

TI-220 Java Orientado a Objetos

ANTONIO CARVALHO - TREINAMENTOS

A solid blue horizontal bar spanning the width of the slide, located at the bottom.

Especificadores de Acesso

Modificadores de Acesso

Existem 4 modificadores de acesso

- *default* (package)
- public
- private
- protected

Modificador de Acesso (default)

- **default (package)**, sinaliza a classe ou o membro que o acesso só pode ser feito por classes que pertençam ao mesmo package.
- Pode ser usado em **classes, métodos e variáveis**

Modificador de Acesso (default)

- Acesso padrão (package)
 - Não possui *keyword* para identificar este modificador, quando quiser usar ***package*** não coloque nada
 - Se uma classe A estiver declarada como package e se uma classe B que estiver em outro pacote tentar acessá-la, ocorrerá erro porque a classe B não está no mesmo pacote da classe A
 - Para que o acesso ocorra a classe B precisa ser criado no mesmo package que a classe A

Modificador de Acesso (default)



```
package ocjp.java.certification;  
class Animal {  
    public void locomover() {  
        System.out.println("locomovendo");  
    }  
}
```

Classe **Gato**
Pertence a outro
Package



```
package ocjp.java.certification;  
public class Cachorro extends Animal {  
    @Override  
    public void locomover() {  
        System.out.println("Andando");  
    }  
}
```

Compile success

```
package ocjp.java.certification.outropkg;  
import ocjp.java.certification.Animal;  
public class Gato extends Animal {  
    @Override  
    public void locomover() {  
        System.out.println("Andando");  
    }  
}
```

ocjp.java.certification.Animal não é público no ocjp.java.certification;
não pode ser acessado por fora do package

Modificador de Acesso (default)

Para corrigir o problema no exemplo anterior

- É preciso colocar ambas as classes **Animal** e **Gato** no mesmo ***package***
- **Ou** trocar o modificador da classe **Animal** para **public**

Modificador de Acesso (default)

Por que usamos o package ?

- O uso do package assegura que a classe não pode ser manipulada por outras classes fora do pacote.
- Como exemplo uma classe chamada Criptografia que contém métodos e variáveis necessários a um sistema de segurança.
 - Neste caso o desenvolvedor do sistema não deseja que sua classe Criptografia seja acessada por sistemas externos, portanto ele declara-a como package, dessa forma esta classe so é visível as classes do sistema de segurança

Modificador de Acesso (public)

public, indica que todas as outras classes independente do pacote em que estejam, podem acessar esta classe ou este membro, assumindo é claro que a classe onde o membro se encontra esteja visível para a classe que vai acessá-lo.

- Pode ser usado em **classes, métodos e variáveis**

Modificador de Acesso (public)

O modificador ***public*** torna a classe, o método ou a variável visível para todas as outras classes.

Por exemplo corrija o problema anterior modificando a classe **Animal** para ***public***

Dessa forma a classe **Animal** será visível para todas as outras classes

A solid blue horizontal bar spanning the width of the slide at the bottom.

Modificador de Acesso (private)

private, demarca o elemento como privado, ou seja apenas os métodos internos da classe podem acessar os elementos marcados como private.

- Somente pode ser usado em **métodos** e **variáveis**

Modificador de Acesso (private)


Um membro private não é transmitido através da herança.

No caso abaixo é possível compreender melhor como se classifica esta situação

- Este caso não é uma sobrescrita porque a classe **Gerente** não recebeu o método **trabalhar** por herança da classe **Funcionario**
- Portanto o que está ocorrendo é apenas a declaração de um novo método chamado **trabalhar** na classe **Gerente**

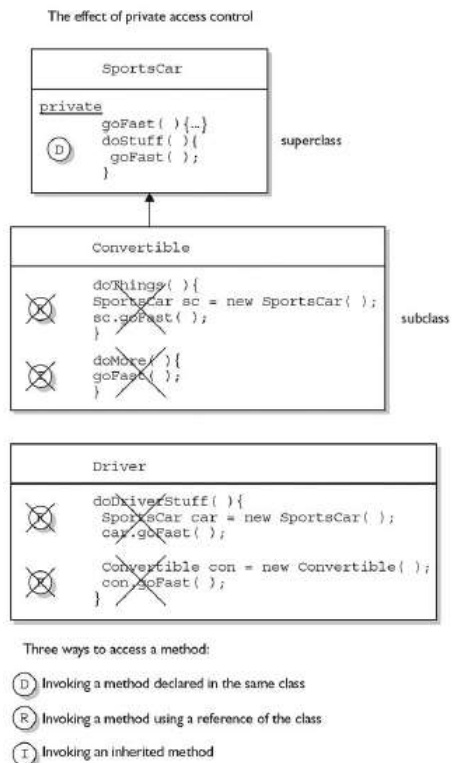
```
package ocjp.java.certification;  
public class Funcionario {  
    private void trabalhar() {  
        System.out.println("trabalhando");  
    }  
}
```

```
package ocjp.java.certification;  
public class Gerente extends Funcionario {  
    private void trabalhar() {  
        System.out.println("trabalhando");  
    }  
}
```

A blue arrow points from the Gerente class box to the Funcionario class box, indicating that Gerente inherits from Funcionario.

Observe a figura 1-3 no livro da Katy Sierra

Modificador de Acesso (private)



Fonte : SCJP Sun Certified Programmer
for Java 6 Study Guide (Exam 310065)


Modificador de Acesso (private)

Ao tentar usar um método ou variável que esta na superclasse ocorrerá erro de compilação.

- A classe **Gerente** não recebeu o método trabalhar por herança da classe **Funcionario**
- Portanto este método não pode ser chamado da classe **Gerente**

```
package ocjp.java.certification;  
public class Funcionario {  
    private void trabalhar() {  
        System.out.println("trabalhando");  
    }  
}
```

```
package ocjp.java.certification;  
public class Gerente extends Funcionario {  
    private void liderar() {  
        trabalhar();  
        System.out.println("liderando");  
    }  
}
```



O método trabalhar() originário do tipo Funcionario não é visível.

Modificador de Acesso (protected)

protected, sinaliza a que o membro somente pode ser acessado por classes que pertençam ao mesmo package ou através da herança mesmo que a classe pertença a outro package


- Pode ser usado em **métodos** e **variáveis**

Modificador de Acesso (protected)

Veja o exemplo ao lado e responda as questões

- Por que o método **trabalhar()** pode ser acessado diretamente ?
- Por que este mesmo método não pode ser acessado via objeto (**f**) do tipo **Funcionario** ?
- O que pode ser feito para que o método **trabalhar** seja executado das duas formas ? (não vale torná-lo público)

```
package ocjp.java.certification;  
public class Funcionario {  
    protected void trabalhar() {  
        System.out.println("trabalhando");  
    }  
}
```



```
package ocjp.java.certification.outro;  
public class Outro extends Funcionario {  
    private void liderar() {  
        Funcionario f = new Funcionario();  
        trabalhar();  
        f.trabalhar();  
        System.out.println("liderando");  
    }  
}
```

The method `trabalhar()` from the type `Funcionario` is not visible

Modificador de Acesso (protected)

Por que o método **trabalhar()** pode ser acessado diretamente ?

- **R:** Porque ele é um método **protected** e está sendo acessado através de herança da classe **Funcionario**

Por que este mesmo método não pode ser acessado via objeto (**f**) do tipo **Funcionario** ?

- **R:** Pelo fato do método ser **protected** ele somente pode ser acessado por classes do mesmo pacote ou através da herança. O objeto (**f**) do tipo **Funcionario** está tentando acessar o método **trabalhar()** diretamente, mas está sendo invocado por uma classe que não pertence ao **package**.

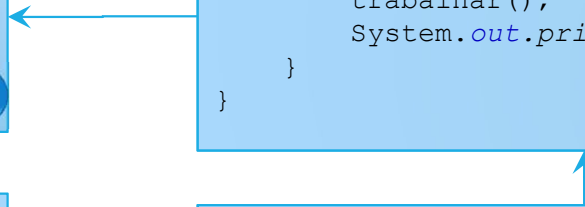
O que pode ser feito para que o método **trabalhar** seja executado das duas formas ? (não vale torná-lo público)

- **R:** Para que o problema seja resolvido basta colocar ambas as classes no mesmo **package**

Modificador de Acesso (protected)

Veja o exemplo abaixo e responda as questões

```
package ocjp.java.certification;  
public class Funcionario {  
    protected void trabalhar() {  
        System.out.println("trabalhando");  
    }  
}
```



1

```
package ocjp.java.certification.outro;  
public class Outro extends Funcionario {  
    private void liderar() {  
        trabalhar();  
        System.out.println("liderando");  
    }  
}
```

2

```
package ocjp.java.certification.outro;  
public class Estagiario {  
    public void acompanharTrabalho() {  
        Outro o = new Outro();  
        o.trabalhar();  
    }  
}
```

3

```
package ocjp.java.certification.outro;  
public class NovoOutro extends Outro {  
    public void acompanharTrabalho() {  
        trabalhar();  
    }  
}
```

4

- Será possível acessar ao método trabalhar da classe outro através de uma relação que não seja a herança ?
- Ocorrerá algum erro em algum dos quadros ?

Modificador de Acesso (protected)

Será possível acessar o método trabalhar da classe **Outro** através de uma relação que não seja a herança ?

- **R:** Não como o método é **protected** a única forma de acesso a ele é estando dentro do mesmo **package**, ou através da **herança**

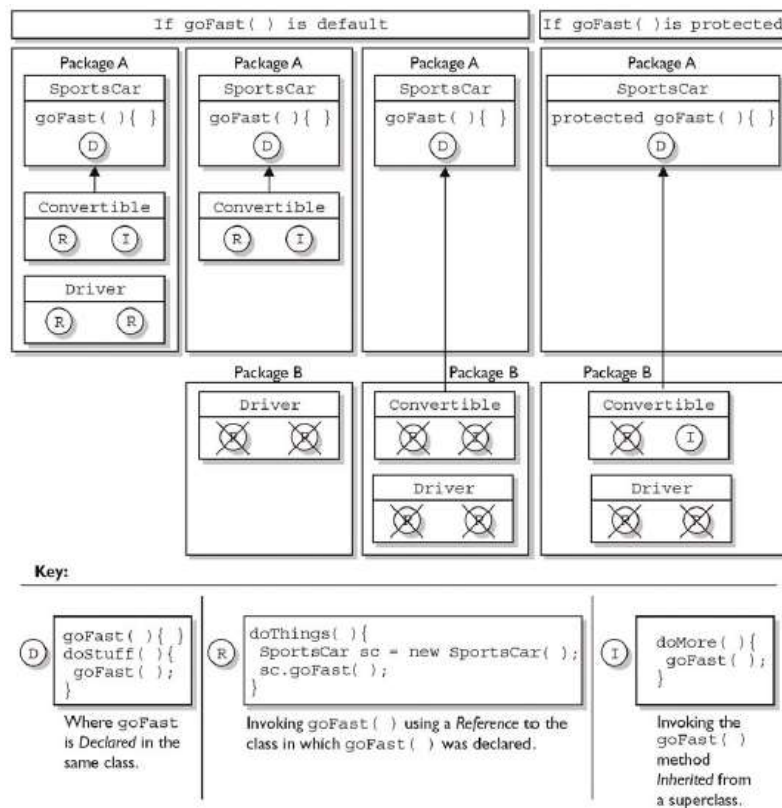
Ocorrerá algum erro em algum dos quadros ?

- **R:** Ocorrerá um erro no quadro 3, pois está tentando fazer um acesso através da **referência** e não através de **herança**

Observe a figura 1-4 no livro da Katy Sierra

Nota : Quando ocorre herança os membros são transmitidos com o mesmo modificador de acesso que foram criados.

Modificador de Acesso (protected)



Fonte : SCJP Sun Certified Programmer for Java 6 Study Guide (Exam 310065)

Modificador de Acesso

Visibilidade	Public	Protected	<i>Default</i>	Private
Da mesma classe	Sim	Sim	Sim	Sim
De qualquer classe do mesmo package	Sim	Sim	Sim	Não
De uma subclasse do mesmo package	Sim	Sim	Sim	Não
De uma subclasse de outro package	Sim	Sim	Não	Não
De qualquer classe de outro package (sem herança)	Sim	Não	Não	Não

Fonte : (SIERRA, 2008) – Adaptado pelo autor

Nota : Variáveis locais não podem receber modificadores de acesso

Outros Modificadores

Existem outros modificadores que não são de acesso:

- **abstract**
- **final**
- **strictfp**

Eles podem ser combinados com os modificadores de acesso

E podem ser combinados entre eles evitando apenas a combinação ***final*** e ***abstract***, pois um elemento não pode ser incompleto e completo ao mesmo tempo.

Outros Modificadores (abstract)

O modificador **abstract** identifica o elemento como sendo incompleto

Uma classe abstract não pode ser instanciada

Ela pode ser herdada por alguma subclasse a qual pode implementar os métodos que faltam, deixando de ser abstrata e permitindo a instanciação de objetos do seu tipo

- Pode ser usado apenas em **classes** e **métodos**

Nota : Os métodos marcados com o modificador **abstract** possuem apenas a assinatura e terminam com (;) no final da assinatura, não há o par de chaves { } que delimita o bloco de código .

Outros Modificadores (abstract)

Uma classe abstrata pode conter métodos **não abstratos**

A classe abstrata **pode** ou **não** conter métodos **abstratos**

Se houve um método for declarado como abstrato, então a classe inteira deve ser declarada como abstrata.

Se uma classe herda de uma classe abstrata é preciso que ela implemente todos os métodos abstratos para que se torne uma classe completa, porém é possível não implementar todos os métodos, tornando a subclasse abstrata também.

Nota : Não é possível criar objetos de uma classe abstrata. Para possibilitar instanciação de objetos é primeiro necessário estender a classe abstrata implementando os métodos abstratos, nesse momento haverá uma classe concreta que pode ser instanciada.

Outros Modificadores (abstract)

Classe abstrata

```
package ocjp.java.certification.abstrato;  
public abstract class Reptil {  
    public abstract void comer();  
    public void andar() {  
        System.out.println("Andando");  
    }  
}
```

```
package ocjp.java.certification.abstrato;  
public class Jacare extends Reptil {  
    @Override  
    public void comer() {  
        System.out.println("Como carne");  
    }  
}
```

Classe concreta

```
package ocjp.java.certification.abstrato;  
public class Lagarto extends Reptil {  
    @Override  
    public void comer() {  
        System.out.println("Como ovos");  
    }  
}
```

Classe concreta

Outros Modificadores (final)

Modificador ***final*** identifica que a classe não pode ser estendida (herdada)

Quando aplicada a um **método**, indica que este não pode ser **sobrescrito**, e quando aplicada em uma **variável**, indica que o seu valor **não pode ser alterado**.

Tornar uma classe final assegura que ninguém pode criar outra classe a partir dela, evitando o uso de uma subclasse modificada.

- Pode ser usado em **classes, métodos e variáveis**

Nota : O uso do modificador **final** deve ser planejado cuidadosamente, pois os métodos marcados com **final** não podem ser reescritos perdendo esta vantagem da Orientação a Objetos, assim como as classes marcadas com **final** não podem ser estendidas.

Outros Modificadores (final)

```
package ocjp.java.certification;
public class Funcionario {
    final public float pisoSalarial = 600.0f;
    protected void trabalhar() {
        System.out.println("trabalhando");
    }
}
```

```
package ocjp.java.certification;
public class Gerente extends Funcionario {
    private void liderar() {
        pisoSalarial = 750.0f;
        trabalhar();
        System.out.println("liderando");
    }
}
```

The final field Funcionario.pisoSalarial cannot be assigned

A variável pisoSalarial é **final** e não pode ter seu conteúdo **modificado**



Outros Modificadores (final)

O modificador **final** pode ser utilizado também em variáveis que são recebidas como parâmetros pelos métodos.

```
package ocjp.java.certification;
public class Funcionario {
    final public float pisoSalarial = 600.0f;
    protected void trabalhar() {
        System.out.println("trabalhando");
    }

    public void calculaSalario( final float valorHora ) {
    }
}
```

- Isto assegura que mesmo que a função seja **sobrescrita** em uma **herança**, o valor da variável não poderá ser modificado ao longo do código.

Outros Modificadores (strictfp)

strictfp, força o método a utilizar números flutuantes e operações com números flutuantes aderentes ao padrão IEEE 754.

- Algumas plataformas tem vantagens em precisão do que outras, ao usar strictfp o método não usará estas vantagens em precisão.

Sintaxe :

[modificador de acesso] strictfp <tipo> <nome> ([parametros])

Exemplo:

```
public strictfp float calculaPagamento() { }
```

- Pode ser usado em **métodos** ou em **classes** indicando que todos os métodos são **strictfp**

Outros Modificadores (synchronized)

synchronized, é aplicável a métodos e indica que o método somente pode ser acessado por uma *Thread* por vez

Sintaxe :

[modificador de acesso] **synchronized** <tipo> <nome> ([parametros])

Exemplo:

```
public synchronized boolean recebimento() { }
```

- Pode ser usado apenas em **métodos**

Outros Modificadores (native)

native, é aplicável a métodos e indica que o método é dependente da plataforma.

- O corpo do método não deve ser implementado, portanto a assinatura deve terminar com (;)

Sintaxe :

[modificador de acesso] native <tipo> <nome> ([parametros]);

Exemplo:

public native boolean recebimento();

- Pode ser usado apenas em **métodos**

Uso dos modificadores

Modificador	Classes	Métodos	Variáveis (não-locais)	Variáveis (locais)
final	Sim	Sim	Sim	Sim
<i>default</i> (package)	Sim	Sim	Sim	Não
public	Sim	Sim	Sim	Não
abstract	Sim	Sim	Não	Não
strictfp	Sim	Sim	Não	Não
private	Não	Sim	Sim	Não
protected	Não	Sim	Sim	Não
static	Não	Sim	Sim	Não
native	Não	Sim	Não	Não
synchronized	Não	Sim	Não	Não
transient	Não	Não	Sim	Não
volatile	Não	Não	Sim	Não

Dúvidas



Modificadores - Exercícios

(Fonte : Sierra, adaptador Autor)

Dados dois arquivos:

```
1. package pkgA;
2. public class Foo {
3.     int a = 5;
4.     protected int b = 6;
5.     public int c = 7;
6. }
3. package pkgB;
4. import pkgA.*;
5. public class Baz {
6.     public static void main(String[] args) {
7.         Foo f = new Foo();
8.         System.out.print(" " + f.a);
9.         System.out.print(" " + f.b);
10.        System.out.print(" " + f.c);
11.    }
12. }
```

Qual será o resultado ? (Escolha todos que se aplicam)

- A) 5 6 7
- B) 5 seguido por uma exceção
- C) Falha de compilação com erro na linha 7
- D) Falha de compilação com erro na linha 8
- E) Falha de compilação com erro na linha 9
- F) Falha de compilação com erro na linha 10

Modificadores - Exercícios

(Fonte : Sierra, adaptador Autor)

Dado o seguinte código:

```
4. public class Frodo extends Hobbit {  
5.     public static void main(String[] args) {  
6.         Short myGold = 7;  
7.         System.out.println(countGold(myGold, 6));  
8.     }  
9. }  
10. class Hobbit {  
11.     int countGold(int x, int y) { return x + y; }  
12. }
```

Qual é o resultado ?

A. 13

B. Falha de compilação devido a múltiplos erros

C. Falha de compilação devido a um erro na linha 6

D. Falha de compilação devido a um erro na linha 7

E. Falha de compilação devido a um erro na linha 1

Bibliografia

BARNES, DAVID J. Programação Orientada a Objetos com Java

DEITEL, Java - Como Programar - 6ª Edição, Pearson Education

BEZERRA, EDUARDO Princípio de Análise e Projeto de Sistemas com UML, Campus

SIERRA, KATHY e BATES BERT, Use a Cabeça Java, Alta Books

SIERRA, KATHY e BATES BERT, OCA/OCP Java SE 7 Programmer I & II Study Guide

