

Programação Orientada Pelos Objectos

Herança de classes



O que é um **subtipo**?

2

- Sejam **A** e **B** tipos
- **B** é subtipo de **A** se toda a expressão do tipo **A** puder ser substituída por uma expressão de tipo **B**, sem que se registre qualquer incompatibilidade
- Por outras palavras, **B** é subtipo de **A** se toda a expressão de tipo **B** puder ser usada onde se espera uma expressão de tipo **A**

+
lógica

Para que serve um **subtipo**?

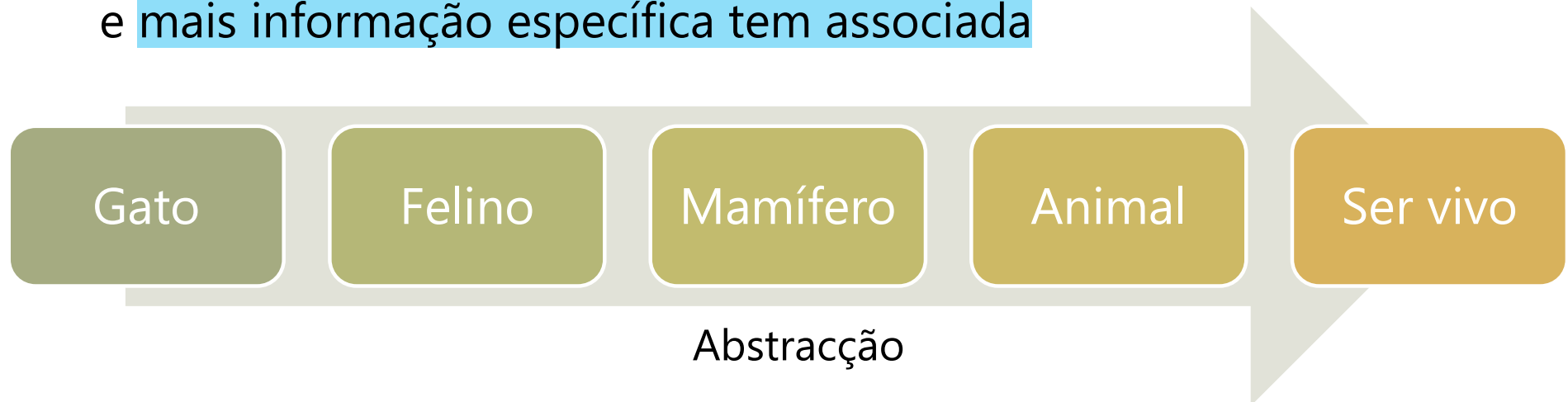
3

- Permitem organizar em níveis de abstracção os conceitos usados nos programas
 - Permitem tratar de objectos de um tipo mais específico como se fossem objectos de um tipo mais geral
- Uma linguagem com subtipos adapta-se bem à forma de pensar dos programadores
 - Os seres humanos estão habituados a pensar e a falar de coisas usando abstracções

Abstracção

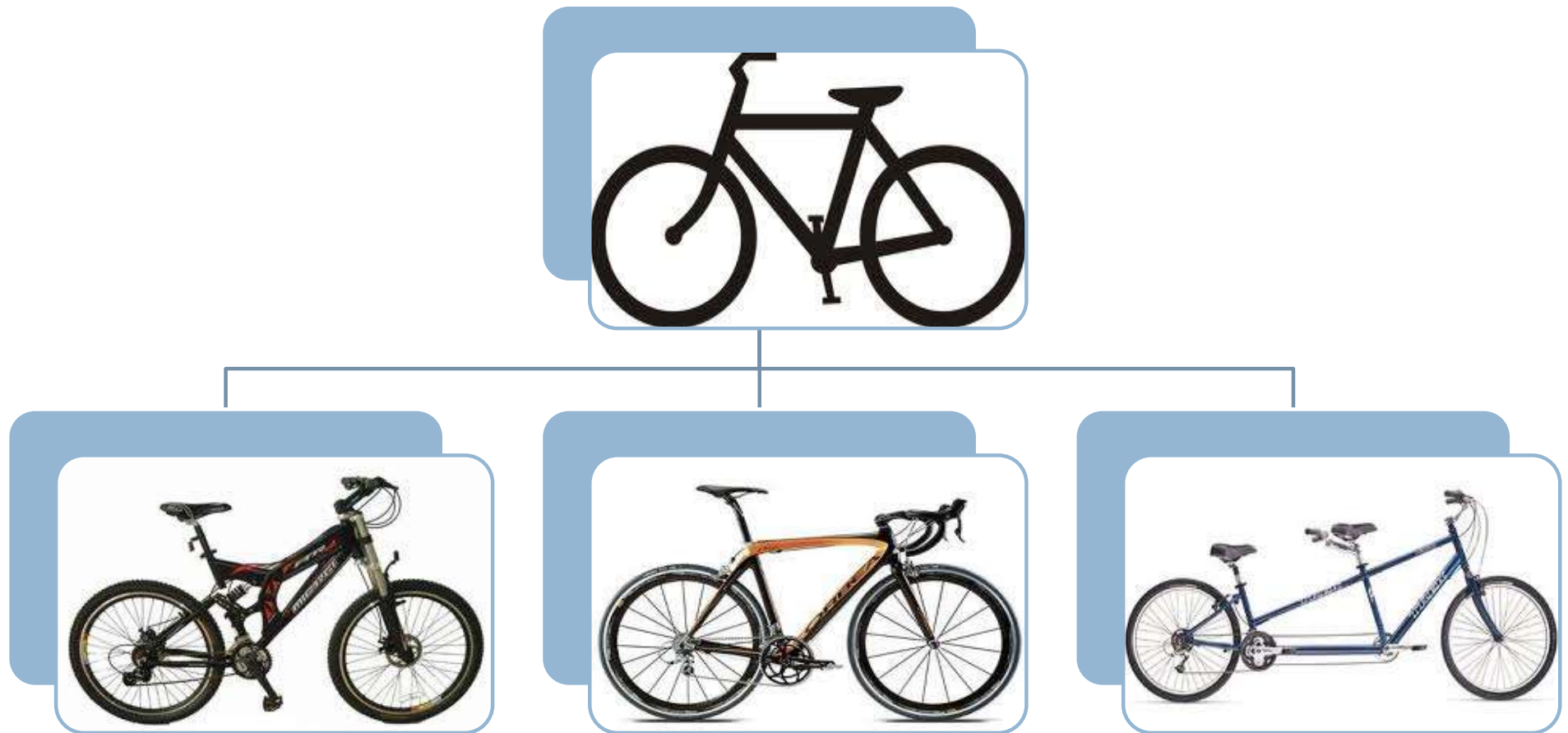
4

- **Abstrair** significa ignorar, deliberadamente, aspectos particulares de uma entidade, ou conceito, com o objectivo de enfatizar os restantes aspectos, considerados de maior relevância
- Por exemplo, Animal é mais abstracto que Gato, porque ignora deliberadamente mais informação
- Quanto mais abstracto é um conceito, mais elementos representa, e menos informação específica tem associada
- Quanto mais concreto é um conceito, menos elementos representa e mais informação específica tem associada



Famílias de entidades relacionadas

5



Famílias?

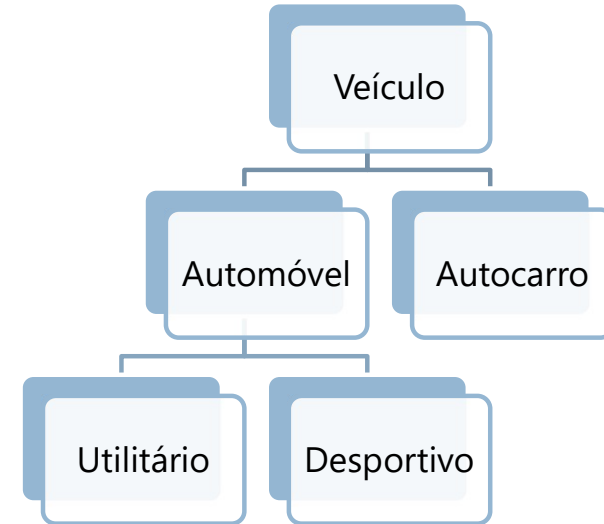
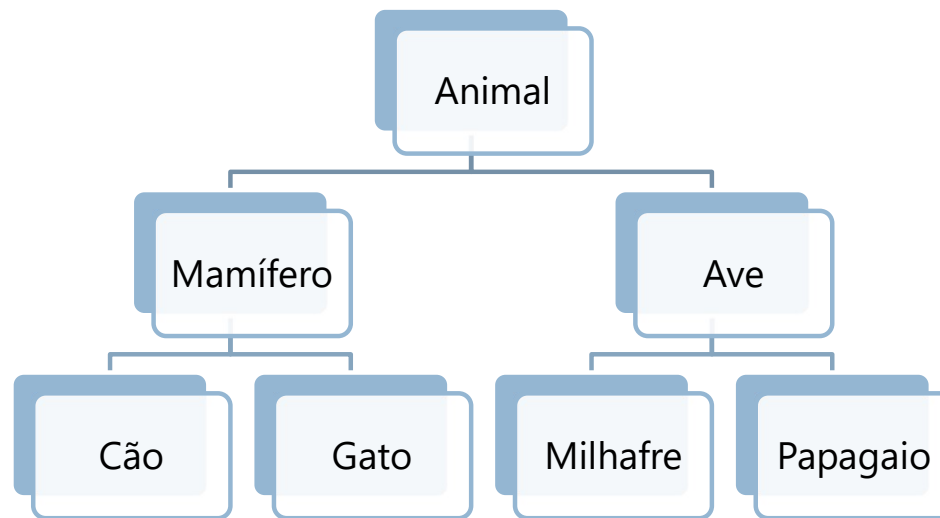
6

- Como tirar partido da abstracção de dados definindo famílias de tipos relacionados?
 - Os membros de uma família têm algum comportamento comum
 - Têm um conjunto de métodos obedecendo ao mesmo protocolo
 - À partida, chamadas a esses métodos resultam nos mesmos comportamentos
 - Contudo, elementos da família podem divergir...
 - Estendendo alguns dos comportamentos comuns da família
 - Acrescentando novos comportamentos

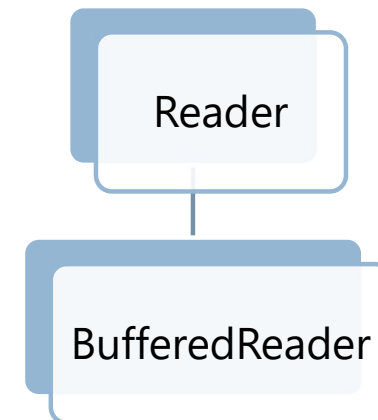
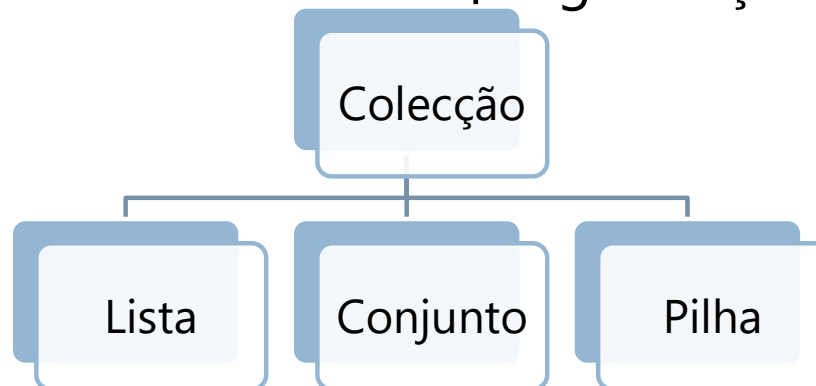
Famílias!

7

- A família pode corresponder a uma hierarquia do mundo real



- Ou no mundo da programação



Família de tipos

8

- Uma família de tipos é definida por uma hierarquia de tipos
 - No topo da hierarquia, está um tipo (de dados ;-)
que define o protocolo comum a todos os membros da família
 - Os outros elementos da família são **subtipos** deste tipo que está no topo da hierarquia: o **supertipo**

Famílias de tipos usadas de duas formas

9

- Famílias usadas para fornecer diferentes implementações de um tipo
- Famílias cujos subtipos estendem o protocolo do seu supertipo

Famílias usadas para fornecer diferentes implementações de um tipo

10

- Neste caso, os subtipos não alteram o protocolo do supertipo, com exceção de que cada tipo terá o seu respectivo construtor
- A classe que implementa o subtipo implementa exactamente o protocolo definido pelo supertipo

Famílias cujos subtipos estendem o protocolo do seu supertipo

11

- 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 subtipo
→ um subtipo pode ter mais métodos que o supertipo

Hierarquia de tipos

12

- Define uma família de tipos consistindo num supertipo e nos seus subtipos
- **Princípio da substituição**
 - Proposto por Barbara Liskov e Jeannette Wing
 - Os subtipos comportam-se de acordo com a especificação dos seus supertipos

Para que serve uma hierarquia de tipos?

13

- Permite “relaxar” a verificação de tipos em certos pontos do programa
- Um programa aceita objectos de tipos diferentes dos declarados, desde que o novo tipo seja um subtipo do tipo declarado
- Serve de suporte à **polimorfia**: a capacidade de um objecto ser de múltiplos tipos simultaneamente

Para que serve uma hierarquia de tipos?

14

- 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
- Um programa aceita objectos de tipos diferentes dos declarados, desde que o novo tipo seja um subtipo do tipo declarado
- Serve de suporte à **polimorfia**: a capacidade de um objecto ser de múltiplos tipos simultaneamente

Tipo declarado vs tipo real

15

- Uma variável pode ser declarada como pertencendo a um tipo, mas na realidade referir-se a um objecto que é de um subtipo 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 subtipo de **Animal**
 - Não devemos confundir o **tipo declarado** com o tipo usado na instanciação – o **tipo real**

Verificação de tipos pelo compilador

16

- O compilador verifica os tipos com base na informação para ele disponível
 - Usa sempre os tipos *declarados*, 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

um processo no qual uma variável é atribuída a um tipo específico de objeto durante sua declaração para criar um objeto vinculado antecipadamente

Dispatching

17

- 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

substituição de um método virtual
como C++ ou à implementação de
uma interface

Como definir uma hierarquia

18

- A especificação do supertipo é, frequentemente, incompleta
 - Por exemplo, pode não ter construtores
- A especificação de subtipos é feita relativamente à especificação dos supertipos
 - Foco no que o subtipo tem de novo
 - Tipicamente, acrescenta construtores do subtipo
 - Métodos adicionais
 - Se o subtipo alterar a especificação de métodos definidos no supertipo, então tem de fornecer a nova especificação desses métodos
 - Há limites para o tipo de alterações permitidas (já voltaremos aqui)

Implementação da hierarquia

19

- Por vezes, os supertipos não são implementados de todo, ou apenas são parcialmente implementados
- Se o supertipo é implementado, ainda que parcialmente, o subtipo é uma **extensão** da implementação do supertipo
 - A implementação do subtipo pode **herdar** variáveis de instância e métodos do supertipo
 - A implementação do subtipo pode também **redefinir** os métodos herdados

Implementação da hierarquia

20

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

Definição de hierarquias, em Java

21

- Utilização do mecanismo de **herança**
 - Este mecanismo permite que uma classe seja uma subclasse de outra classe (a superclasse) e que implemente zero ou mais interfaces
- Supertipos 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 supertipo

Relação de herança entre classes

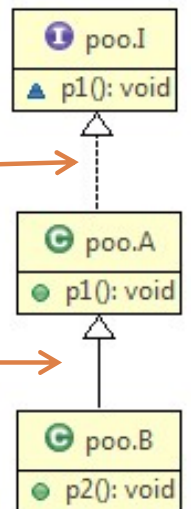
22

- Usando o mecanismo de herança, o programador consegue criar uma nova classe (subclasse) com base numa classe já existente (superclasse)
 - Terá apenas de definir as componentes da subclasse que são adicionadas, ou modificadas, face à superclasse
 - As componentes da superclasse 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 ();
}

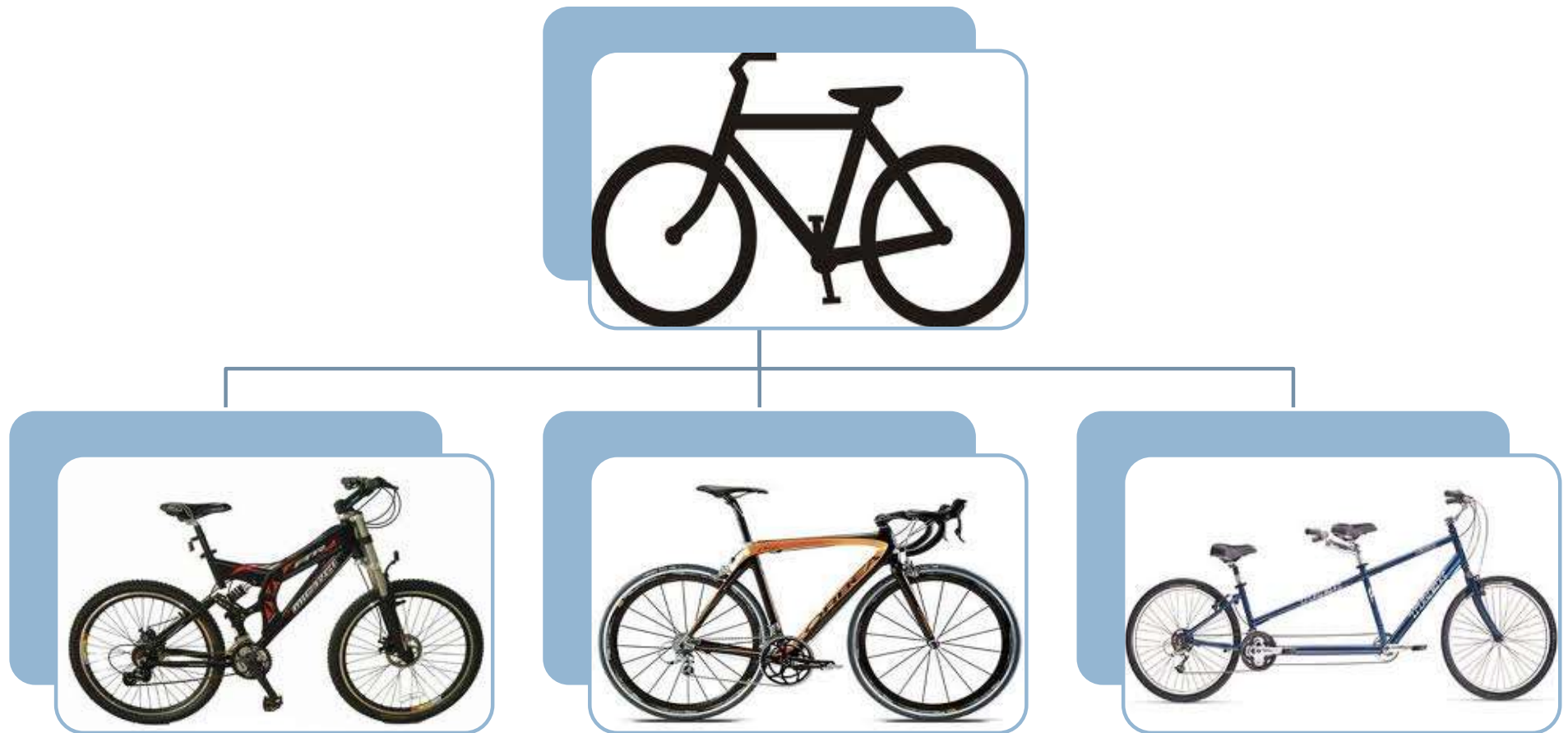
public class A implements I { // superclasse
    public void p1 () { }
}

public class B extends A { // subclasse
    public void p2 () { } // B herda o método p1 () de A
}
```

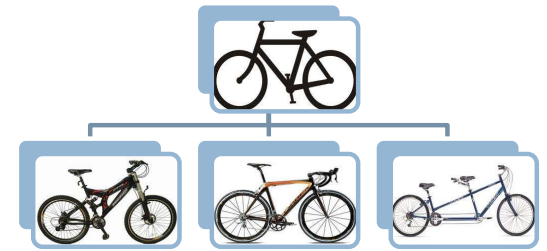


Declaração de classes e subclasses: Bicicletas

23



Classe BicycleClass



24

```
public class BicycleClass implements Bicycle{
```

```
    private int speed;  
    private int cadence;  
    private int gear;
```

3 variáveis de
instância

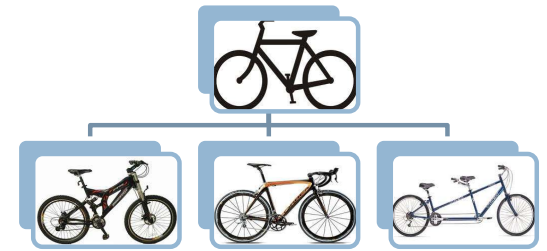
```
    public BicycleClass(int startSpeed, int startCadence, int startGear) {  
        speed = startSpeed;  
        cadence = startCadence; // rotações por minuto  
        gear = startGear;  
    }
```

```
    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;  
    }  
}
```

Construtor

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

Subclasse MountainBikeClass



25

```
public class MountainBikeClass extends BicycleClass {
    private int seatHeight; // Altura do assento;
    /**
     * Construtor de MountainBike
     * @param startHeight - altura do assento
     * @param startSpeed - velocidade inicial
     * @param startCadence - cadência da pedalada
     * @param startGear - mudança inicial
     */
    public MountainBikeClass(int startHeight,
                             int startSpeed, int startCadence, int startGear) {
        super(startSpeed, startCadence, 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 subclasse tem uma nova variável de instância

A subclasse tem o seu próprio construtor que invoca o construtor da superclasse, para fazer a inicialização correspondente à superclasse

A subclasse tem novos métodos

Declaração de uma subclasse

26

- Uma subclasse declara a sua superclasse indicando no cabeçalho da sua definição que estende (**extends**) essa classe
- Uma subclasse apenas pode estender uma superclasse



O que é que a subclasse herda?

27

- **Membros** (atributos, métodos e classes aninhadas) da superclasse que sejam **públicos** ou **protegidos**, com os mesmos nomes e assinaturas
- **Membros** com a visibilidade **package** (a visibilidade por omissão), **se a superclasse estiver no mesmo package**

A subclasse pode acrescentar e modificar membros

28

- Pode acrescentar membros novos
- Pode **reimplementar**, ou **redefinir** (*override*), os métodos da superclasse
 - Estes métodos têm de ter assinaturas compatíveis com as definidas na superclasse
 - 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 ter visibilidade de package ou pública

Shadowing de variáveis de instância

29

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

Ocultação de métodos na subclasse

30

- Pode escrever um novo método `static` na subclasse com a mesma assinatura de um método estático da superclasse, **ocultando** assim o método da superclasse

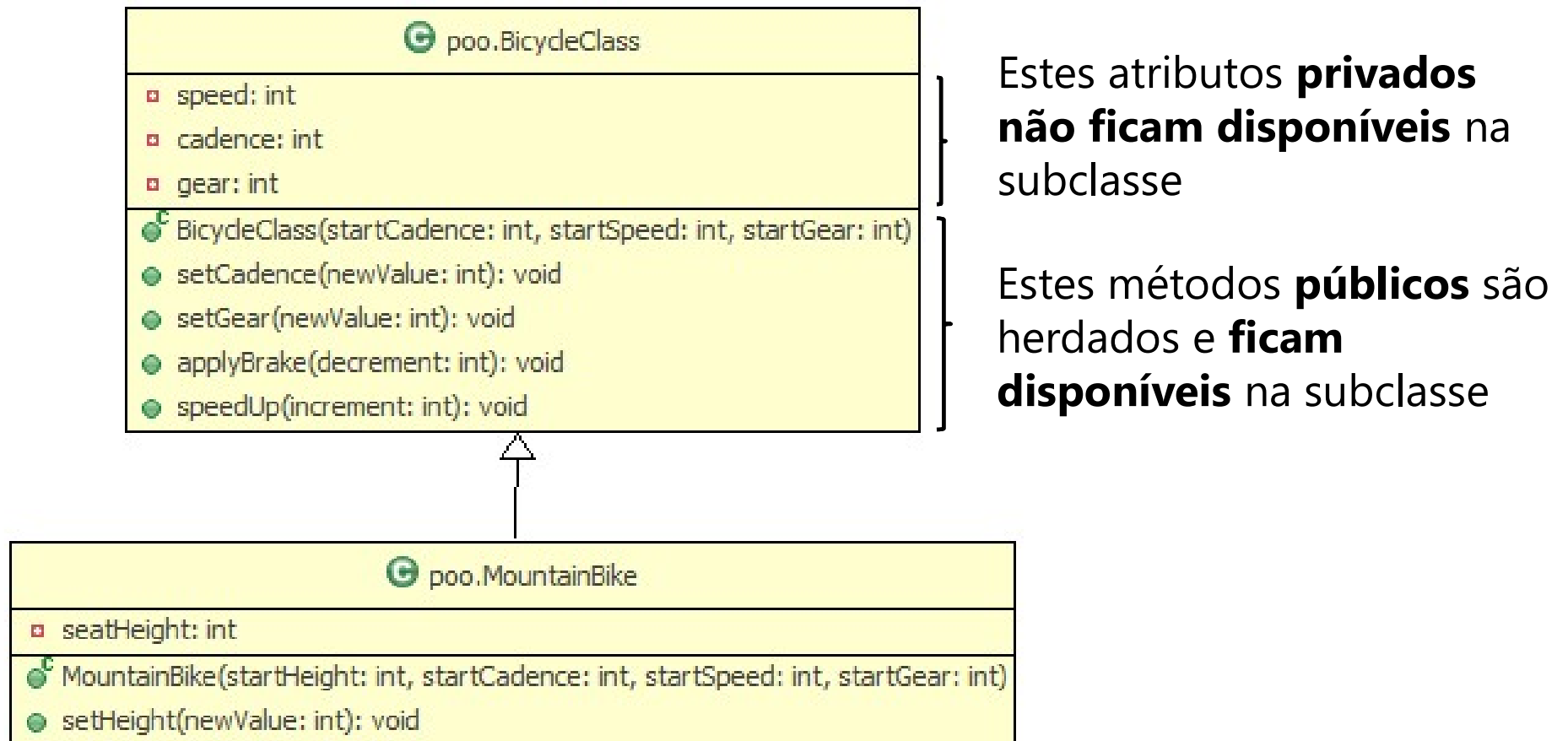
O que é que a subclasse **não** herda?

31

- **Membros** da superclasse que sejam **privados**, **redefinidos**, ou **escondidos não são herdados**
- **Construtores e blocos de inicialização**
 - No caso de acrescentar novas variáveis de instância, estas devem ser inicializadas nos seus próprios construtores
 - O construtor da subclasse pode invocar o construtor da superclasse implicitamente, ou usando a palavra reservada `super`

O que cabe na herança?

32



Representação de um objecto da subclasse

33

- Inclui as variáveis de instância declaradas na superclasse e as declaradas na subclasse

Representação na subclasse **MountainBikeClass**

```
private int speed; // definida na superclasse  
private int cadence; // definida na superclasse  
private int gear; // definida na superclasse
```

Sem acesso directo
(declaradas em
BicycleClass)

```
private int seatHeight; // criada na subclasse
```

Com acesso directo
(declaradas em
MountainBikeClass)

- O **acesso directo** a detalhes de representação da superclasse apenas é possível se a superclasse tornar partes da sua implementação acessíveis às subclasses

O que acontece aos membros de instância privados?

34

- A subclasse **não tem acesso** a membros privados da superclasse!

```
// Algures na classe MountainBikeClass...
```

```
public void slide() {
```

```
    this.applyBrake(1);
```

```
    // Além de travar, faz mais qualquer coisa para derrapar
```

```
}
```

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

Herança de métodos de instância

35

- Numa subclasse não existe acesso directo a:
 - métodos declarados na superclasse como privados
 - métodos da superclasse que sejam redefinidos na subclasse

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

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

Cuidado no desenho da superclasse...

36

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

Como tornar membros visíveis para as subclasses?

37

- Declarar a visibilidade como **protected**
 - Os membros protegidos são visíveis em:
 - todas as classes no mesmo package
 - todas as subclasses, 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 (e.g. **gets** e **sets**) protegidos
 - Esta segunda abordagem é melhor, se permitir preservar invariantes da superclasse

Representação de um objecto da subclasse

38

- Inclui as variáveis de instância declaradas na superclasse e as declaradas na subclasse

Representação na subclasse **MountainBikeClass**

```
protected int speed;      // herdada da superclasse  
protected int cadence;    // herdada da superclasse  
protected int gear;       // herdada da superclasse
```

Com acesso directo
(declaradas em BicycleClass)

```
private int seatHeight;   // criada na subclasse
```

Com acesso directo (declaradas em MountainBikeClass)

- O **acesso directo** a detalhes de representação da superclasse apenas é possível se a superclasse tornar partes da sua implementação acessíveis às subclasses

Desvantagens dos membros protegidos

39

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



Que classes herdaram o quê?

40

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

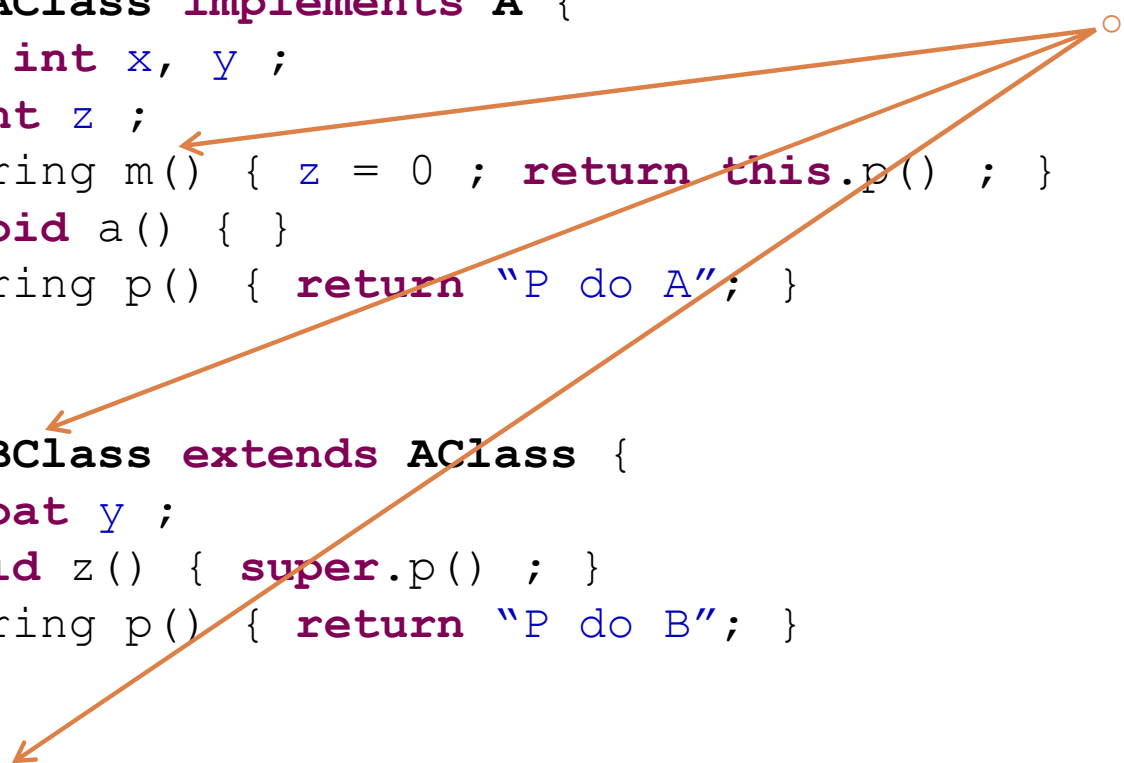
public class CClass extends BClass {
    private double y ;
    public String p() { return "P do C"; }
    public void a() { y = super.y + ((AClass)this).y ; }
}
```

Que classes herdam o quê?

41

```
public class AClass implements 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 BClass extends AClass {  
    public float y ;  
    public void z() { super.p() ; }  
    public String p() { return "P do B"; }  
}  
  
public class CClass extends BClass {  
    private double y ;  
    public String p() { return "P do C"; }  
    public void a() { y = super.y + ((AClass)this).y ; }  
}
```

m() de AClass herdado
pelas classes BClass e
CClass



Que classes herdam o quê?

42

```
public class AClass implements 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"; }  
}
```

○ m() de AClass herdado pelas classes BClass e CClass

○ z() de BClass herdado pela classe CClass

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

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

Que classes herdam o quê?

43

```
public class AClass implements 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"; }  
}
```

- m() de AClass herdado pelas classes BClass e CClass
- z() de BClass herdado pela classe CClass
- a() de AClass não é herdado por ser privado

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

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

Que classes herdam o quê?

44

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

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

- m() de AClass herdado pelas classes BClass e CClass
- z() de BClass herdado pela classe CClass
- a() de AClass não é herdado por ser privado
- p() de AClass não é herdado por ser redefinido em BClass

Que classes herdam o quê?

45

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

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

- m() de AClass herdado pelas classes BClass e CClass
- z() de BClass herdado pela classe CClass
- a() de AClass não é herdado por ser privado
- p() de AClass não é herdado por ser redefinido em BClass
- p() de BClass não é herdado por ser redefinido em CClass

Que classes herdam o quê?

46

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

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

- m() de AClass herdado pelas classes BClass e CClass
- z() de BClass herdado pela classe CClass
- a() de AClass não é herdado por ser privado
- p() de AClass não é herdado por ser redefinido em BClass
- p() de BClass não é herdado por ser redefinido em CClass
- a() de CClass não é considerado redefinição de a() de AClass, porque a() de AClass era privado
 - Só se redefinem métodos não privados

Que classes herdam o quê?

47

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

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

- m() de AClass herdado pelas classes BClass e CClass
- z() de BClass herdado pela classe CClass
- a() de AClass não é herdado por ser privado
- p() de AClass não é herdado por ser redefinido em BClass
- p() de BClass não é herdado por ser redefinido em CClass
- a() de CClass não é considerado redefinição de a() de AClass, porque a() de AClass era privado
 - Só se redefinem métodos não privados
- z() de BClass acede ao método redefinido p() de AClass, usando a palavra reservada `super`

acessar os métodos da sua super class

Que classes herdam o quê?

48

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

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

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

Acesso a variáveis herdadas

49

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

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

O identificador super

50

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

```
public class AClass extends A {  
    //...  
    public String p() { return "P do A"; }  
}  
  
public class BClass extends AClass {  
    public float y ;  
    public void z() { super.p() ; }  
    //...  
}  
  
public class CClass extends BClass {  
    private double y ;  
    public String p() { return "P do A"; }  
    public void a() { y = super.y + ((AClass) this).y ; }  
}
```



Que variáveis têm os objectos?

51

```
public class AClass implements 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"; }  
}
```

○ Objectos de AClass:

- x
- y
- z

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

○ Objectos de BClass:

- x
- super.y
- ⊖ z
- y

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

○ Objectos de CClass:

- x
- ((AClass)this).y
- ⊖ z
- super.y
- y

this e a reinterpretação dos métodos herdados nas subclasses

52

- Quando um método é herdado, o seu corpo é reinterpretado nas subclasses

```
public class AClass implements A {  
    ...  
    public String m() { z = 0 ; return this.p() ;  
    public String p() { return "P do A"; }  
    ...  
}  
  
public class BClass extends AClass { ...  
    public String p() { return "P do B"; }  
}  
  
public class CClass extends BClass { ...  
    public String p() { return "P do C"; }  
}
```

- Qual é o efeito?

- new AClass().m()
 - "P do A"
- new BClass().m()
 - "P do B"
- new CClass().m()
 - "P do C"

this VS. super

53

- 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 *superclasse imediata* da classe onde a palavra **super** aparece escrita
 - Repare que isso é determinado de modo **estático** (**early binding**) – ver implementação do método **z ()** da classe **B**

Checkpoint!



54

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

Conjuntos de inteiros

ZZZZZZZZZZ

Conjuntos de inteiros

56

- 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

57

- 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

58

- `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 subconjunto 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 as listagens

A interface IntSet

59

// Comentários omitidos por economia de espaço no slide :-(

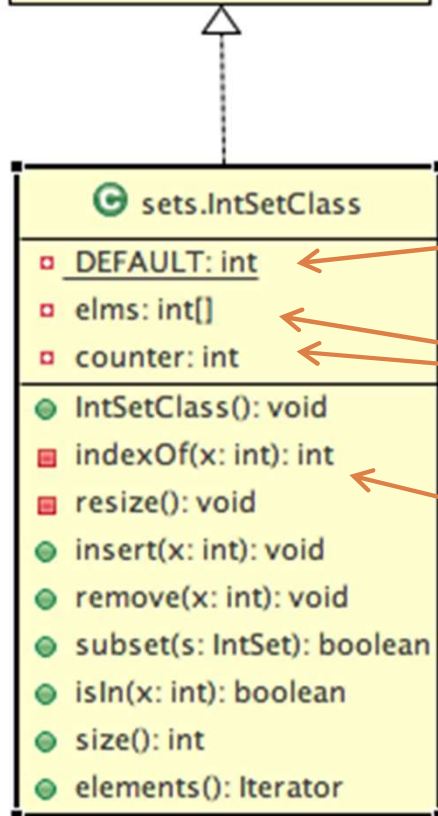
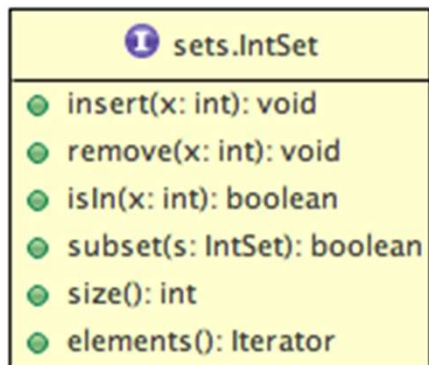
```
public interface IntSet {  
    void insert(int x);  
    void remove(int x);  
    boolean isIn(int x);  
    boolean subset(IntSet s);  
    int size();  
    Iterator elements();  
}
```

I 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`

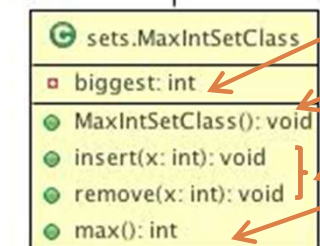
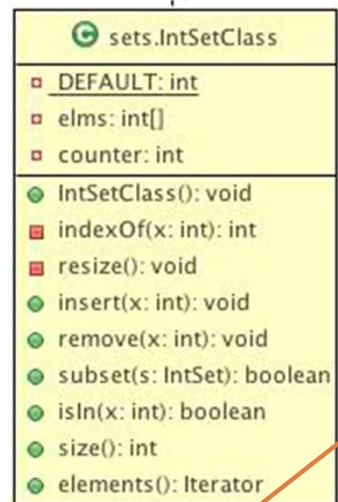
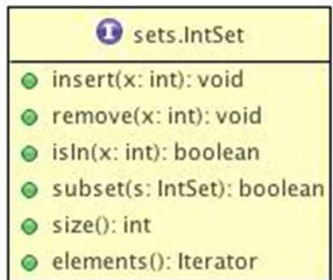
60



- 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

61



- **MaxIntSet** é uma subclasse de **IntSetClass**
- Por ser subclasse 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

62

```
public class IntSetClass implements IntSet {
```

```
    public IntSetClass() {}
```

```
    public void insert(int x) {...}
```

```
    public void remove(int x) {...}
```

```
    public boolean isIn(int x) {...}
```

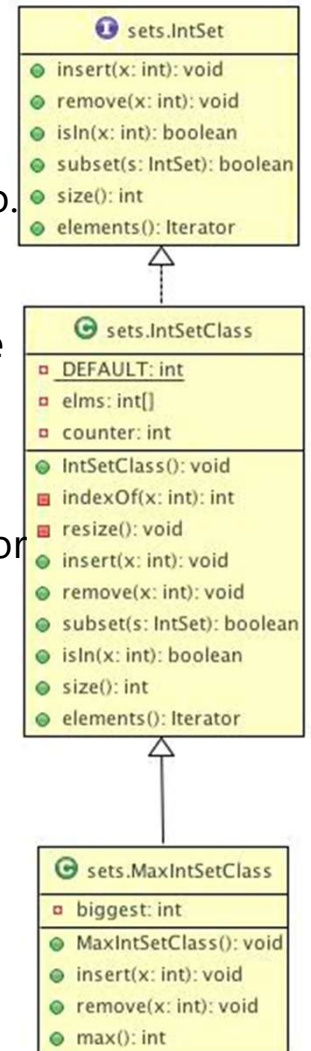
```
    public boolean subset(IntSet s) {...}
```

```
    public int size() {...}
```

```
    public Iterator elements() {...}
```

```
}
```

Não são usados membros protegidos, neste exemplo. Isto significa que as subclasses 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.

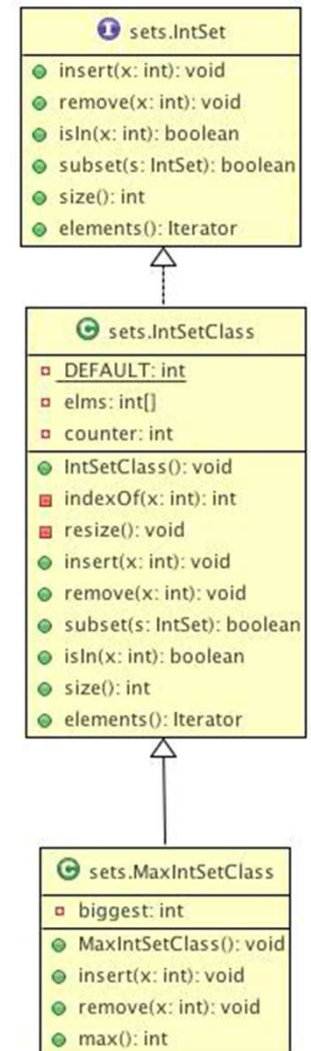


Implementação de IntSetClass

63

```
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 métodos `indexOf` e `resize` são privados. São inacessíveis fora desta classe, mesmo para as subclasses.



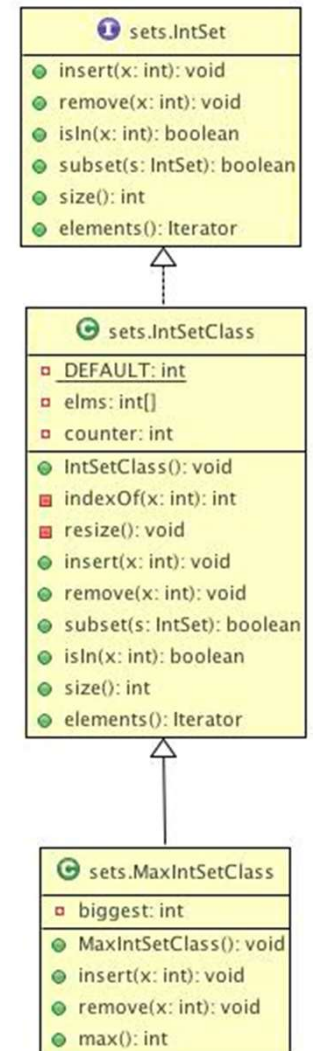
Implementação de IntSetClass

64

```
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;
    }

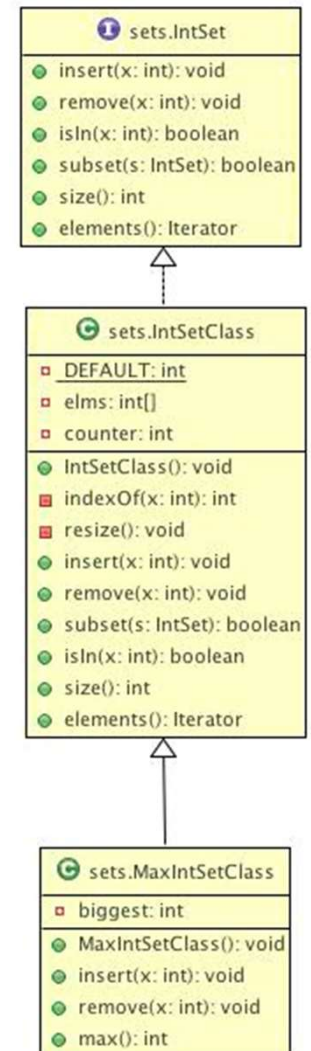
    private int indexOf(int x) {
        int i = 0, pos = -1;
        while (i < counter && pos == -1) {
            if (elms[i] == x)
                pos = i;
            i++;
        }
        return pos;
    }
    ...
}
```



Implementação de IntSetClass

65

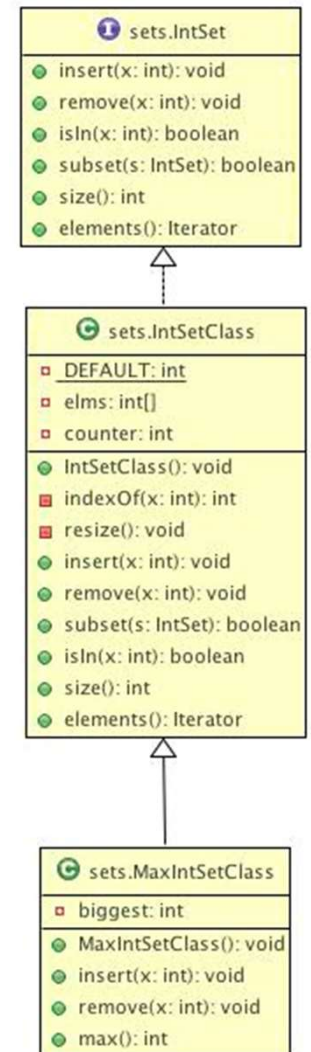
```
public class IntSetClass implements IntSet {  
    ...  
    public void insert(int x) {  
        if (counter == elms.length)  
            resize();  
        elms[counter++] = x;  
    }  
  
    private void resize() {  
        int[] tmp = new int[elms.length*2];  
        for (int i = 0; i < counter; i++)  
            tmp[i] = elms[i];  
        elms = tmp;  
    }  
}
```



Implementação de IntSetClass

66

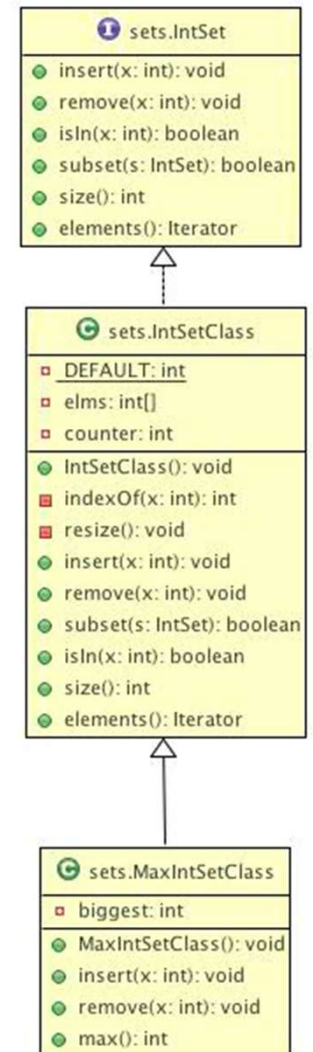
```
public class IntSetClass implements IntSet {  
    ...  
    public void insert(int x) {  
        if (counter == elms.length)  
            resize();  
        elms[counter++] = x;  
    }  
  
    private void resize() {  
        int[] tmp = new int[elms.length*2];  
        for (int i = 0; i < 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

67

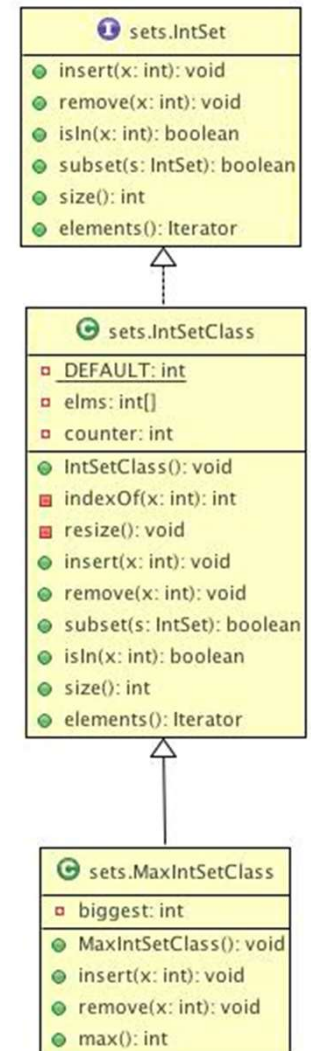
```
public class IntSetClass implements IntSet {  
    ...  
    public boolean isIn(int x) {  
        return (indexOf(x) != -1);  
    }  
}
```



Implementação de IntSetClass

68

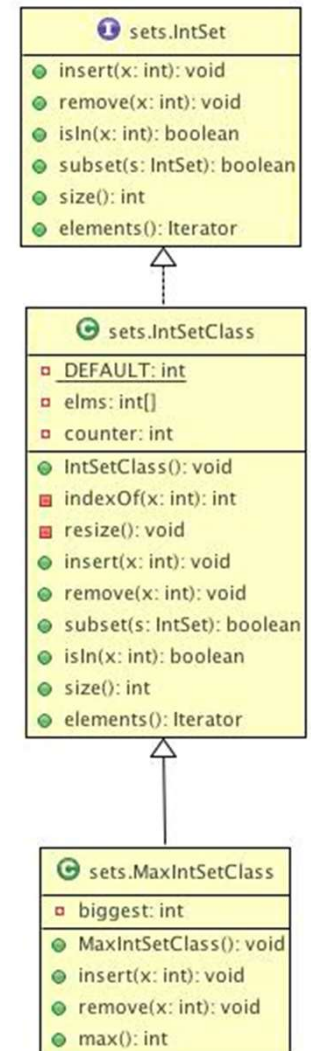
```
public class IntSetClass implements IntSet {  
    ...  
    public boolean isIn(int x) {  
        return (indexOf(x) != -1);  
    }  
  
    // Se this é subconjunto de s retorna true,  
    // caso contrário, retorna false  
    public boolean subset(IntSet s) {  
        if (s.size() < this.size()) return false;  
        for (int i = 0; i < counter; i++)  
            if (!s.isIn(elms[i]))  
                return false;  
        return true;  
    }  
}
```



Implementação de IntSetClass

69

```
public class IntSetClass implements IntSet {  
    ...  
    public boolean isIn(int x) {  
        return (indexOf(x) != -1);  
    }  
  
    // Se this é subconjunto de s retorna true,  
    // caso contrário, retorna false  
    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);  
    }  
}
```



Implementação de IntSetClass

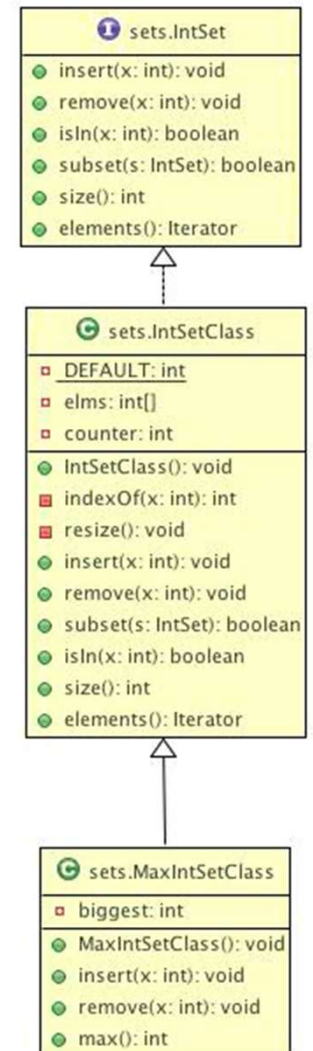
70

```
public class IntSetClass implements IntSet {
    ...
    public boolean isIn(int x) {
        return (indexOf(x) != -1);
    }

    // Se this é subconjunto de s retorna true,
    // caso contrário, retorna false
    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);
    }
}
```



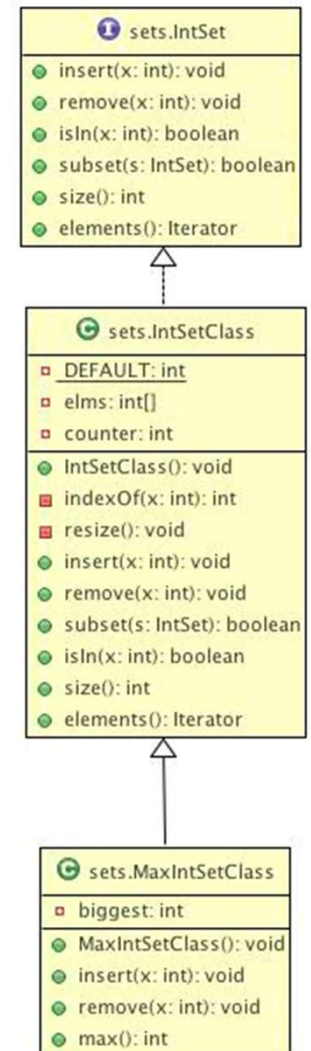
Especificação de `MaxIntSetClass`

71

```
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 privada, não é herdada
- As duas variáveis de instância da superclasse, `elms` e `counter`, não podem ser acedidas directamente por serem privadas
- O construtor da superclasse não é herdado
- O método `indexOf` da superclasse não pode ser acedido directamente, 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`**



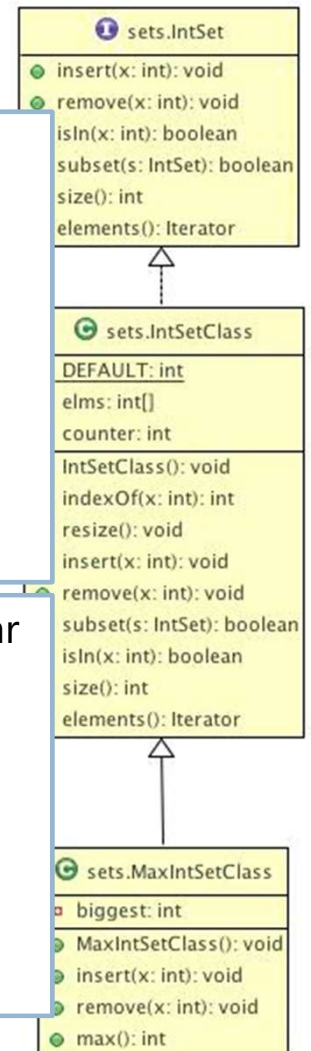
Implementação de MaxIntSetClass

72

```
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 superclasse também são inicializadas, usa-se a chamada ao construtor da superclasse!

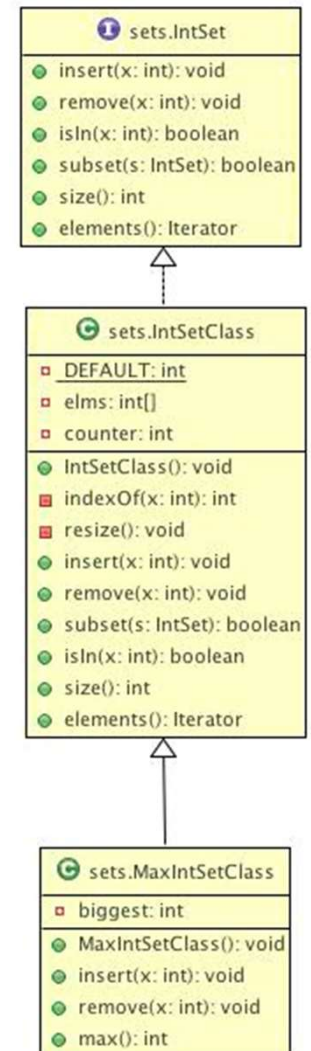
O insert começa por tratar do caso especial levantado pela necessidade de actualizar a variável biggest. No resto, seria igual ao da superclasse, portanto, delega nela a implementação.



Implementação de MaxIntSetClass

73

```
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; }  
}
```

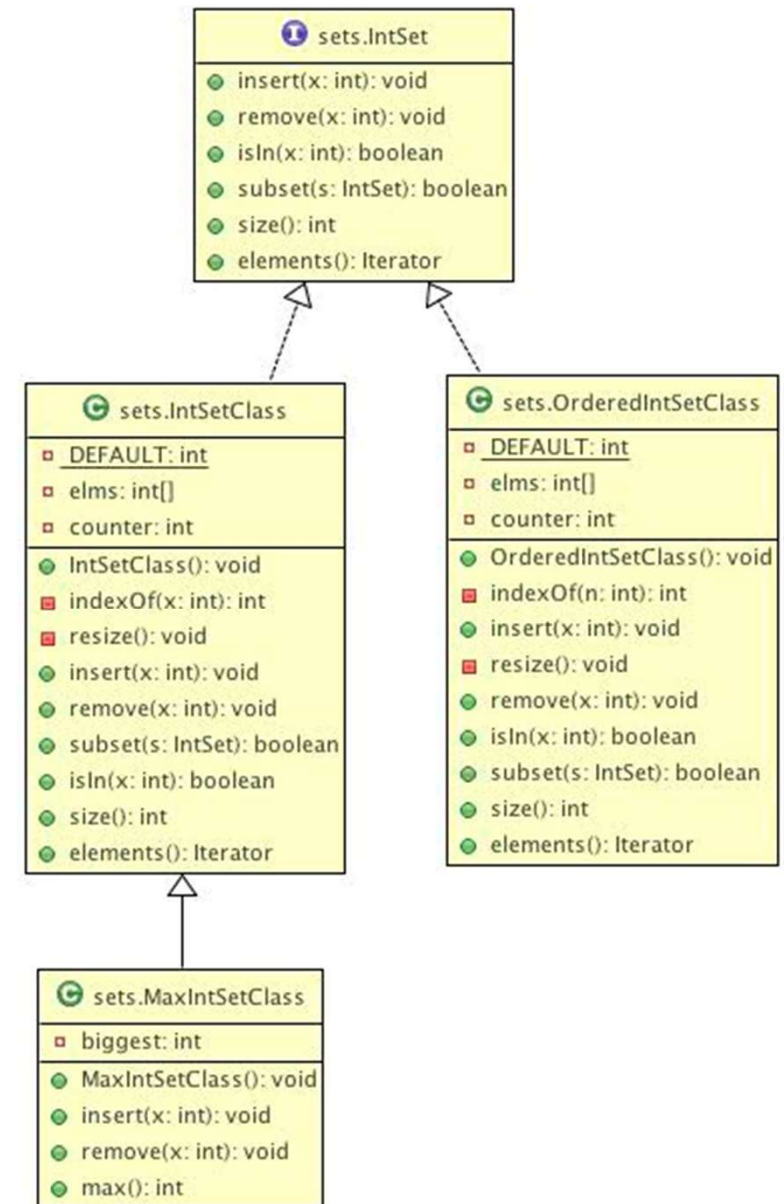


Implementação de OrderedIntSetClass

74

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

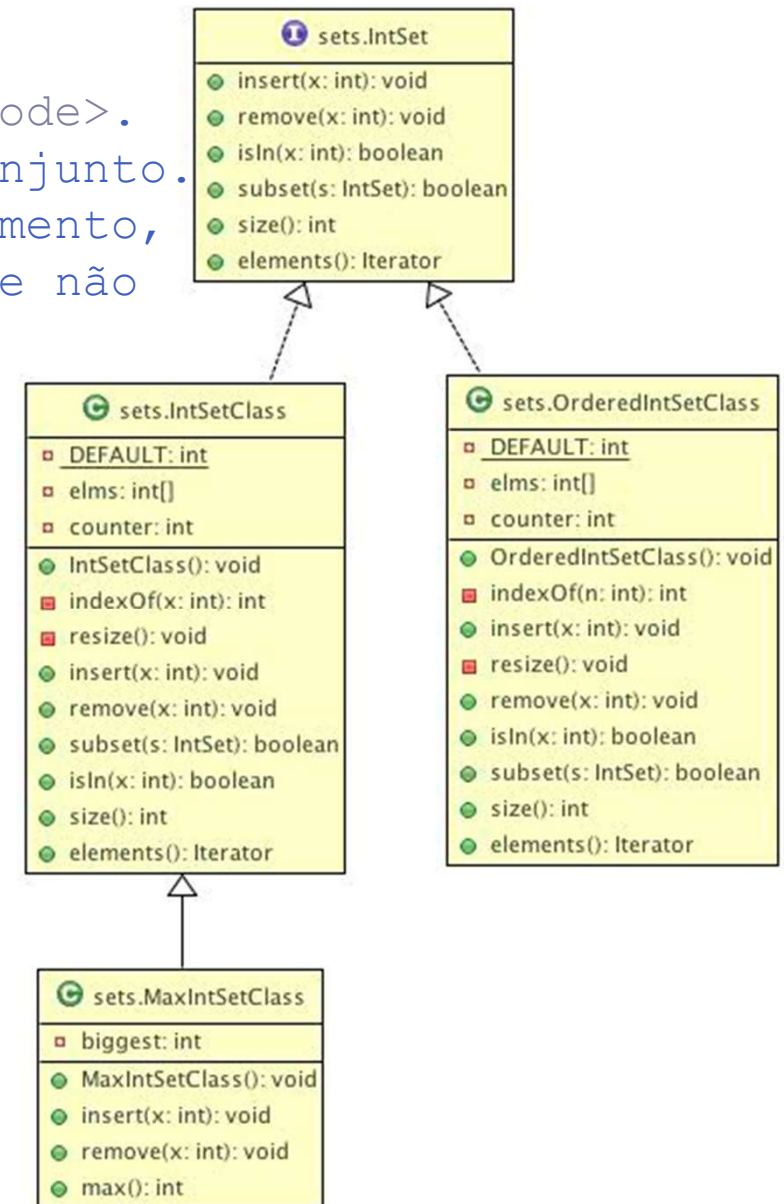
    /**
     * Inicializa o vector acompanhado,
     * de modo a representar
     * um conjunto vazio de inteiros.
     */
    public OrderedIntSetClass () {
        elms = new int[DEFAULT];
        counter = 0;
    }
}
```



Implementação de OrderedIntSetClass

75

```
/**
 * 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;
}
```



Execução de indexOf (10)

76

```
► 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;  
}
```

```
elms = [3, 9, 13, 14, 17, ...]  
counter = 5
```

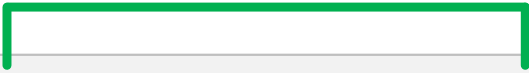
n	low	mid	high
10			

Execução de indexOf (10)

77

```
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;  
}
```

elms = [3, 9, 13, 14, 17, ...]
counter = 5



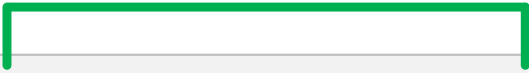
n	low	mid	high
10	0	-1	4

Execução de indexOf (10)

78

```
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;  
}
```

elms = [3, 9, 13, 14, 17, ...]
counter = 5



n	low	mid	high
10	0	-1	4

Execução de indexOf (10)

79

```
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;  
}
```

elms = [3, 9, 13, 14, 17, ...]
counter = 5

n	low	mid	high
10	0	-1	4
		2	

Execução de indexOf (10)

80

```
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;  
}
```

elms = [3, 9, 13, 14, 17, ...]
counter = 5



n	low	mid	high
10	0	-1	4
		2	

Execução de indexOf (10)

81

```
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;  
}
```

elms = [3, 9, 13, 14, 17, ...]
counter = 5

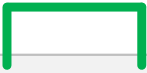


n	low	mid	high
10	0	-1	4
		2	1

Execução de indexOf (10)

82

```
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;  
}
```

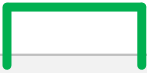

elms = [3, 9, 13, 14, 17, ...]
counter = 5

n	low	mid	high
10	0	-1	4
		2	1

Execução de indexOf (10)

83

```
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;  
}
```


elms = [3, 9, 13, 14, 17, ...]
counter = 5


n	low	mid	high
10	0	-1	4
		2	1
		0	

Execução de indexOf (10)

84

```
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;  
}
```

elms = [3, 9, 13, 14, 17, ...]
counter = 5



n	low	mid	high
10	0	-1	4
		2	1
		0	

Execução de indexOf (10)

85

```
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;  
}
```

□
elms = [3, 9, 13, 14, 17, ...]
counter = 5

n	low	mid	high
10	0	-1	4
		2	1
	1	0	

Execução de indexOf (10)

86

```
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;  
}
```

□
elms = [3, 9, 13, 14, 17, ...]
counter = 5

n	low	mid	high
10	0	-1	4
		2	1
	1	0	

Execução de indexOf (10)

87

```
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;  
}
```

elms = [3, 9, 13, 14, 17, ...]
counter = 5


n	low	mid	high
10	0	-1	4
		2	1
	1	0	
		1	

Execução de indexOf (10)

88

```
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;  
}
```

elms = [3, 9, 13, 14, 17, ...]
counter = 5



n	low	mid	high
10	0	-1	4
		2	1
	1	0	
		1	

Execução de indexOf (10)

89

```
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;  
}
```

```
elms = [3, 9, 13, 14, 17, ...]  
counter = 5
```

n	low	mid	high
10	0	-1	4
		2	1
	1	0	
	2	1	

Execução de indexOf (10)

90

```
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;  
}
```

```
elms = [3, 9, 13, 14, 17, ...]  
counter = 5
```

n	low	mid	high
10	0	-1	4
		2	1
	1	0	
	2	1	

Execução de indexOf (10)

91

```
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;  
}
```

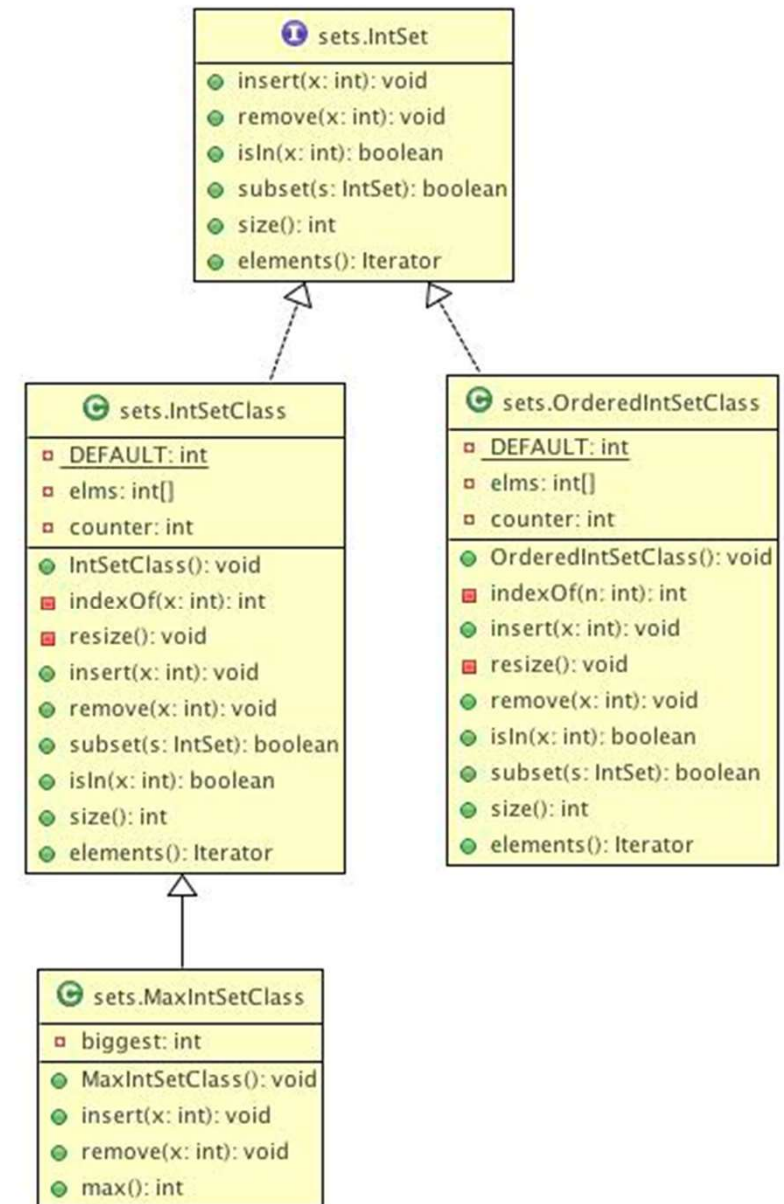
```
elms = [3, 9, 13, 14, 17, ...]  
counter = 5
```

n	low	mid	high
10	0	-1	4
		2	1
	1	0	
	2	1	

Implementação de OrderedIntSetClass

92

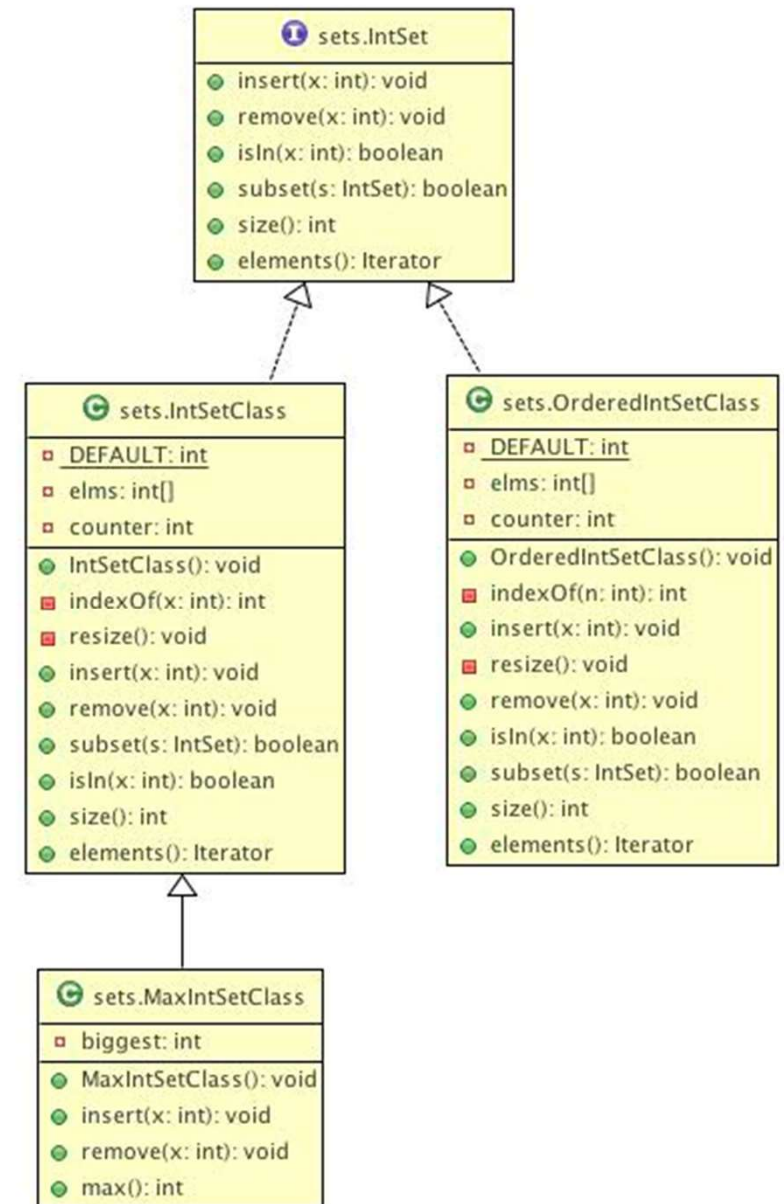
```
public void insert(int x) {  
    int pos = indexOf(x);  
    if (counter == elms.length)  
        resize();  
    for (int i = counter; i > pos; i--)  
        elms[i] = elms[i-1];  
    elms[pos] = x;  
    counter++;  
}  
  
private void resize() {  
    int[] tmp = new int[elms.length*2]; 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



Implementação de OrderedIntSetClass

93

```
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

94

```
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);  
}  
}
```

