo** um método com a assinatura apropriada
    - Pode é não saber qual é o tipo *real*...
  - Recorde o conceito de *early binding*, na aula anterior

Dispatching

- Por vezes o compilador pode não saber qual é o tipo real de um objecto
- O código a correr depende do tipo real do objecto
- A chamada ao método correcto é conseguida através de um mecanismo denominado **dispatching**
  - Em vez de gerar código para chamar directamente o método, o compilador gera código para descobrir, em tempo de execução, qual o método que deve ser executado, indo depois para esse método
- Recorde o conceito de *late binding*, na aula anterior

Como definir uma hierarquia

- Especificação do super-tipo é, frequentemente, incompleta
  - Por exemplo, pode não ter construtores
- Especificação de sub-tipos é feita relativamente à especificação dos super-tipos
  - Foco no que o sub-tipo tem de novo, sempre no pressuposto que as partes públicas do super-tipo são respeitadas
    - Tipicamente, acrescenta construtores do sub-tipo
    - Métodos adicionais
    - Se o sub-tipo alterar a especificação de métodos definidos no super-tipo, então tem de fornecer a nova especificação desses métodos
      - Há limites para o tipo de alterações permitidas (já voltaremos aqui)

Implementação da hierarquia

- Por vezes, os super-tipos não são implementados de todo, ou apenas são parcialmente implementados
- A implementação do super-tipo pode disponibilizar informação extra a potenciais sub-tipos, com métodos e campos destinados exclusivamente aos sub-tipos
- Se o super-tipo é implementado, ainda que parcialmente, o sub-tipo é uma extensão da implementação do super-tipo
  - A implementação do sub-tipo pode **herdar** variáveis de instância e métodos do super-tipo
  - A implementação do sub-tipo pode também **redefinir** os métodos herdados

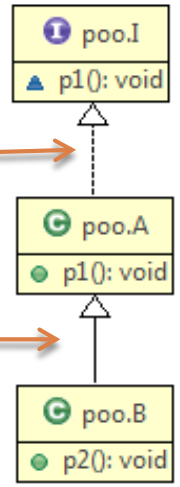
Definição de hierarquias, em Java

- Utilização do mecanismo de **herança**
  - Este mecanismo permite que uma classe seja uma subclasse de outra classe (a super-classe) e que implemente zero ou mais interfaces
- Super-tipos definidos como classes ou interfaces
  - Em qualquer caso, a classe ou interface fornece uma especificação do tipo
    - No caso da interface, apenas fornece a especificação
    - No caso da classe, também pode fornecer uma implementação parcial ou total do super-tipo

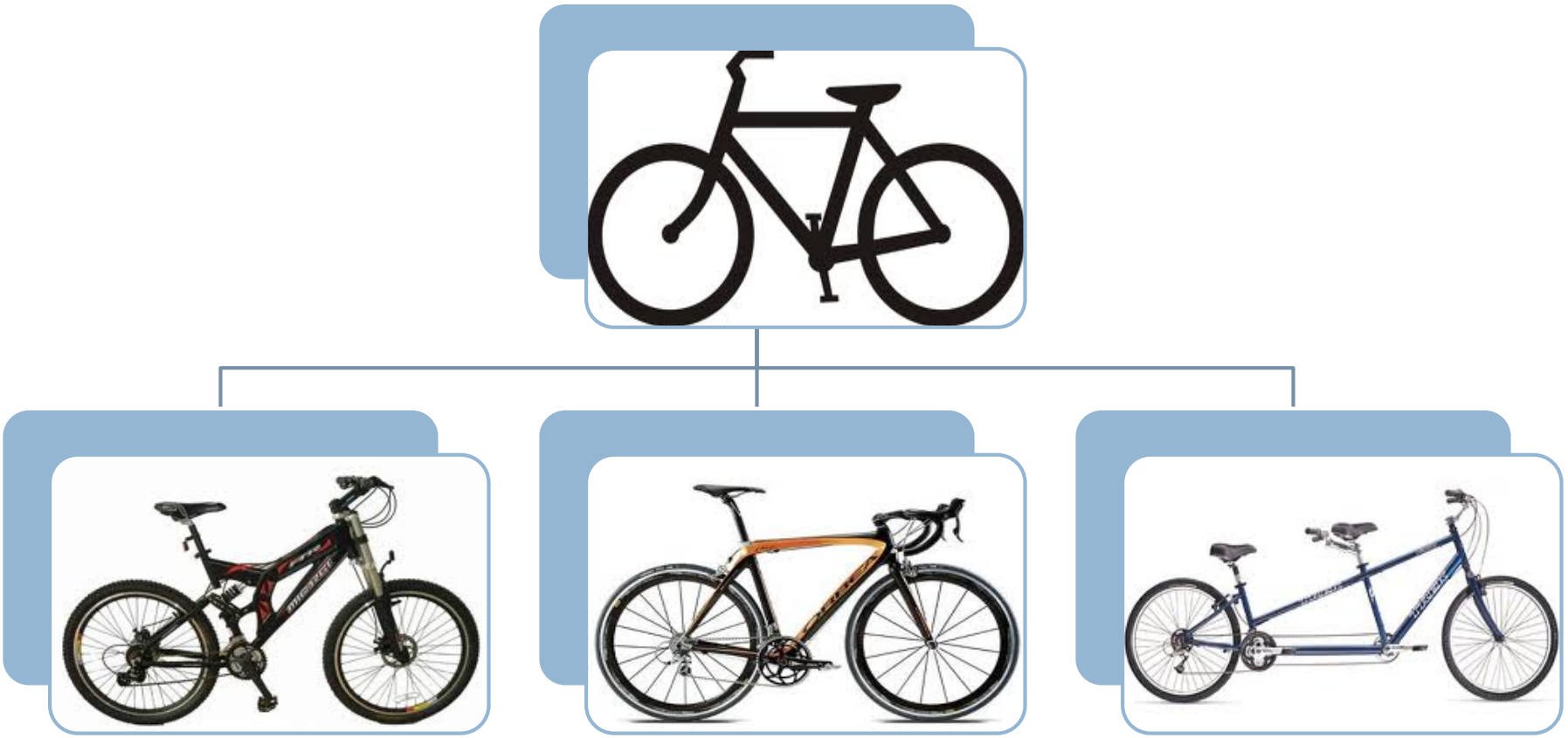
Relação de herança entre classes

- Usando o mecanismo de herança, o programador consegue criar uma nova classe (sub-classe) com base numa classe já existente (super-classe)
  - Terá apenas de definir as componentes da sub-classe que são adicionadas, ou modificadas, face à super-classe
  - As componentes da super-classe que não forem redefinidas são automaticamente herdadas
- Em Java, utiliza-se a palavra reservada **`extends`** para indicar que uma classe é extensão de outra classe

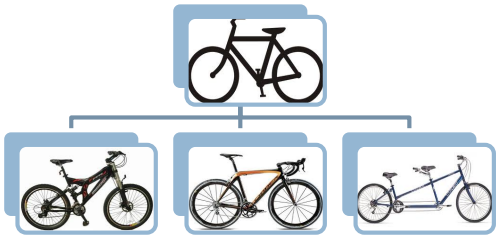
```
public interface I {                    // interface
    void p1();
}
public class A implements I {           // superclasse
    public void p1() { }
}
public class B extends A {              // subclasse
    public void p2() { } // B herda o método p1() de A
}
```



Declaração de classes e sub-classes: Bicicletas



Classe `BicycleClass`



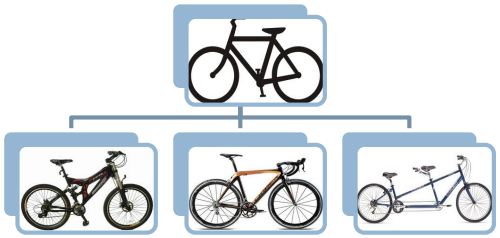

```
public class BicycleClass {
  private int speed;
  private int cadence;
  private int gear;

  public BicycleClass(int startCadence, int startSpeed, int startGear) {
    gear = startGear;
    cadence = startCadence;
    speed = startSpeed;
  }
  public void setCadence(int newValue) {
    cadence = newValue;
  }
  public void setGear(int newValue) {
    gear = newValue;
  }
  public void applyBrake(int decrement) {
    speed -= decrement;
  }
  public void speedUp(int increment) {
    speed += increment;
  }
}
```

3 variáveis de instância

Construtor

Outros métodos da classe BicycleClass (certamente haveria mais a acrescentar)

Sub-classe MountainBike




```
public class MountainBike extends BicycleClass {
    private int seatHeight; // Altura do assento;
     /**
      * Construtor de MountainBike
      * @param startHeight - altura do assento
      * @param startCadence - cadência da pedalada
      * @param startSpeed - velocidade inicial
      * @param startGear - mudança inicial
      */
     public MountainBike(int startHeight, int startCadence,
                         int startSpeed, int startGear) {
         super(startCadence, startSpeed, startGear);
         seatHeight = startHeight;
     }
     /**
      * Regula a altura do assento
      * @param newValue - nova altura do assento
      */
     public void setHeight(int newValue) {
         seatHeight = newValue;
     }
}
```

Esta classe especializa a definição da classe BicycleClass

A sub-classe tem uma nova variável de instância

A sub-classe tem o seu próprio construtor que invoca o construtor da super-classe, acrescentando, depois, nova funcionalidade

A sub-classe tem novos métodos

Declaração de uma sub-classe

- Uma sub-classe declara a sua super-classe indicando no cabeçalho da sua definição que estende (**`extends`**) essa classe
  - Isto significa que herda, automaticamente, todos os métodos da super-classe, com os mesmos nomes e assinaturas
  - Além disso, a sub-classe pode fornecer métodos e variáveis de instância **adicionais**
  - A sub-classe **não herda os construtores**. No caso de acrescentar novas variáveis de instância, estas devem ser inicializadas nos seus próprios construtores.

Declaração de uma sub-classe

- Uma sub-classe concreta pode acrescentar os seus próprios métodos
  - Além disso, a sub-classe pode **reimplementar**, ou **redefinir** (*override*), os métodos da super-classe
    - Estes métodos têm de ter assinaturas compatíveis com as definidas na super-classe
    - Não pode restringir a visibilidade do método herdado
    - Se o método herdado é público, a redefinição tem de ser igualmente pública
    - Se o método herdado tem visibilidade package, a redefinição tem de ser de package ou pública

Shadowing de variáveis de instância

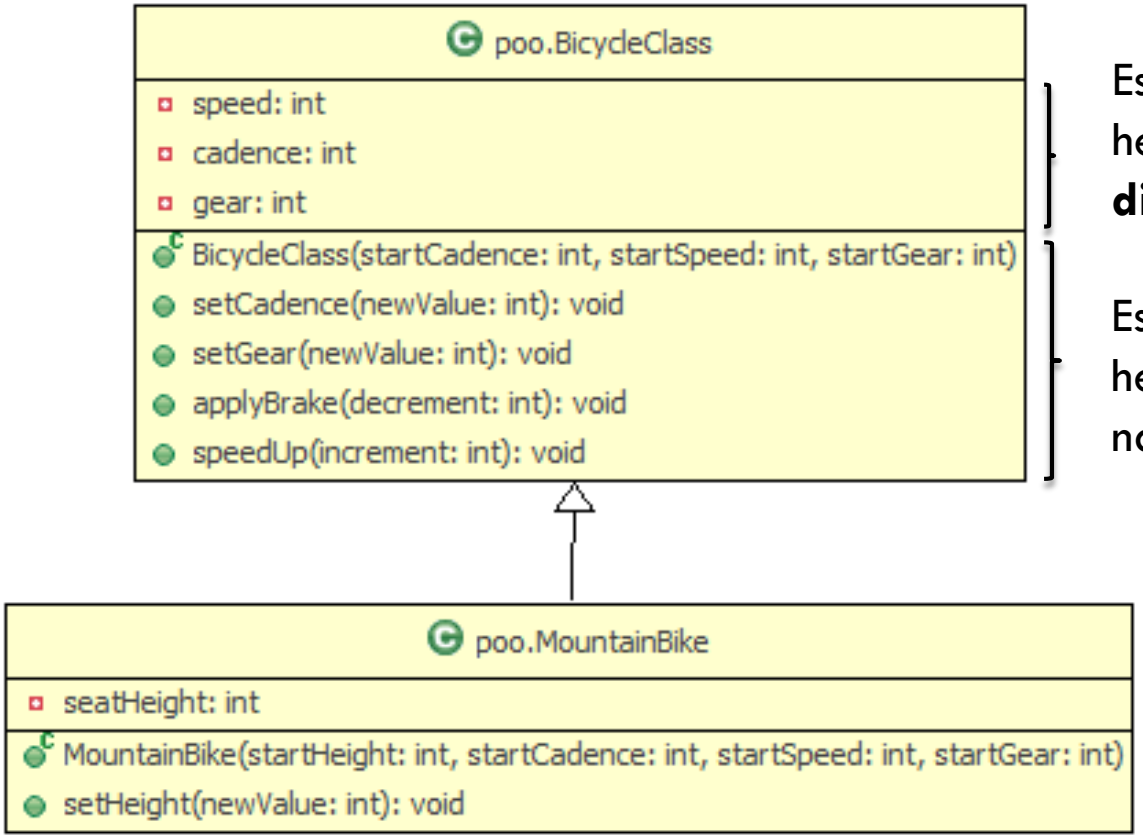
- Uma sub-classe **não pode** redefinir variáveis de instância, apenas métodos
  - Se uma subclasse define uma variável com o mesmo nome que uma variável herdada, **não redifine** a variável da super-classe
  - Apenas **tapa ou oculta** a variável herdada
  - Em Inglês isso é chamado de *shadowing* e é **mau estilo**, porque redunda em código confuso
    - O fenómeno de *shadowing* (variáveis ocultas) é **sempre** de evitar

O que cabe numa herança?

- O mecanismo de herança permite reutilizar nas sub-classes
  - Variáveis de instância definidas nas super-classes
  - Métodos de instância definidos nas super-classes
- Não são herdados
  - Os construtores da super-classe
  - Variáveis de classe definidas na super-classe
  - Métodos de classe definidos na super-classe
  - Constantes de classe definidas na super-classe

Ou seja, tudo o que declaramos com o modificador `static`

O que cabe na herança?



Estes atributos **privados** são herdados mas **não ficam disponíveis** na sub-classe

Estes métodos **públicos** são herdados e **ficam disponíveis** na sub-classe

Representação de um objecto da sub-classe

- Inclui as variáveis de instância declaradas na super-classe e as declaradas na sub-classe

Representação na sub-classe **MountainBike**

```
private int speed;        // herdada da super-classe
private int cadence;      // herdada da super-classe
private int gear;         // herdada da super-classe

private int seatHeight; // criada na sub-classe
```

Sem acesso directo (declaradas em `BicycleClass`)

Com acesso directo (declaradas em `MountainBike`)

- O **acesso directo** a detalhes de representação da super-classe apenas é possível se a super-classe tornar partes da sua implementação acessíveis às sub-classes

O que acontece aos membros de instância privados?

- A sub-classe **herda** os membros privados da sua super-classe, mas **não tem acesso** a eles!

```
// Algures na classe MountainBike...
29  public void slide() {
30      this.applyBrake(1);
31      // Além de travar, faz mais qualquer coisa para derrapar
32  }
```

- No entanto, se a super-classe tiver métodos acessíveis que usem os membros privados, os membros privados são usados **indirectamente**

Herança de métodos de instância

- Numa sub-classe não existe acesso directo a:
  - métodos declarados na super-classe como privados
  - métodos da super-classe que sejam re-definidos na sub-classe

```
public class A {
  private void a() { }
  public String p() { return "P do A"; }
}

public class B extends A {
  // B não tem acesso directo a a(), por ser privada
  // B não tem acesso directo a p() da classe A, por ser redefinida
  public String p() { return "P do B"; } // p() de B esconde p() de A
}
```

Cuidado no desenho da super-classe...

- Qual a interface a oferecer às sub-classes?
  - Idealmente, as sub-classes devem aceder apenas à super-classe através da sua interface pública
    - Preserva completamente a abstracção
    - Permite que a super-classe seja completamente re-implementada, sem que isso afecte as sub-classes
  - Na prática, essa interface pode não ser adequada para construir sub-classes eficientes
    - Nesse caso, a super-classe pode declarar variáveis, métodos e construtores como **protegidos** (visíveis para as sub-classes)
      - Mas isso também os torna visíveis dentro do package

Como tornar membros visíveis para as sub-classes?

- Declarar a visibilidade como **`protected`**
  - Os membros protegidos são visíveis em:
    - todas as classes no mesmo package
    - todas as sub-classes, quer estejam no mesmo package ou em outros packages
- Os membros protegidos são introduzidos para permitir implementações mais eficientes das classes
  - Pode haver variáveis de instância protegidas
  - As variáveis de instância podem ser privadas mas ter métodos de acesso (**`gets`** e **`sets`**) protegidos
    - Esta segunda abordagem é melhor, se permitir preservar invariantes da super-classe

Representação de um objecto da sub-classe

- Inclui as variáveis de instância declaradas na super-classe e as declaradas na sub-classe

Representação na sub-classe **MountainBike**

```
protected int speed;      // herdada da super-classe
protected int cadence;    // herdada da super-classe
protected int gear;       // herdada da super-classe

private int seatHeight;   // criada na sub-classe
```

Com acesso directo (declaradas em `BicycleClass`)

Com acesso directo (declaradas em `MountainBike`)

- O **acesso directo** a detalhes de representação da super-classe apenas é possível se a super-classe tornar partes da sua implementação acessíveis às sub-classes

Desvantagens dos membros protegidos

- Devemos **evitar** o uso de membros protegidos a menos que tenhamos um bom motivo
  - Sem eles, a implementação da super-classe pode ser alterada sem afectar as sub-classes
  - Dado que os membros protegidos são visíveis em todo o package, isso constitui uma quebra no encapsulamento da classe
    - Corre-se o risco do código de outras classes da package pode interferir com a implementação da super-classe

Que classes herdam o quê?


```
public class A {
    protected int x, y ;
    private int z ;
    public String m() { z = 0 ; return this.p() ; }
    private void a() { }
    public String p() { return "P do A"; }
}

public class B extends A {
    public float y ;
    public void z() { super.p() ; }
    public String p() { return "P do B"; }
}

public class C extends B {
    private double y ;
    public String p() { return "P do C"; }
    public void a() { y = super.y + ((A)this).y ; }
}
```

- m() de A herdado pelas classes B e C
- z() de B herdado pela classe C
- a() de A não é herdado por ser privado
- p() de A não é herdado por ser redefinido em B
- p() de B não é herdado por ser redefinido em C
- a() de C não é considerado redefinição de a() de A, porque a() de A era privado
  - Só se redifinem métodos não privados
- z() de B acede ao método redefinido p() de A, usando a palavra reservada super
- a() de C tem dois acessos indirectos a variáveis ocultas

Acesso a variáveis herdadas

- Todas as variáveis de instância são herdadas pelas sub-classes, mas...
  - A sub-classe não tem acesso às variáveis privadas
  - A sub-classe apenas tem acesso indirecto às variáveis não privadas que são redefinidas (ocultas)
    - **Relembrar**: as variáveis **ocultas** (*shadowed*) são de evitar
      - Uma variável oculta da super-classe imediata pode ser acedida com o identificador `super`
      - Uma variável de outra super-classe mais acima na hierarquia pode ser acedida usando um *cast* de `this` para o tipo que essa super-classe constitui

```
public void a() { y = super.y + ((A)this).y ; }
```

O identificador super

- Existe uma forma especial de acesso a métodos redefinidos e atributos ocultos da super-classe desde que eles estejam na *super-classe imediata*
  - Para tal, usa-se o identificador **`super`**

```
public class A {
    //...
    public String p() { return "P do A"; }
}

public class B extends A {
    public float y ;
    public void z() { super.p() ; }
    //...
}

public class C extends B {
    private double y ;
    public String p() { return "P do A"; }
    public void a() { y = super.y + ((A)this).y ; }
}
```

Que variáveis têm os objectos?


```
public class A {
    protected int x, y ;
    private int z ;
    public String m() { z = 0 ; return this.p() ; }
    private void a() { }
    public String p() { return "P do A"; }
}

public class B extends A {
    public float y ;
    public void z() { super.p() ; }
    public String p() { return "P do B"; }
}

public class C extends B {
    private double y ;
    public String p() { return "P do C"; }
    public void a() { y = super.y + ((A)this).y ; }
}
```

- Objectos de A:
  - x
  - y
  - z
- Objectos de B:
  - x
  - super.y
  - ~~z~~
  - y
- Objectos de C:
  - x
  - ((A)this).y
  - ~~z~~
  - super.y
  - y

`this` e a reinterpretação dos métodos herdados nas sub-classes

- Quando um método é herdado, o seu corpo é reinterpretado nas sub-classes (recorde que o método **`p()`** estava redefinido nas sub-classes)

```
public String m() { z = 0 ; return this.p() ; }
```

- Qual é o efe
  - `new A()`
  - `new B()`
  - `new C()`

```
public class A { ...
        public String p() { return "P do A"; }
}

public class B extends A { ...
        public String p() { return "P do B"; }
}

public class C extends B { ...
        public String p() { return "P do C"; }
}
```

`this` vs. `super`

- Dentro de um método, **`this`** e **`super`** representam o objecto do método, ou seja, o receptor da mensagem
- Ambas referem o mesmo objecto. A diferença está no modo como as mensagens são tratadas:
  - As mensagens enviadas para **`this`** invocam métodos da *classe real* do receptor da mensagem
    - Repare que isso é determinado de modo **dinâmico** – ver o slide anterior
  - As mensagens enviadas para **`super`** invocam métodos da *super-classe imediata* da classe onde a palavra **`super`** aparece escrita
    - Repare que isso é determinado de modo **estático** – ver implementação do método **`z()`** da classe **`B`**

Checkpoint!

- Uma classe pode estender/especializar outra classe, usando o mecanismo de herança
  - A palavra reservada **`extends`** indica essa relação
- A sub-classe herda definições de métodos e atributos da super-classe
  - Mas nem todas são directamente acessíveis...
- A sub-classe pode redefinir métodos e atributos da super-classe

Conjuntos de inteiros

$\mathbb{ZZZZZZZZ}$

Conjuntos de inteiros

- Os conjuntos de inteiros têm diversas aplicações em muitos domínios
  - Números do totoloto
  - Números de telefone
  - ...
- Pretendemos construir diversos tipos de conjuntos de inteiros, para depois escolher o mais adequado consoante as situações
  - Em particular, queremos
    - Um conjunto simples de inteiros
    - Um conjunto de inteiros em que seja fácil saber qual o maior
    - Um conjunto de inteiros ordenado

O nosso programa deve permitir

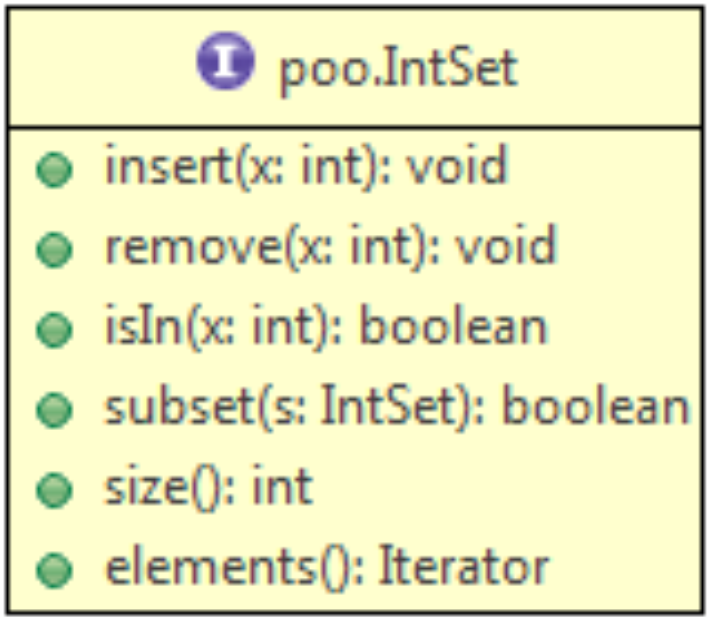
- Criar um conjunto de inteiros
  - Simples, ou com a funcionalidade extra de encontrar rapidamente o máximo
- Acrescentar números inteiros a um conjunto
- Remover um número inteiro do conjunto
- Testar se um número pertence ao conjunto
- Testar a relação de subconjunto
- Listar os elementos de um conjunto

Vamos definir uma família de conjuntos de inteiros

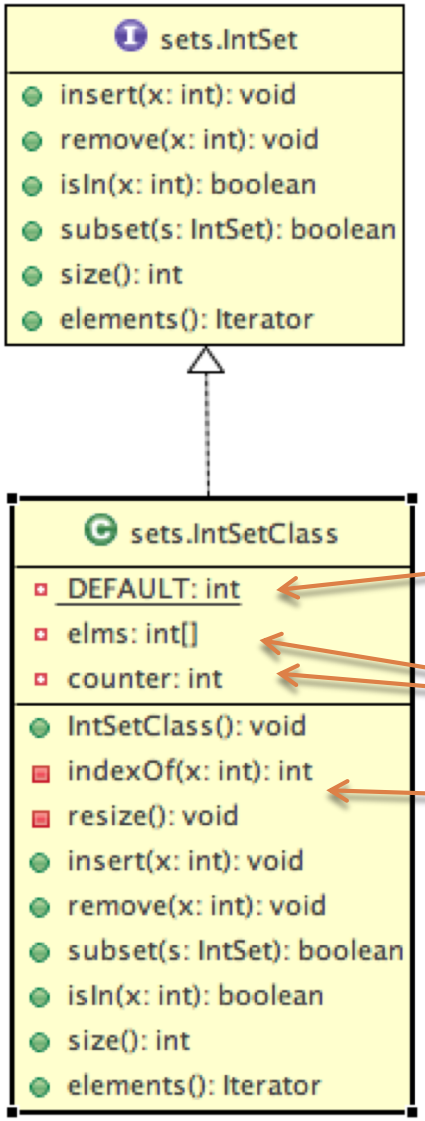
- `IntSet` fornece um conjunto adequado de métodos para conjuntos alteráveis, não limitados, de inteiros
  - **`public void`** `insert(int x)`
    - Acrescenta o inteiro `x` ao conjunto **PRE:** `!isIn(x)`
  - **`public void`** `remove(int x)`
    - Remove o inteiro `x` do conjunto **PRE:** `isIn(x)`
  - **`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 int`** `size()`
    - Retorna o tamanho do conjunto
  - **`public`** `Iterator elements()`
    - Retorna um iterador de inteiros, para suportar as listagens

A interface IntSet


```
// 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();
}
```

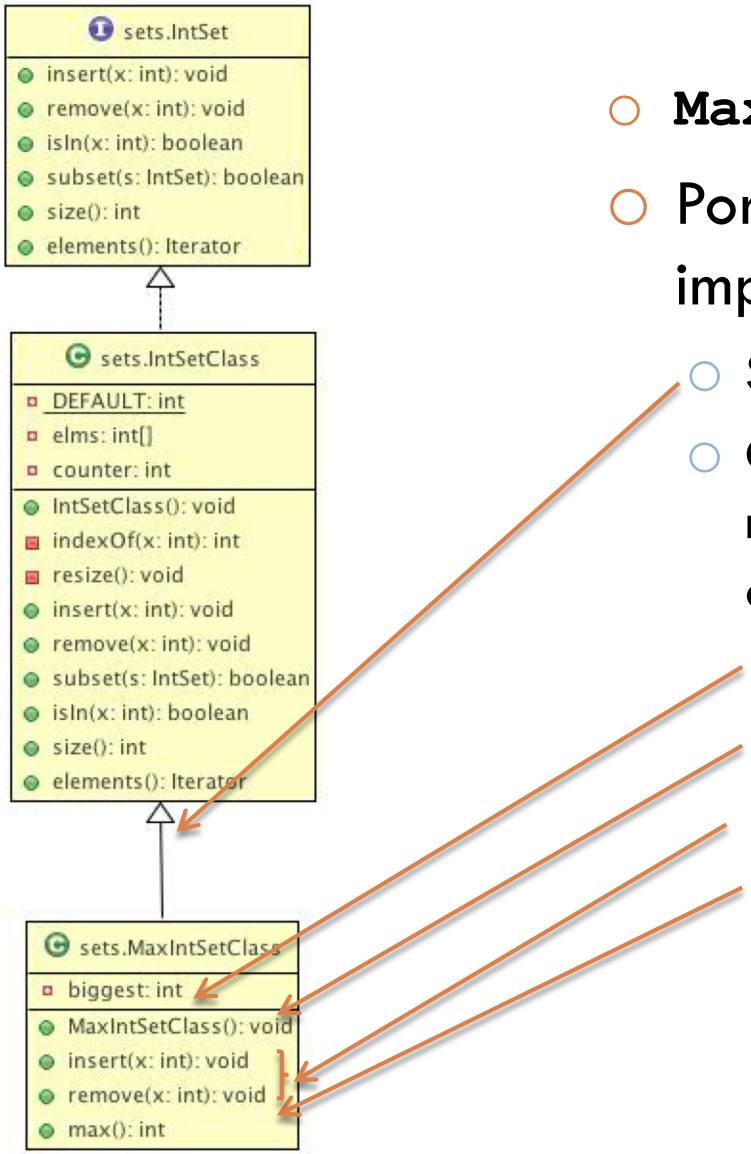


A classe IntSetClass



- 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
    - Dois métodos auxiliares, privados, já nossos conhecidos...

Uma família de conjuntos de inteiros

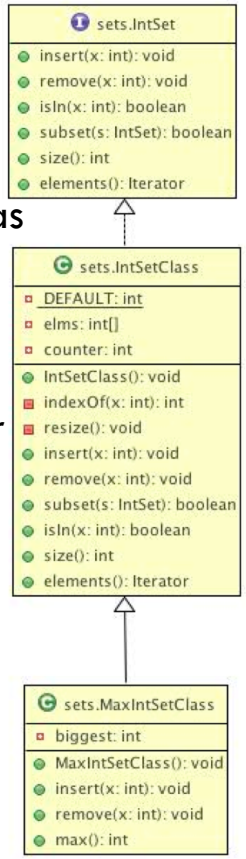


- **`MaxIntSet`** é uma sub-classe de **`IntSetClass`**
- Por ser sub-classe de **`IntSetClass`**, também implementa a interface **`IntSet`**:
  - Símbolo de herança
  - Comportamento semelhante ao de **`IntSetClass`**, mas com um método extra **`max`** que retorna o maior elemento do conjunto
    -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

Especificação de `IntSetClass`


```
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() {...}
}
```

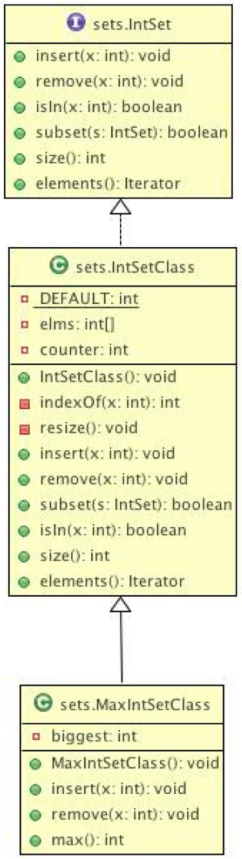
Não são usados membros protegidos, neste exemplo. Isto significa que as sub-classes de IntSetClass apenas lhe podem aceder através da sua interface pública. O nível de acesso é aceitável, porque o iterador permite visitar todos os elementos da colecção.



Implementação de `IntSetClass`

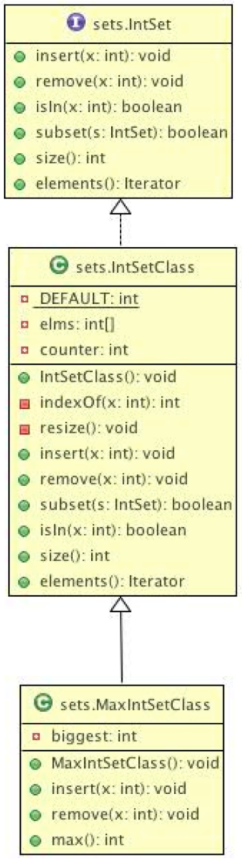

```
public class IntSetClass implements IntSet {
  private static final int DEFAULT = 10;
  private int[] elms;
  private int counter;

  public IntSetClass(){...}
  private int indexOf(int x) {...}
  private void resize() {...}
  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 os método `indexOf` e `resize` são privados.
São inacessíveis fora desta classe, mesmo para as sub-classes.



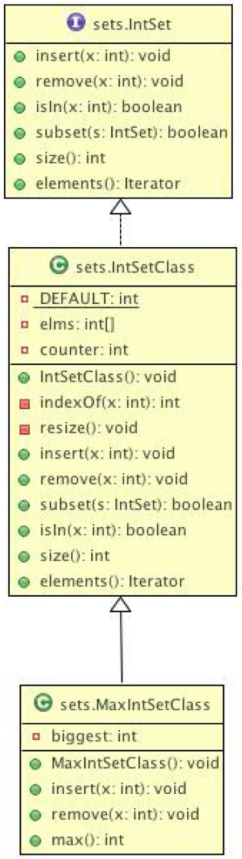
Implementação de `IntSetClass`


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



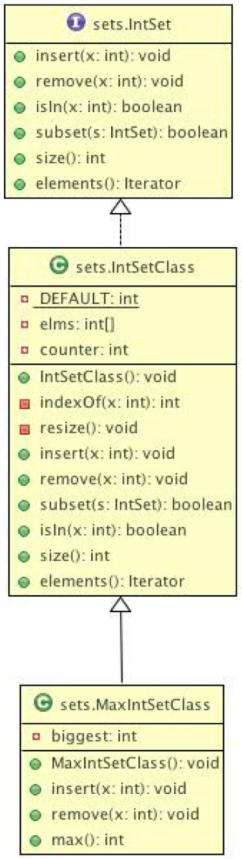
Implementação de `IntSetClass`


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

private void resize() {
  int[] tmp = new int[elms.length*2];
  for (int i = 0; i < counter; i++)
      tmp[i] = elms[i];
  elms = tmp;
}

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



Implementação de `IntSetClass`

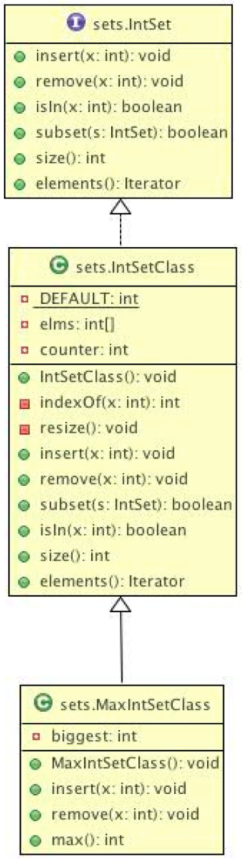

```
  public boolean subset(IntSet s) {
    if (s.size() < this.size()) return false;
    for (int i = 0; i < counter; i++)
      if (!s.isIn(elms[i]))
        return false;
    return true;
  }

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

  public int size(){
    return counter;
  }

  public Iterator elements() {
    return new IteratorClass(elms, counter);
  }
}
```



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 serem 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`**

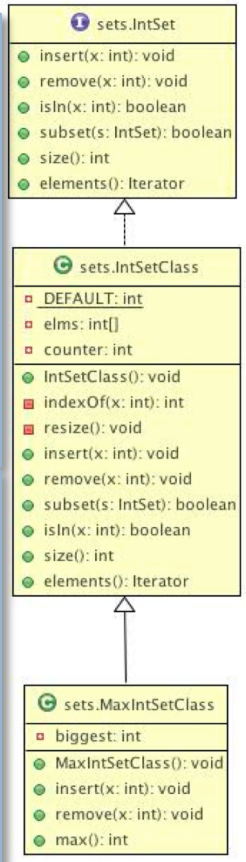


Implementação de `MaxIntSetClass`


```
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);
  }
```

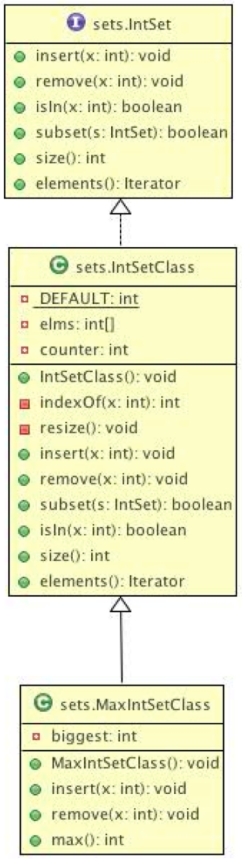
Como em todos os construtores devemos inicializar todas as variáveis de instância. Para garantir que as variáveis inacessíveis da super-classe também são inicializadas, usa-se a chamada ao construtor da super-classe!

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.



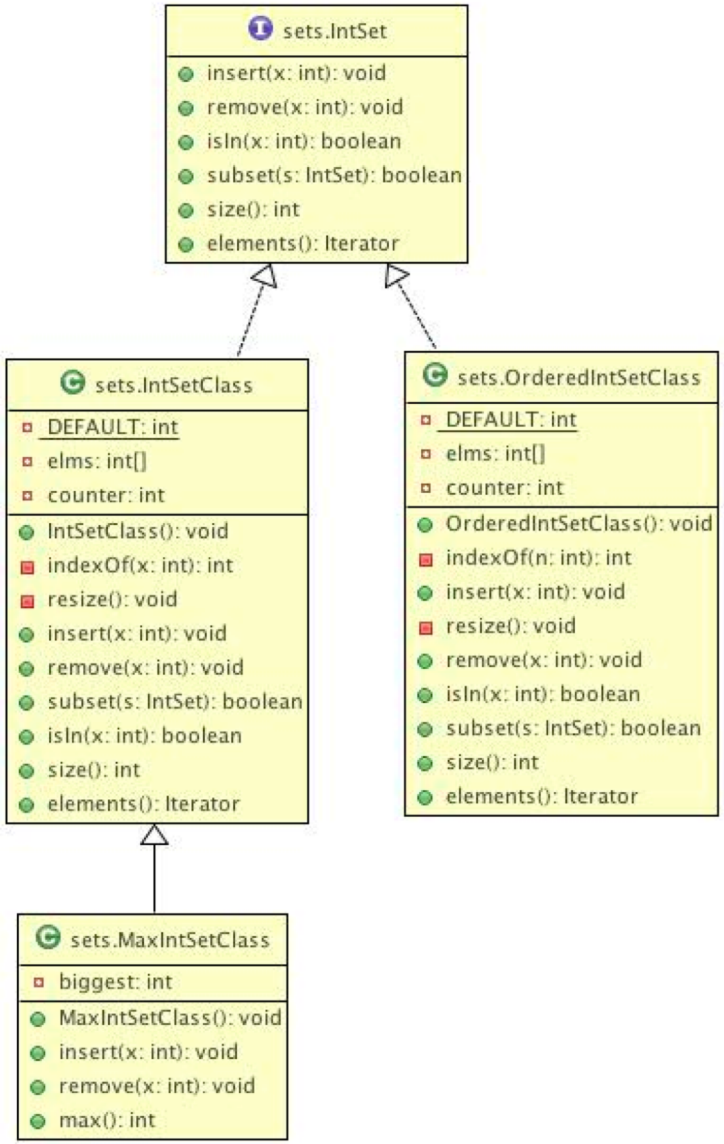
Implementação de `MaxIntSetClass`


```
  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; }
}
```



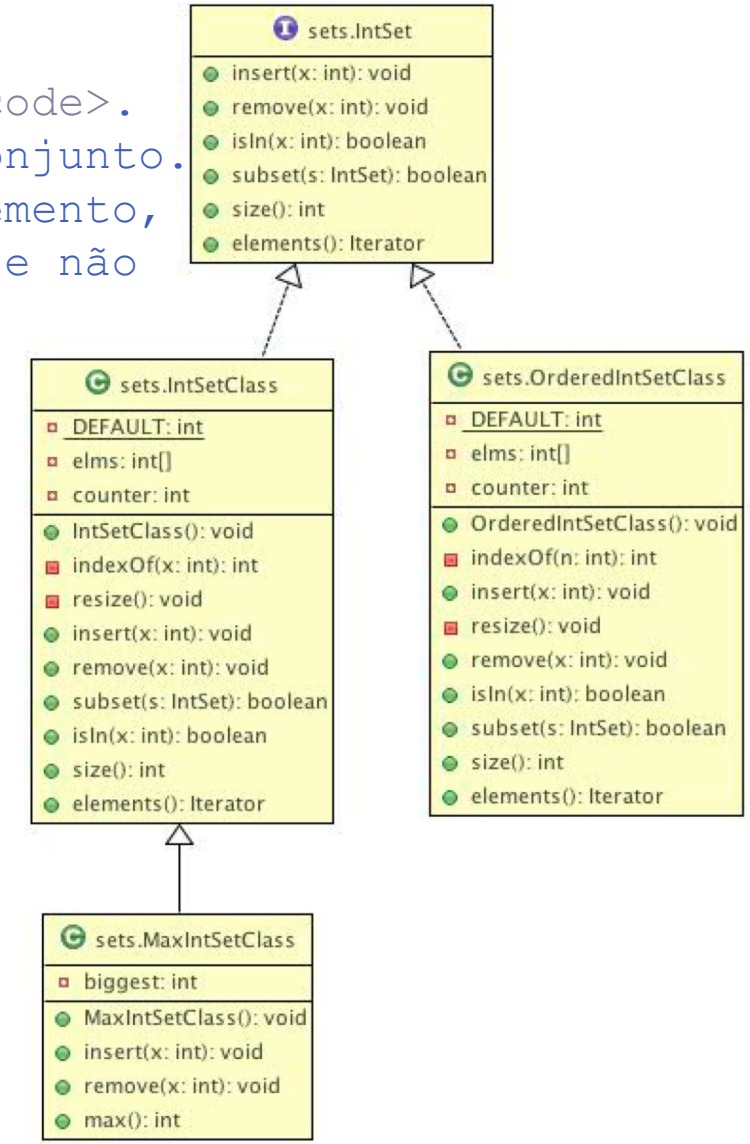
Implementação de `OrderedIntSetClass`


```
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;
  }
```



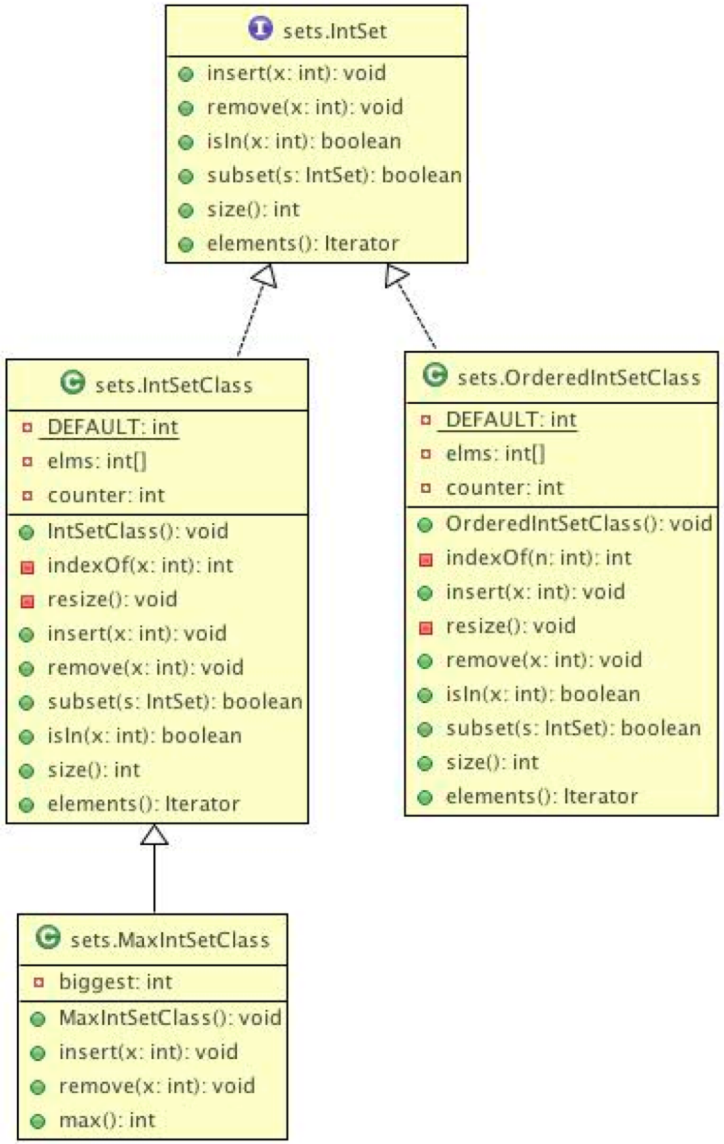
Implementação de `OrderedIntSetClass`


```
/**
 * 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 o índice do primeiro inteiro maior se não
 * existir.
 */
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;
}
```



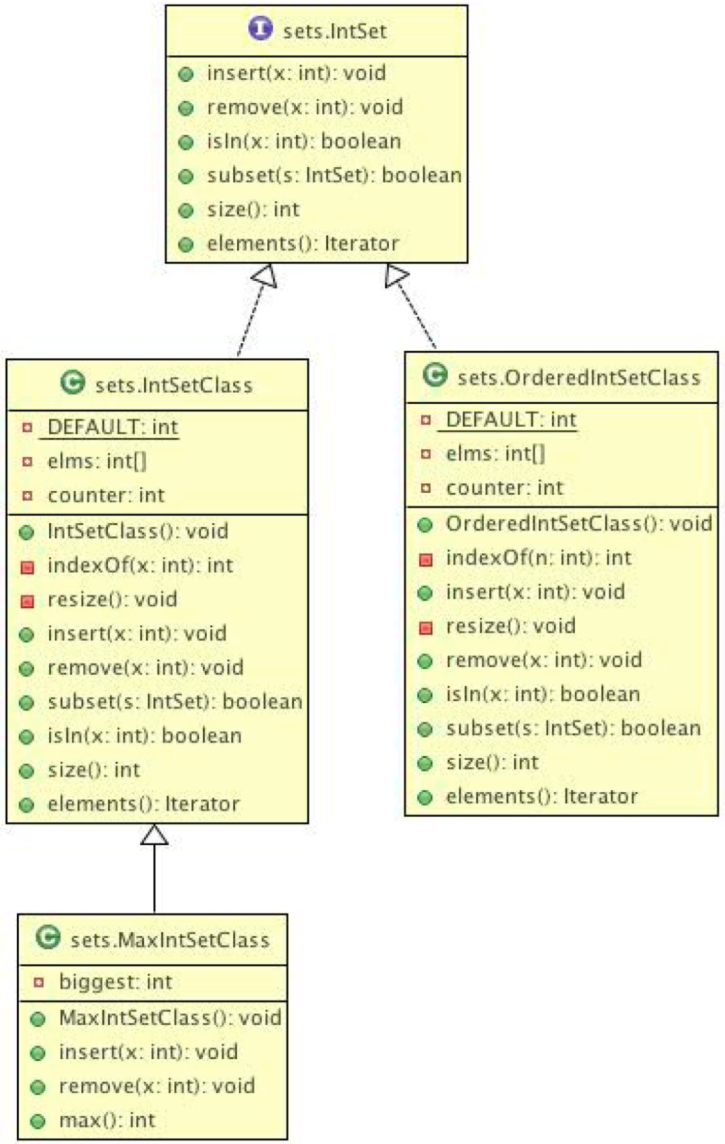
Implementação de `OrderedIntSetClass`


```
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++;
}

private void resize() {
  int[] tmp = new int[elms.length*2];
  for (int i = 0; i < counter; i++)
      tmp[i] = elms[i];
  elms = tmp;
}
```



Implementação de `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;
}
```



Implementação de `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);
  }
}
```