Linguagem de Programação I

Prof. Edson Kaneshima

Herança

Aula 04

Herança

- A idéia por trás da herança é que podemos **criar novas classes com base em classes já existentes**.

- Quando se herda de uma classe existente, reutilizamos (ou herdamos) seus métodos e atributos, além de adicionarmos novos métodos e atributos para adaptar a nova classe a novas situações.

Classe, Superclasse e Subclasse

- Considere a classe Empregado que discutimos nas aulas anteriores.

- Suponha que uma empresa trata os gerentes de forma diferente que os outros empregados. Os gerentes certamente são iguais a quaisquer outros empregados em vários aspectos.

- Tanto os empregados como os gerentes recebem um salário. Entretanto, enquanto os empregados devem finalizar as tarefas atribuídas a eles como retorno por receberem um salário, os gerentes ganham um bônus se realizarem o trabalho a eles atribuído.

Classe, Superclasse e Subclasse

- Esse é o tipo de situação que pede o uso da herança.

- Precisa-se definir uma nova classe Gerente e adicionar funcionalidades a ela. Porém, pode-se reter um pouco do que já foi programado na classe Empregado, e todos os atributos da classe original podem ser preservados.

- De forma mais abstrata, há um relacionamento “é-um” entre Gerente e Empregado. Todo gerente **é um** empregado: esse relacionamento “é-um” é a marca da herança.

Classe, Superclasse e Subclasse

- Definindo uma classe Gerente que herda da classe Empregado.

- Usa-se a palavra extends para indicar a herança.

```
public class Gerente extends Empregado {
        atributos e métodos adicionados...
}
```

- A palavra-chave extends indica a criação de uma nova classe que deriva de uma classe existente.

Classe, Superclasse e Subclasse

- A classe existente é chamada de superclasse ou classe base. A nova classe é chamada de subclasse ou classe derivada.

- Os termos superclasse e subclasse são os mais freqüentemente usados pelos programadores, embora alguns prefiram a analogia pai/filho, que se adapta bem ao tema “herança”.

- A classe Empregado é uma superclasse, mas não por ser superior à sua subclasse ou conter mais funcionalidades. Na realidade, acontece o contrário: as subclasses têm mais funcionalidades do que suas superclasses.

Classe, Superclasse e Subclasse

- A nova classe Gerente possui um novo atributo para armazenar o bônus, e um novo método para defini-lo:

```
public class Gerente extends Empregado {
        private double bonus;
        public void setBonus (double bonus) {
                this.bonus = bonus;
        }
}
```

Classe, Superclasse e Subclasse

- Não há nada especial sobre esse método e atributo. Desde que se tenha um objeto Gerente, pode-se simplesmente aplicar o método setBonus().

```
Gerente chefe = new Gerente();

chefe.setBonus(5000);
```

- É claro que se tiver um objeto Empregado não poderá aplicar o método setBonus() – ele não está entre os métodos definidos na classe Empregado.
- Entretanto, pode-se usar métodos como getNome() com objetos Gerente, pois eles são herdados automaticamente da superclasse Empregado.

Classe, Superclasse e Subclasse

- Do mesmo modo, os atributos nome, salario são herdados da superclasse. Assim, todo objeto Gerente possui os atributos: nome, salario e bonus.

- Ao definir uma subclasse estendendo sua superclasse, é necessário apenas indicar as diferenças entre a subclasse e a superclasse.

- Ao elaborar classes, coloca-se os métodos mais gerais dentro da superclasse e os mais especializados dentro da subclasse.

Classe, Superclasse e Subclasse

- No entanto, alguns dos métodos da superclasse não são apropriados para a subclasse Gerente. Em particular, o método getSalario deve retornar a soma do salário base com o bônus. É necessário fornecer um novo método para sobrepor o método da superclasse:

```
public class Gerente extends Empregado  {

        ...

        public double getSalario( ) {

                ...

        }

        ...

}
```

Classe, Superclasse e Subclasse

- Como implementar esse método? À primeira vista, parece ser simples: basta retornar a soma dos atributos salario e bonus.

```
public double getSalario( ) {
        return salario + bônus;   //não funcionará
}
```

- O método getSalario( ) da classe Gerente não possui acesso direto aos atributos privados da superclasse. Isso significa que o método getSalario( ) da classe Gerente não pode acessar diretamente o atributo salario, mesmo que todo objeto Gerente possua um atributo salario.

Classe, Superclasse e Subclasse

- Somente os métodos da classe Empregado têm acesso aos atributos privados. Se os métodos Gerente quiserem acessar esses atributos privados, eles precisam fazer o que todos os outros métodos fazem – usar a interface pública, neste caso o método público getSalario da classe Empregado.
- Tentando novamente...

```
public double getSalario( ) {
        return getSalario( ) + bônus; //ainda não funcionará
}
```

Classe, Superclasse e Subclasse

- Problema: a chamada de getSalario( ) simplesmente chama a si mesma, porque a classe Gerente possui um método getSalario( ) (que é o método que estamos tentando implementar).

- A conseqüência é um conjunto infinito de chamadas ao mesmo método, levando ao travamento do programa.

- É necessário indicar que se quer invocar o método getSalario( ) da superclasse Empregado, e não a classe corrente. Use a palavra-chave especial **super** para esse propósito.

Classe, Superclasse e Subclasse

- Chama o método getSalario( ) da classe Empregado. Segue a versão correta do método getSalario( ) para a classe Gerente:

```
public double getSalario( ) {
        return super.getSalario( ) + bonus;
}
```

Classe, Superclasse e Subclasse

- Finalmente, vamos definir um construtor:

```
public Gerente (String nome, double salario, double bonus) {
        super (nome, salario);
        this.bonus = bonus;
}
```

- Agora a palavra-chave super tem um significado diferente. A instrução

```
super (nome, salario);
```

- é uma abreviação para “chame o construtor da superclasse Empregado com nome e salario como parâmetros”.

Classe, Superclasse e Subclasse

- Pelo fato do construtor de Gerente não poder acessar os atributos privados da classe Empregado, ele precisa inicializá-los através de um construtor.
- O construtor é invocado com a sintaxe especial super. A chamada usando super deve ser o primeiro comando do construtor da subclasse.
- Se o construtor da subclasse não chamar um construtor da superclasse explicitamente, então o construtor padrão (sem parâmetros) da superclasse é invocado. Se a superclasse não possuir nenhum construtor padrão e o construtor da subclasse não chamar explicitamente outro construtor para a superclasse, então o compilador Java apresenta um erro.

Classe, Superclasse e Subclasse

- Redefinido o método getSalario( ) para objetos Gerente, os gerentes automaticamente terão os bônus adicionados a seus salários.

- Exemplo: criando um novo gerente:

```
Gerente chefe = new Gerente (“Pedro”, 8000, 3000);

System.out.println (chefe.getSalario( ) );
```

Hierarquias de Herança

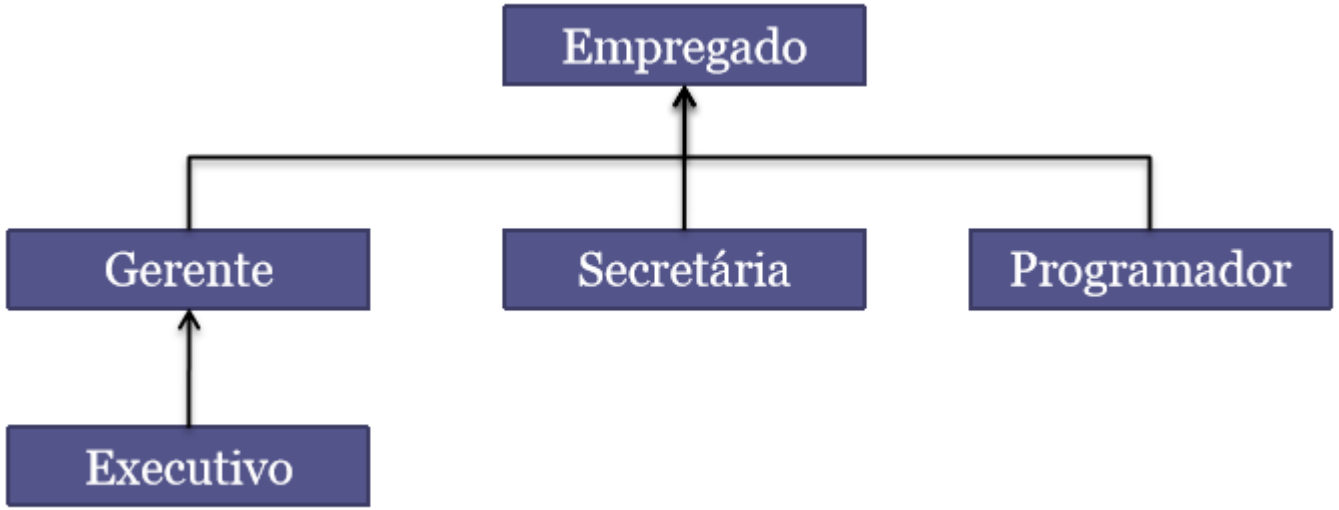
- A herança não precisa se limitar a derivar uma camada de classes. Poderíamos ter uma classe Executivo que estendesse Gerente, por exemplo.

- A coleção de todas as classes que se estendem a partir de uma superclasse em comum é chamada de hierarquia de herança.

- O caminho de uma determinada classe até seus ancestrais dentro da hierarquia de herança é sua cadeia de herança.

Hierarquias de Herança

- Normalmente, há mais de uma cadeia descendendo de uma classe ancestral distante. Poderíamos formar uma subclasse Programador ou Secretária que estendesse Empregado, e elas não teriam nada a ver com a classe Gerente(ou uma com a outra). Esse processo pode continuar por tanto tempo quanto necessário.



Evitando a Herança: Classes e Métodos Finais

- Ocasionalmente, pode-se impedir que alguém forme uma subclasse a partir de uma de suas classes. As classes que não podem ser estendidas são chamadas de classes finais, e usa-se o modificador final na definição da classe para indicar essa situação.
- Ex.: Suponha que se deseje impedir que outras pessoas criem subclasses a partir de Executivo.

```
public final class Executivo extends Gerente {

        ...

}
```

Evitando a Herança: Classes e Métodos Finais

- Pode-se também tornar final um método específico de uma classe. Se o fizer, nenhuma subclasse poderá substituir esse método.

```
public class Empregado {

        ...

        public final String getNome( ) {

                return nome;

        }

        ...

}
```

- Todos os métodos de uma classe final são automaticamente final.

Evitando a Herança: Classes e Métodos Finais

- Só existe uma boa razão para tornar um método ou classe final: certificar-se de que a semântica não possa ser modificada em uma subclasse.

- Por exemplo, a classe String é uma classe final. Isso significa que não se pode definir uma subclasse de String. Em outras palavras, caso se tenha uma referência a String, então sabe-se que ela referese a uma String e nada mais.