


Laboratório de Engenharia de Software

INF1636 – PROGRAMAÇÃO ORIENTADA A OBJETOS

Departamento de Informática – PUC-Rio

Ivan Mathias Filho
ivan@inf.puc-rio.br




Laboratório de Engenharia de Software

Programa – Capítulo 5

- Conversão de Tipos
- Polimorfismo
- O Modificador `final`
- O Modificador `abstract`
- Exercícios – IR e Lista Encadeada

© LES/PUC-Rio 2

Programa – Capítulo 5




Laboratório de Engenharia de Software

- **Conversão de Tipos**
- Polimorfismo
- O Modificador `final`
- O Modificador `abstract`
- Exercícios – IR e Lista Encadeada

© LES/PUC-Rio 3

Conversão (1)



Laboratório de Engenharia de Software

- O trecho de código a seguir é uma boa oportunidade para se estudar a compatibilidade entre classes, subclasses e superclasses:


```
Public class Ex {  
  
    public static void main(String[] args) {  
        Poligono p1=new Quadrado();  
        Retangulo r1=new Quadrado();  
    }  
}
```

- Nesse exemplo pode-se notar que a variável `p1`, do tipo `Poligono`, irá referenciar um objeto do tipo `Quadrado` após a execução de `new Quadrado()`.

© LES/PUC-Rio 4

Laboratório de Engenharia de Software

Conversão (2)




- Isso pode ser feito porque todo quadrado é um retângulo, isto é, todas as propriedades de um retângulo são válidas para um quadrado;
- O mesmo se aplica aos polígonos, pois todas as suas propriedades são válidas para os quadrados;
- Logo, é possível referenciar um objeto de uma classe a partir de uma referência para um ancestral dessa classe;
- A atribuição no sentido inverso (**classe=ancestral**) também pode ser feita, mas em condições especiais.

© LES/PUC-Rio 5

Laboratório de Engenharia de Software

Conversão explícita




- Assim como se pode converter um `double` em um `int`, embora possa haver perda de informação, também se pode converter um objeto de uma classe em um objeto de uma subclasse;
- Entretanto, isso deve ser feito de maneira explícita;
- Para tal, deve-se pôr, imediatamente antes do objeto que será convertido, o nome da subclasse entre parênteses.

```
public static void main(String[] args) {  
    Poligono p1=new Quadrado();  
    Quadrado q1=(Quadrado)p1;  
}
```

© LES/PUC-Rio 6

Laboratório de Engenharia de Software

Regras gerais de conversão



- É sempre possível atribuir **ancestral=descendente** sem uma conversão explícita, pois uma instância de um descendente é uma instância de um ancestral;
- A atribuição **descendente=(descendente)ancestral** também pode ser feita;
- Neste caso, entretanto, a compatibilidade será verificada em tempo de execução, e uma exceção poderá ser levantada.


```
Poligono p1=new Quadrado();
Retangulo q1=(Quadrado)p1; //OK p1 é um retângulo
```

```
Poligono p2=new Triangulo();
Quadrado q2=(Quadrado)p2; //erro de tempo de execução
                        //um triângulo não é um quadrado
```

© LES/PUC-Rio
7

Laboratório de Engenharia de Software

Conversão na passagem de parâmetros



- A conversão de tipos também se aplica à passagem de parâmetros. O exemplo a seguir ilustra essa situação:

```
public class UmaClasse {
    Object p;
    public UmaClasse(Object x) {
        p=x;
    }
}
```

```
public static void main(String[] args) {
    Quadrado p1=new Quadrado();

    UmaClasse u=new UmaClasse(p1); //Object é ancestral de todas
                                   //as classes
}
```

© LES/PUC-Rio
8

O operador instanceof (1)



- O trecho de código abaixo irá gerar uma exceção quando for feita uma tentativa de conversão do terceiro elemento do array (**Triangulo**) para o tipo **Retangulo**:

```
public class Main {
    public static void main(String[] args) {
        Poligono []lst=new Poligono[3];
        Retangulo x;
        lst[0]=new Retangulo();
        lst[1]=new Quadrado();
        lst[2]=new Triangulo();
        for(Poligono p:lst)
            x=(Retangulo)p;
    }
}
```

```
Exception in thread "main" java.lang.ClassCastException: Triangulo
cannot be cast to Retangulo at Main.main(Main.java:19)
```

© LES/PUC-Rio

9

O operador instanceof (2)



- Para evitar a ocorrência de tais exceções, a linguagem Java fornece um operador que permite testar o tipo de um objeto em tempo de execução;
- A sintaxe desse operador é a seguinte:


```
<varObj> instanceof <tipoObj>
```

- O operador **instanceof** retorna true se o objeto <varObj> for uma instância de <tipoObj> ou de um descendente do mesmo;
- Caso contrário, o operador retorna false.

© LES/PUC-Rio

10

Exemplo



Laboratório de Engenharia de Software

```
public class Main {  
    public static void main(String[] args) {  
        Poligono []lst=new Poligono[3];  
        Retangulo x;  
        lst[0]=new Retangulo();  
        lst[1]=new Quadrado();  
        lst[2]=new Triangulo();  
        for(Poligono p:lst)  
            if(p instanceof Retangulo)  
                x=(Retangulo)p;  
    }  
}
```

© LES/PUC-Rio

11

Programa – Capítulo 5



Laboratório de Engenharia de Software


- Conversão de Tipos
- **Polimorfismo**
- O Modificador final
- O Modificador abstract
- Exercícios – IR e Lista Encadeada

© LES/PUC-Rio

12

Laboratório de Engenharia de Software

Polimorfismo (1)




- Polimorfismo (**poli-** + **-morfismo**) é a capacidade de assumir várias formas;
- Em termos práticos, pode-se dizer que é um meio de usar um mesmo nome para se referir a vários métodos distintos;
- Em Java existem dois tipos de polimorfismo:
 - O primeiro, já visto, é chamado de **overloading** (sobrecarga);
 - Ele se resume a uma classe possuir vários métodos com o mesmo nome, mas com assinaturas distintas;

© LES/PUC-Rio

13

Laboratório de Engenharia de Software

Polimorfismo (2)




- (cont)
 - O segundo, chamado **overriding** (sobrescrita), ocorre quando uma classe possui um método com a mesma assinatura (nome, tipo e ordem dos parâmetros) que um método de sua superclasse;
 - Quando isso acontece, dizemos que o método da classe derivada (subclasse) **sobrescreve** o método da classe da qual ele foi herdado.

© LES/PUC-Rio

14

Sobrescrita




- Na sobrecarga, a amarração entre a chamada do método e o método em si é resolvida em tempo de compilação (*early binding* ou amarração estática);
- Na sobrescrita, a amarração entre a chamada do método e o método em si é resolvida em tempo de execução (*late binding* ou amarração dinâmica);
- Isso ocorre porque, como foi visto anteriormente, uma referência para uma classe pode, em tempo de execução, referenciar objetos da própria classe ou objetos das classes derivadas.

Laboratório de Engenharia de Software

© LES/PUC-Rio 15

Sobrescrita – Exemplo



- Sejam as classes C1 e C2 abaixo:

```
public class C1 {  
    public void m1() {  
        System.out.println("PUC-Rio");  
    }  
}
```


```
public class C2 extends C1 {  
    public void m1() {  
        System.out.println("Departamento de Informática");  
    }  
}
```

Laboratório de Engenharia de Software

© LES/PUC-Rio 16

Laboratório de Engenharia de Software

Sobrescrita – Exemplo (1)




- No exemplo a seguir a chamada `p.m1()` irá executar o método `m1()` definido na classe `C1`, pois `p` referencia um objeto de `C1` em tempo de execução;
- Neste exemplo, a mensagem **PUC-Rio** será exibida no console.

```
public class Ex09 {  
    public static void main(String[] args) {  
        C1 p=new C1();  
  
        p.m1();  
    }  
}
```

© LES/PUC-Rio 17

Laboratório de Engenharia de Software

Sobrescrita – Exemplo (2)



- No exemplo a seguir a chamada `p.m1()` irá executar o método `m1()` definido na classe `C2`, pois `p` referencia um objeto de `C2` em tempo de execução;
- Nesse exemplo, a mensagem **Departamento de Informática** será exibida no console.

```
public class Ex09 {  
    public static void main(String[] args) {  
        C1 p=new C2();  
  
        p.m1();  
    }  
}
```

© LES/PUC-Rio 18

Sobrescrita – Visibilidade



- Embora uma variável de uma certa classe possa referenciar objetos de classes derivadas em tempo de execução, somente as propriedades definidas na superclasse são visíveis para essa variável;
- Isso quer dizer que mesmo que se acrescente um método `m2()` na classe `C2`, o comando a seguir irá causar um erro de compilação.

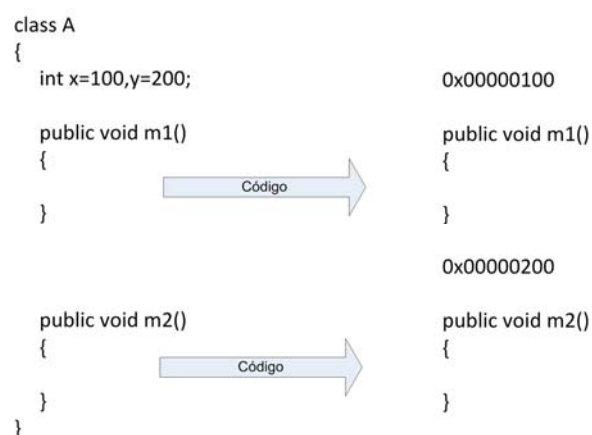
```
public class Ex09 {
    public static void main(String[] args) {
        C1 p=new C2();

        p.m2(); //erro: o método m2()
               //não está definido para o tipo C1
    }
}
```

© LES/PUC-Rio

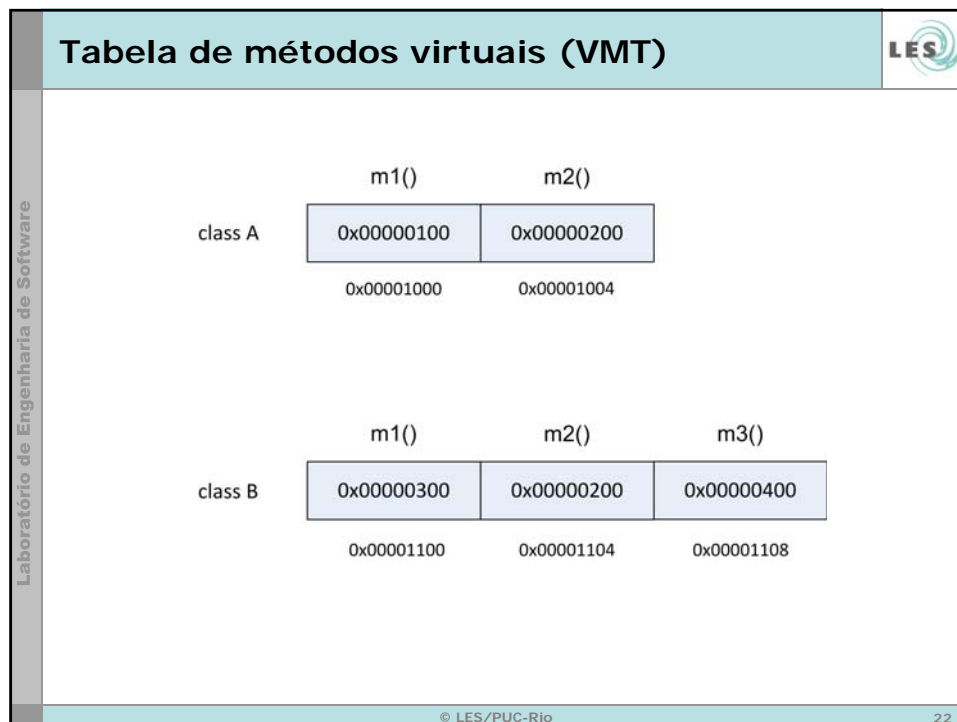
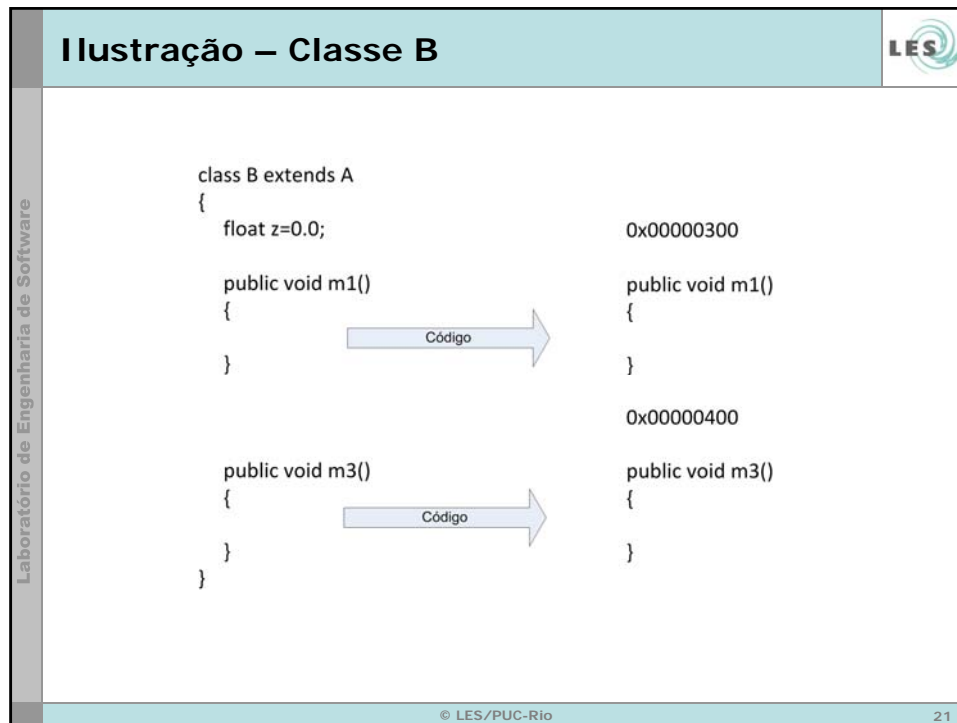
19

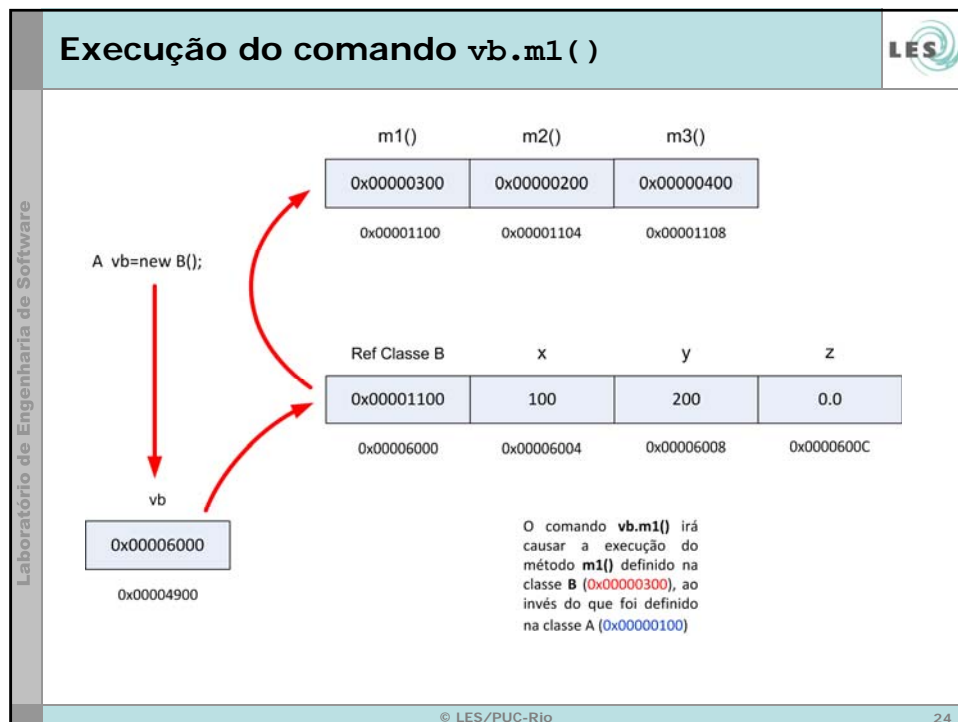
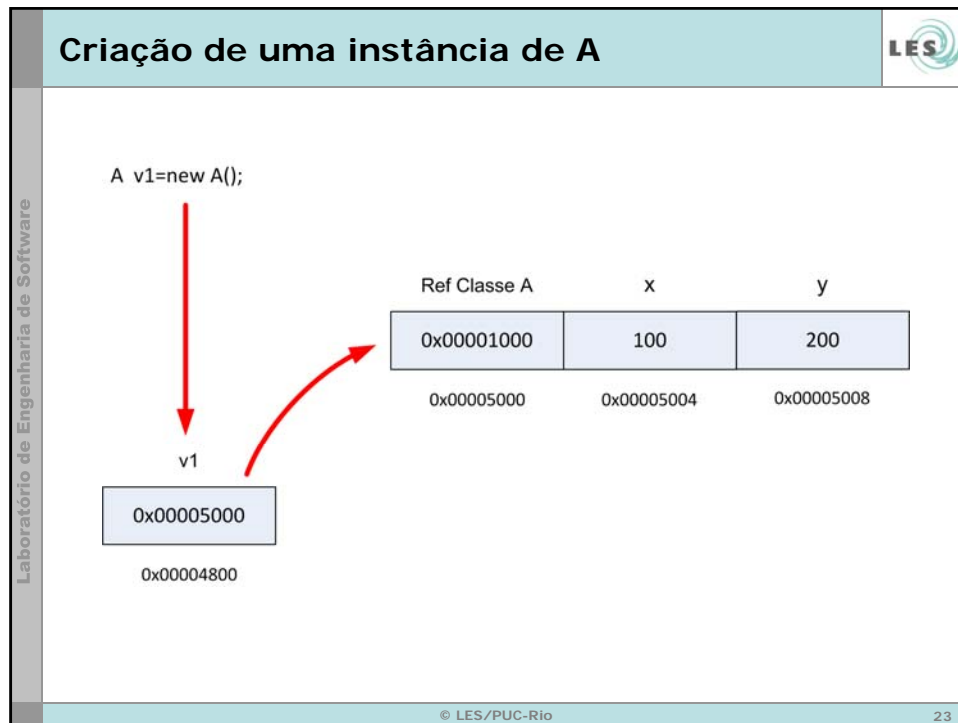
Ilustração – Classe A

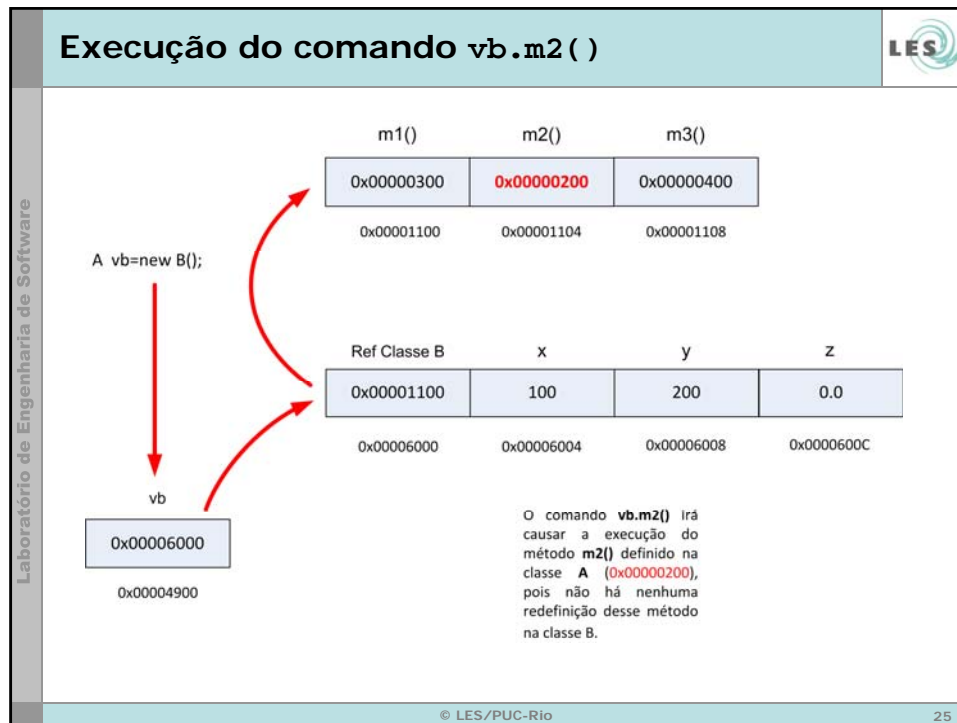


© LES/PUC-Rio

20







Programa – Capítulo 5

vb

0x00006000
0x00004900

m1()	m2()	m3()
0x00000300	0x00000200	0x00000400
0x00001100	0x00001104	0x00001108

Ref Classe B	x	y	z
0x00001100	100	200	0.0
0x00006000	0x00006004	0x00006008	0x0000600C

O comando `vb.m2()` irá causar a execução do método `m2()` definido na classe **A** (0x00000200), pois não há nenhuma redefinição desse método na classe B.

© LES/PUC-Rio
26

- Conversão de Tipos
- Polimorfismo
- **O Modificador final**
- O Modificador abstract
- Exercícios – IR e Lista Encadeada

O modificador final



- O modificador `final` já foi visto anteriormente. Naquela ocasião ele foi usado para definir constantes:

```
private final int x=2; //define uma constante inteira
                    //cujo valor é 2
```

- Quando o modificador `final` é usado em uma classe fica-se impedido de criar subclasses da mesma.

```
public final class C3 {}

public class C4 extends C3 {} //erro: a classe final C3
                             //não pode se estendida
```

© LES/PUC-Rio

27

O modificador final - Uso em métodos




- Quando o modificador `final` é usado em um método fica-se impedido de sobrescrevê-lo.


```
public class C3 {
    public final void m3()
    { }
}

public class C4 extends C3 {
    public void m3() //erro: o método final m3()
                   //não pode se sobrescrito
    { }
}
```

© LES/PUC-Rio


28

Laboratório de Engenharia de Software	Programa – Capítulo 5	
	<ul style="list-style-type: none">• Conversão de Tipos• Polimorfismo• O Modificador <code>final</code>• O Modificador <code>abstract</code>• Exercícios – IR e Lista Encadeada	
© LES/PUC-Rio		29

Laboratório de Engenharia de Software	O modificador <code>abstract</code> (1)	
	<ul style="list-style-type: none">• Quando o modificador <code>abstract</code> é usado na declaração de uma classe ele define que essa classe é abstrata;• Uma classe abstrata normalmente possui um ou mais métodos abstratos;• Um método abstrato não possui implementação, seu propósito é obrigar que as classes descendentes forneçam implementações para esse método;• Rotular um método com abstrato obriga que a classe na qual ele se encontra também seja declarada abstrata.	
© LES/PUC-Rio		30

Laboratório de Engenharia de Software

O modificador abstract (2)




- Deve-se declarar uma classe abstrata quando as três condições a seguir forem verdadeiras:
 - Várias subclasses de uma certa classe (raiz da hierarquia) serão definidas;
 - Todos os objetos serão tratados como instâncias da classe raiz;
 - A classe raiz, por si só, não deve ser instanciada, pois ela não contém todas as informações necessárias para tal.

© LES/PUC-Rio

31

Laboratório de Engenharia de Software

O modificador abstract – Exemplo (1)



- A classe `java.awt.Component` pode ser tomada como exemplo para as três condições anteriores;
- Ela é superclasse de muitas classes Java usadas para a construção de interfaces gráficas (botões, labels, menus, campos de texto e etc.);
- Dessa forma, ela atende à primeira condição apresentada anteriormente, pois ela possui várias subclasses.

© LES/PUC-Rio

32

O modificador abstract – Exemplo (2)



- Um **container** é um componente GUI cujo propósito é agrupar e exibir uma coleção de objetos gráficos;
- Devido à grande diversidade de componentes visuais, não seria muito prático fornecer um método diferente para registrar cada tipo componente junto a um **container**;
- Logo, seria conveniente que se pudesse escrever o seguinte trecho de código Java:
`myContainer.add(Component c);`
- Neste ponto passa-se a atender à segunda condição, pois, por conveniência, deve-se tratar todos os componentes como instâncias da superclasse `Component`.

O modificador abstract – Exemplo (3)



- Os componentes gráficos são objetos concretos, logo, eles deverão ser **renderizados** na tela do usuário;
- Uma instância de `Component`, entretanto, não contém as informações sobre a forma e o comportamento dos componentes gráficos de modo a se poder exibi-los;
- Apenas as subclasses de `Component` poderão fazer a **renderização** dos componentes, sobrescrevendo, para tal, um método abstrato que foi declarado na classe `Component`;
- Com isso atende-se ao terceiro e último requisito, pois a classe `Component` não deve ser instanciada, uma vez que não contém todas as informações necessárias para tal.

O modificador abstract – Exemplo (4)



- Abaixo é apresentada a forma geral da classe Component. Ela uma simplificação da verdadeira classe `java.awt.Component`;
- O método `draw()` também não existe como tal. Ele foi usado no lugar do método `paintComponent()` com objetivos didáticos.

```
public abstract class Component {
    public void setBackground(java.awt.Color c) { /* código */}
    public void setVisible(boolean r) { /* código */}
    public void setLocation(int x,int y) { /* código */}
    public abstract void draw();
}
```

© LES/PUC-Rio

35

O modificador abstract – Exemplo (5)



- Alguns métodos, como `setBackground()` e `setVisible()`, são aplicáveis a todos os componentes, logo, podem ser implementados na própria classe Component;
- Outros, como o método `draw()`, devem ser implementados pelas próprias subclasses, por isso devem se declarados abstratos e devem ser sobrescritos nas subclasses.

```
public class Button extends Component {
    public void draw() { /* código */}
}

public class Scrollbar extends Component {
    public void draw() { /* código */}
}
```

© LES/PUC-Rio

36

O modificador abstract – Exemplo (6)



- O último exemplo de código se refere a uma classe que precisa manipular componentes gráficos;
- Ela irá tratar todos eles como instâncias de `Component`;
- Quando houver necessidade de desenhar um componente qualquer ela enviará a mensagem `draw()` para ele, que **polimorficamente** irá executar a implementação de `draw()` definida na classe do objeto referenciado.

```
public class Container {
    public void remove(Component comp) { /* código */}
    public void add(Component comp) {
        comp.draw();
    }
}
```

© LES/PUC-Rio

37

Sobrescrita de método estático (1)




- Métodos estáticos não podem ser sobrescritos, embora o compilador Java não aponte erro quando existem dois métodos estáticos, em classes distintas, com a mesma assinatura;
- Isso parece contraditório, mas ficará claro após a explicação a seguir:
 - Sejam duas classes A e B, em que B é subclasse de A.
 - A existência de dois métodos estáticos com a mesma assinatura, um em cada classe, não produz erro de compilação.

© LES/PUC-Rio

38

Laboratório de Engenharia de Software

Sobrescrita de método estático (2)



```

public class A {
    static int x=5;

    public static int m1() {
        return x+1;
    }
}

public class B extends A {
    public static int m1() {
        return x*3;
    }
}


```

© LES/PUC-Rio

39

Laboratório de Engenharia de Software

Sobrescrita de método estático (3)



- A inexistência de erro de compilação no exemplo anterior não significa que **B.m1()** seja uma sobrescrita de **A.m1()**, quando se pensa em polimorfismo;
- A razão é muito simples: em uma chamada polimórfica, a amarração é feita em tempo de execução (amarração dinâmica);
- Entretanto, quando um método estático é chamado a amarração é feita em tempo de compilação (amarração estática);
- Dessa forma, não há polimorfismo, e, por conseguinte, não se pode caracterizar o exemplo anterior como sobrescrita.

© LES/PUC-Rio

40

Sobrescrita de método estático (4)



- Seja a classe Main a seguir:

```
public class Main {
    public static void main(String[] args) {
        A o1=new B();

        System.out.println(A.m1());
        System.out.println(B.m1());
        System.out.println(o1.m1());
    }
}
```

- O primeiro **println()** irá exibir no console o valor **6**, pois o método chamado foi **A.m1()**;
- O segundo **println()** irá exibir o valor **15**, pois o método chamado foi **B.m1()**;
- E o terceiro **println()**? O que ele irá exibir?

© LES/PUC-Rio

41


Sobrescrita de método estático (5)



- Embora métodos estáticos devam ser chamados, preferencialmente, por meio de nomes de classes, não há impedimento de chamá-los por meio de nomes de variáveis de objetos (apenas um **warning** será exibido);
- Uma vez que a amarração é estática, o compilador não leva em consideração (nem teria como) o fato de **o1** referenciar um objeto da classe **B**;
- Como **o1** foi declarado como objeto da classe **A**, o compilador faz uma amarração estática e chama o método **A.m1()** no terceiro **println()**, o que resulta na exibição do valor **6**;
- Logo, no exemplo apresentado não há sobrescrita, pois não há polimorfismo, mas, sim, amarração estática.

© LES/PUC-Rio

42

Laboratório de Engenharia de Software	Programa – Capítulo 5	
	<ul style="list-style-type: none">• Conversão de Tipos• Polimorfismo• O Modificador <code>final</code>• O Modificador <code>abstract</code>• Exercícios – IR e Lista Encadeada	
© LES/PUC-Rio		43