PROGRAMAÇÃO ORIENTADA PELOS OBJECTOS

Aula 7

Herança de Classes

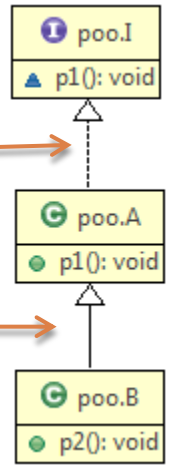
… Continuando a aula passada…

- Famílias de tipos
- Hierarquias de classes
- Classes concretas e abstractas

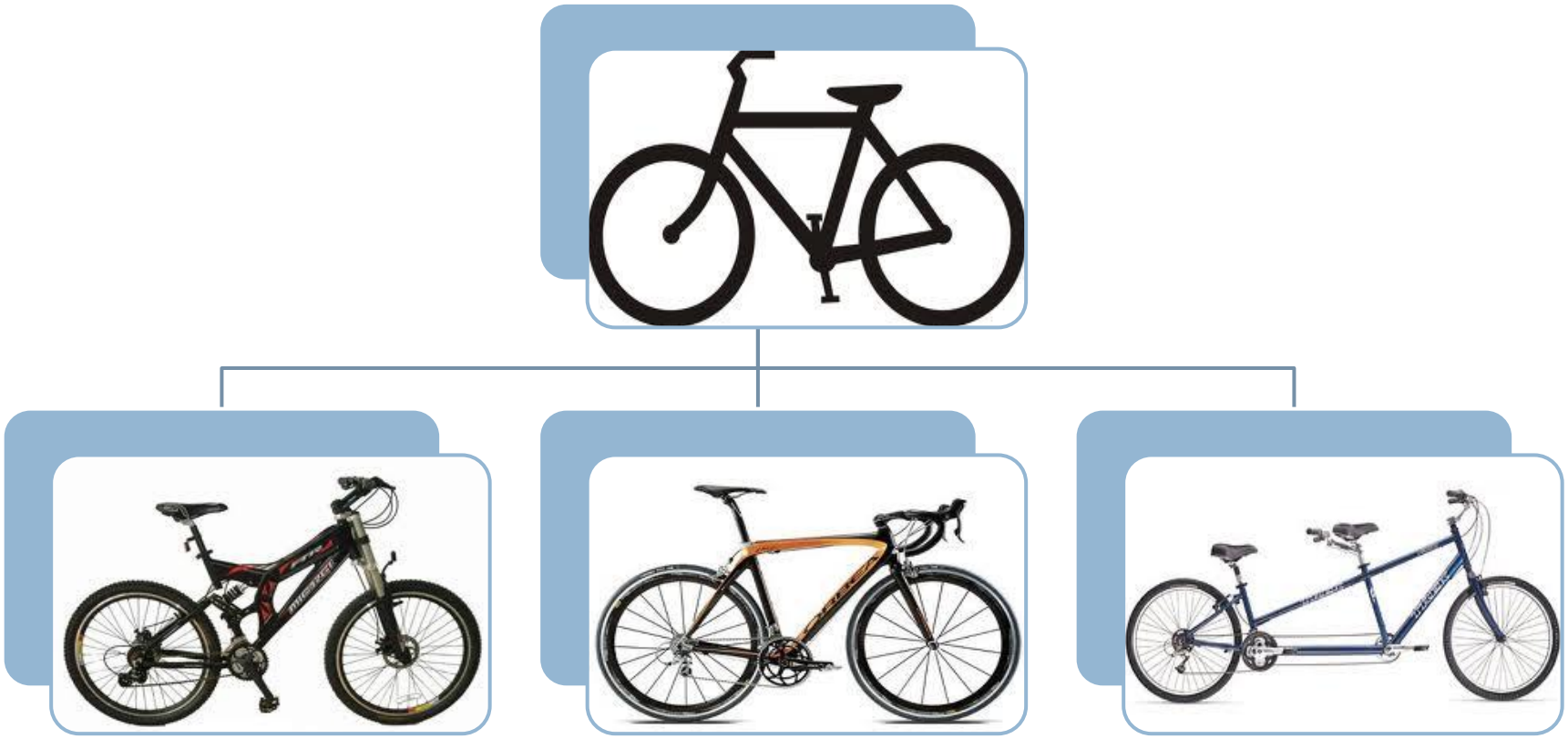
Relação de herança entre classes

- Usando o mecanismo de herança, o programador consegue criar, de forma expedita, 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

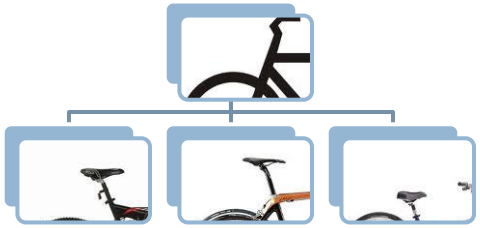
```
public interface I {                    // interface
    void p1();
}
public class A implements I {           // superclasse
    public void p1() { }
}
public class B extends A {              // subclasse
    public void p2() { } // B herda o método p1() de A
}
```



Declaração de classes e sub-classes: Bicicletas



Classe BicycleClass



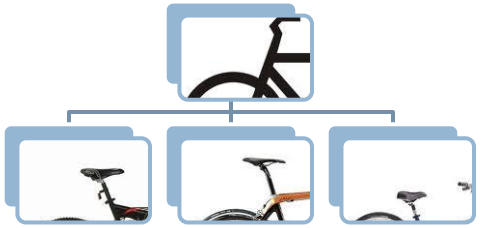

```
public class BicycleClass {
  private int speed;
  private int cadence;
  private int gear;

  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

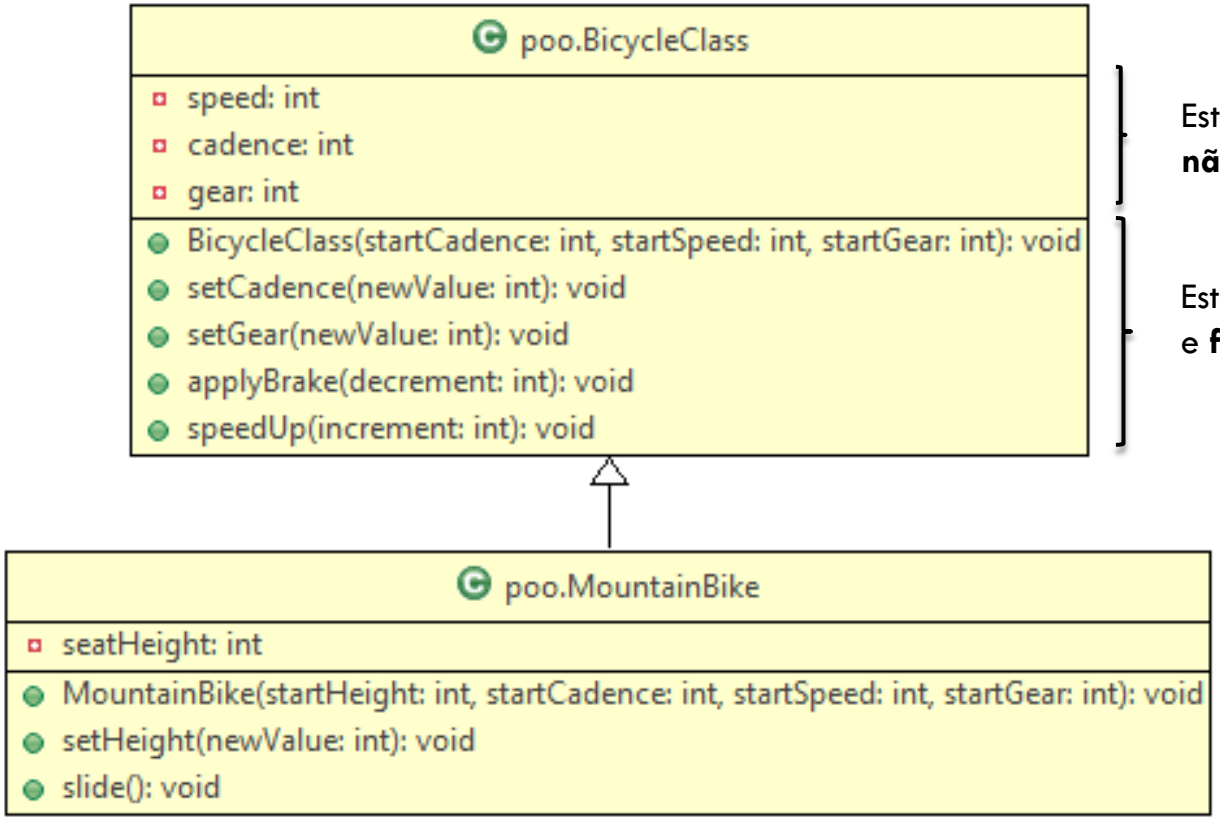
- Uma sub-classe declara a sua super-classe indicando no cabeçalho da sua definição que estende (**`extends`**) essa classe
  - Isto significa que herda, automaticamente, todos os métodos da super-classe, com os mesmos nomes e assinaturas
  - Além disso, a sub-classe pode fornecer **métodos adicionais**
- Uma sub-classe concreta **tem de implementar** todos os **métodos adicionais**, bem como os **métodos abstractos** da sua super-classe
  - Além disso, a sub-classe concreta pode **reimplementar**, ou **redefinir**, os métodos normais da super-classe
    - Estes métodos têm de ter assinaturas compatíveis com as definidas na super-classe
  - Herda da super-classe os métodos finais

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 (dados e operações) visíveis para as sub-classes?

- Declarar a visibilidade como **`protected`**
- Os membros protegidos são introduzidos para permitir implementações mais eficientes das classes
  - Podem 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`)

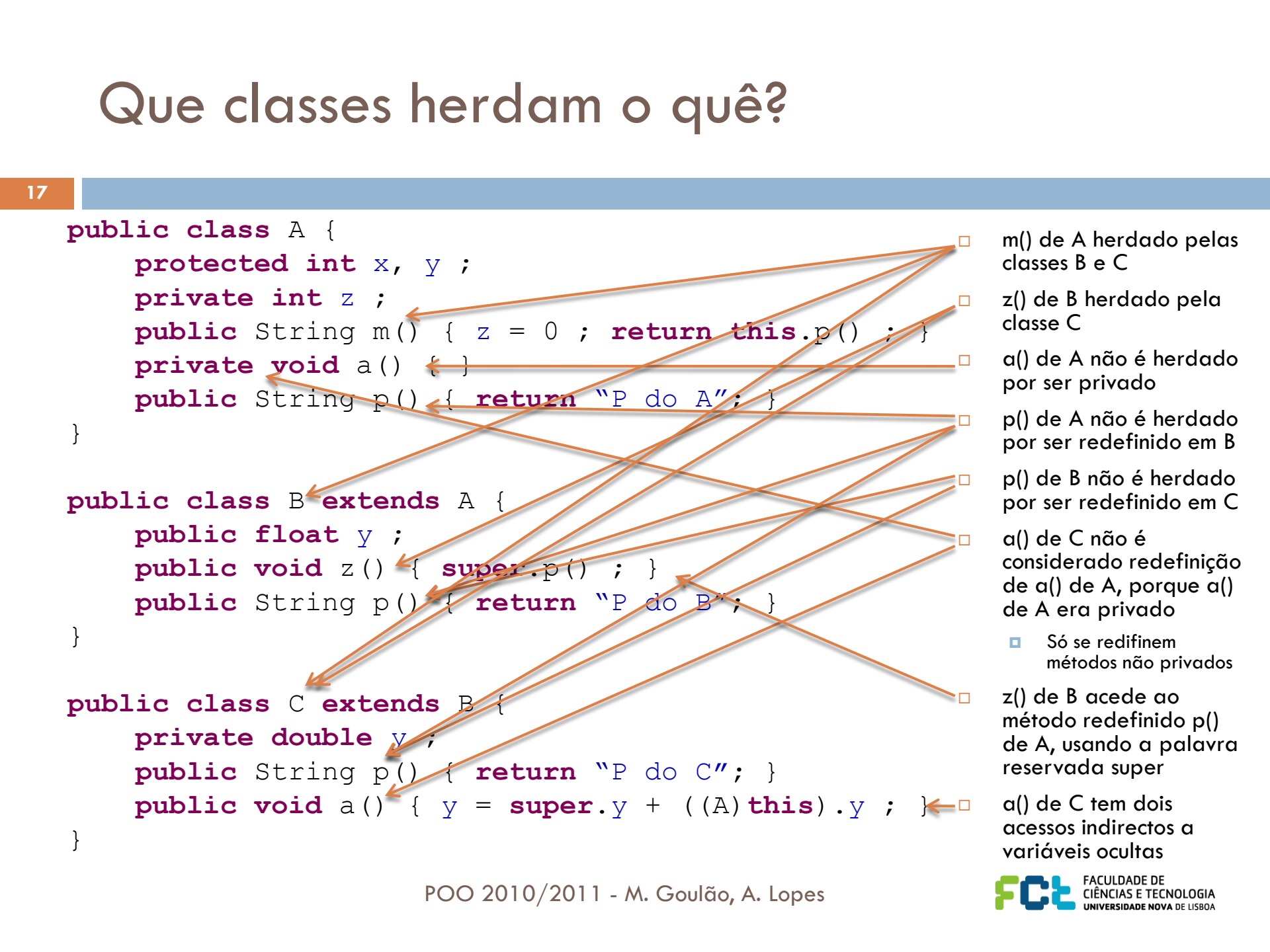
- O **acesso directo** a detalhes de representação da super-classe apenas é possível se a super-classe tornar partes da sua implementação acessíveis às sub-classes

Desvantagens dos membros protegidos

- Em geral, devemos **evitar** o uso de membros protegidos
  - Sem eles, a implementação da super-classe pode ser alterada sem afectar as sub-classes
  - Os membros protegidos são visíveis em todo o package, o que representa uma quebra no encapsulamento da classe
    - 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
    - Estas últimas dizem-se **ocultas** (do inglês **hidden**, ou **shadowed**)
      - 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 void 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 void m() { z = 0 ; 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 void m() { z = 0 ; this.p() ; } // Definido na
                                       // classe A!
```

- Quanto vale?
  - `new A().m()` → "P do A"
  - `new B().m()` → "P do B"
  - `new C().m()` → "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 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 e/ou 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, além de acrescentar métodos e atributos novos