Java - Herança

BCC 221 - Programação Orientada a Objectos(POO)

Guillermo Cámara-Chávez

Departamento de Computação - UFOP

Herança

- A herança é uma das características primárias da orientação a objetos
  - Uma forma de **reuso de software** pela qual uma classe nova é criada e absorve os membros de **classes já existentes, aprimorando-os**;
  - Diminui o tempo de implementação;
  - Aumenta a confiabilidade e qualidade do software

Herança (cont.)

- Uma classe já existente e que é herdada é chamada de **superclasse**
  - A **nova classe** que herdará os membros é chamada de **subclasse**.
- Uma subclasse é uma forma especializada da superclasse;

Herança (cont.)

- Uma subclasse também pode vir a ser uma superclasse.
- A superclasse direta é a superclasse da qual a subclasse herda explicitamente
  - As outras são consideradas superclasses indiretas.

Herança (cont.)

- Na herança única, uma subclasse herda somente de uma superclasse direta
- Java não permite a realização de herança múltipla, em que uma subclasse pode herdar de mais de uma superclasse direta;
- No entanto, é possível utilizar interfaces para desfrutar de alguns dos benefícios da herança múltipla

Herança (cont.)

- Um problema com a herança é que a subclasse pode herdar métodos que não precisa ou que não deveria ter
  - Ainda, o método pode ser necessário, mas inadequado;
  - A classe pode sobrescrever (*override*) um método herdado para adequá-lo

Especificadores de Acesso

- **public**:
  - Os membros *public* de uma classe são acessíveis em qualquer parte de um programa em que haja uma referência a um objeto da classe ou das subclasses.
- **private**:
  - Membros *private* são acessíveis apenas dentro da própria classe.
- **protected**:
  - Membros *protected* podem ser acessados por por membros da própria classe, de subclasses e de classes do mesmo pacote

Promoção de Argumentos

- Todos os membros *public* e *protected* de uma superclasse mantêm seus especificadores de acesso quando se tornam membros de uma subclasse
- Subclasses se referem a estes membros simplesmente pelo nome;
- Quando uma **subclasse sobrescreve um método da superclasse**, o **método original** da superclasse ainda **pode ser acessado** quando antecedido pela **palavra super** seguida de **.**

```
super.metodo();
```

Classe *Object*

- A hierarquia das classes em Java é iniciada pela classe *Object*
- Todas as outras classes herdam (ou estendem) direta ou indiretamente a partir dela
- Define um construtor e 11 métodos
  - Alguns devem ser sobrescritos pelas subclasses para melhor funcionamento.
- Não possui atributos.

Classe *Object* (cont.)

Métodos da classe Object	
clone()	*getclass()*
equals()	*hashCode()*
finalize()	*notify(), notifyAll()*
toString()	*wait()* 3 versões

Classe *Object* (cont.)

- **clone()**
  - Método protected;
  - Retorna uma referência para Object
    - Exige um *cast* para o objeto original.
  - Realiza a cópia do objeto a partir do qual foi invocado;

Classe *Object* (cont.)

- Classes devem sobrescrevê-lo como um método público
- Devem também implementar a interface *Cloneable*.
- A implementação padrão realiza uma **cópia rasa** (*shallow copy*)
- Uma implementação sobrescrita normalmente realiza uma **cópia profunda** (*deep copy*).

Classe *Object* (cont.)

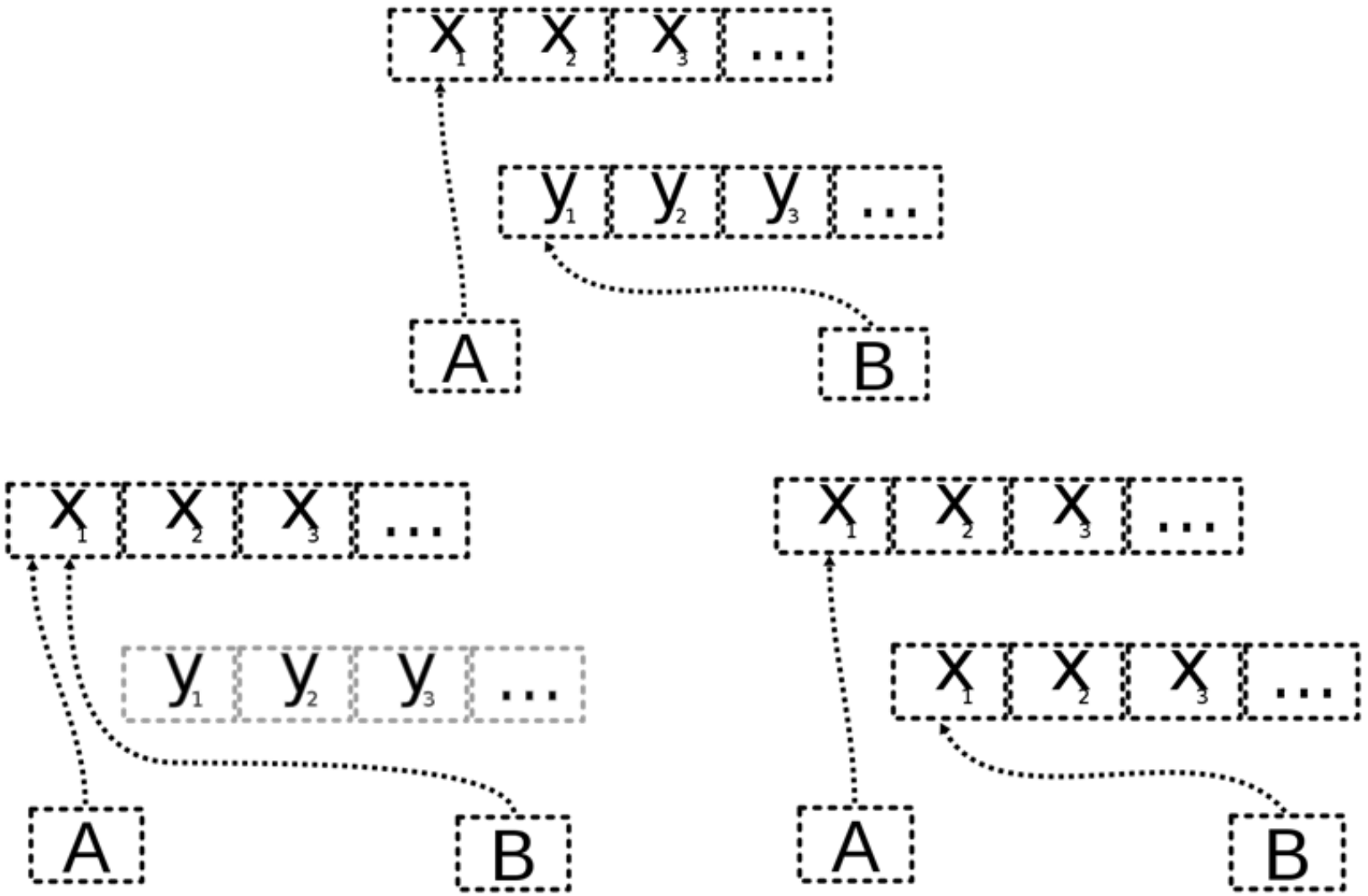
- A cópia rasa (*shallow copy*) realiza o mínimo de duplicação possível
  - É uma cópia de referência, não dos elementos
  - Não existe para classes que possuem apenas tipos primitivos.
  - Dois objetos compartilham os mesmos membros.

Classe *Object* (cont.)

- A cópia profunda (*deep copy*) realiza o máximo de duplicação possível
  - Cria um novo objeto completo e independente;
  - Mesmo conteúdo do objeto clonado, membro a membro.

`http://www.java2s.com/Code/Java/Class/ShallowCopyTest.htm`

Classe *Object* (cont.)



Classe *Object* (cont.)

- **equals()**
  - Compara dois objetos quanto a igualdade e retorna *true* caso sejam iguais, *false* caso contrário;
  - Quando dois objetos de uma classe em particular precisarem ser comparados, este método deve ser sobrescrito

`https://www.sitepoint.com/implement-javas-equals-method-correctly/`

Classe *Object* (cont.)

- **finalize()**
  - Invocado pelo coletor de lixo automático para realizar a terminação de um objeto prestes a ser coletado;
  - Não há garantia de que o objeto será coletado, portanto, não há garantia de que este método será executado;

Classe *Object* (cont.)

- **getClass()**
  - Todos objetos em Java conhecem o seu tipo em tempo de execução;
  - Este método retorna um objeto da classe *Class* (pacote *java.lang*) que contém informações sobre o objeto, como o nome da classe (obtido pelo método *getName()*, *getSimpleName*).

```
Person p = new Person();
System.out.printl("Classe: " +
    p.getClass().getSimpleName());
```

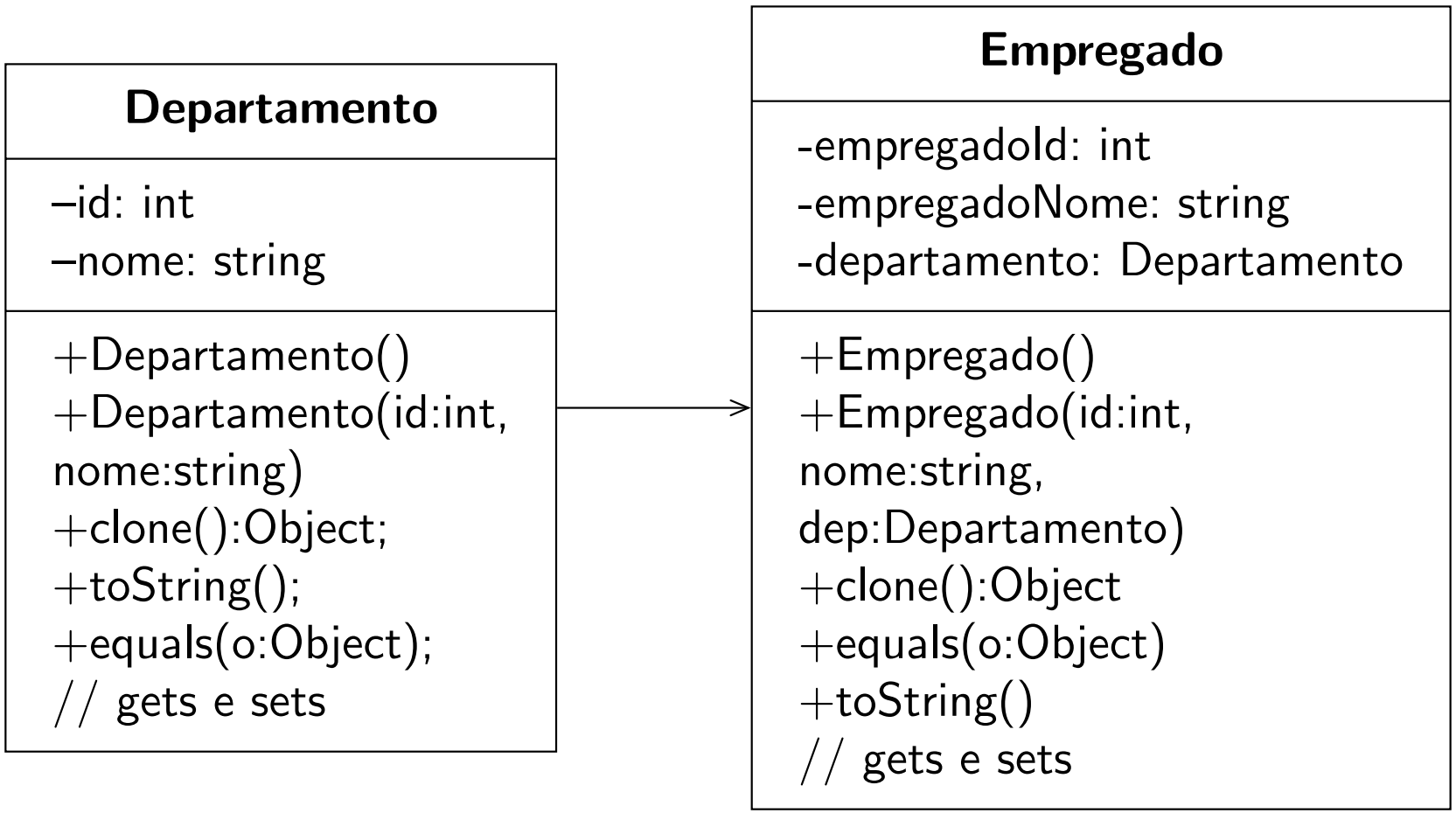
Classe *Object* (cont.)

- **toString()**
  - Retorna a representação do objeto que o invocou em formato de *string*;
  - A implementação padrão retorna os nomes do pacote e da classe, seguidos pela representação em hexadecimal do valor retornado pelo método *hashCode()*;
  - É recomendado que todas as subclasses sobrescrevam este método;
  - Pode ser utilizado em substituição de métodos *print()*

Exemplo

Sejam as classe **Departamento** e **Empregado**. Sobrescreva os métodos *clone()* e *equals()* da classe **Object**. Demonstre a diferença entre cópia rasa e cópia profunda de objetos desta classe através da utilização do métodos *clone()* e da comparação dos resultados através do método *equals()* e do operador ==.

Exemplo (cont.)



Departamento.java

```
public class Departamento implements Cloneable {
    private int id;
    private String nome;

    public Departamento(){}

    public Departamento(int id, String nome){
        this.id = id;
        this.nome= nome;
    }
    public void setNome(String nome){
        this.nome = nome;
    }

    public String getNome(){
        return nome;
    }
```

Departamento.java (cont.)

```
public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
}

public String toString(){
   return String.format("\n%s\n%s: %d\n%s: %s",
            "Dados Departamento",
            "Identificacao", getId(),
            "Nome:", getNome());
}
```

Departamento.java (cont.)

```
    public Object clone(){
        try{
            return super.clone();
        }
        catch(CloneNotSupportedException e){
            return null;
        }
    }

    public boolean equals(Object o){
        if (this == o) return true;
        if (o == null) return false;
        if (getClass() != o.getClass()) return false;
        final Departamento cp = (Departamento)o;
        return ( id == cp.getId() && nome.equals(cp.
            getNome()) );
    }
}
```

Empregado.java

```
public class Empregado implements Cloneable {
    private int empregadoId;
    private String empregadoNome;
    private Departamento departamento;

    public Empregado(){
        departamento = new Departamento();
    }

    public Empregado(int id, String nome, Departamento
        dep){
        setEmpregadoId(id);
        setEmpregadoNome(nome);
        setDepartamento(dep);
    }
```

Empregado.java (cont.)

```
public int getEmpregadoId() {
    return empregadoId;
}

public void setEmpregadoId(int empregadoId) {
    this.empregadoId = empregadoId;
}

public String getEmpregadoNome() {
    return empregadoNome;
}

public void setEmpregadoNome(String empregadoNome){
    this.empregadoNome = empregadoNome;
}

public Departamento getDepartamento() {
    return departamento;
}
```

Empregado.java (cont.)

```
public void setDepartamento(Departamento
    departamento) {
    this.departamento = departamento;
}

public void setDepartamentoNome(String nome){
    departamento.setNome(nome);
}

public String toString(){
    return String.format("\n%s\n%s: %d\n%s: %s%s",
            "Dados Empregado",
            "Identificacao", getEmpregadoId(),
            "Nome", getEmpregadoNome(),
            departamento );
}
```

Empregado.java (cont.)

```
public Object clone(){
    try{
        Empregado novo = (Empregado)super.clone();
        novo.setDepartamento((Departamento)novo.
            getDepartamento().clone());
        return novo;
    } catch( CloneNotSupportedException e){
        return null;
    }
}
```

Empregado.java (cont.)

```
    public boolean equals(Object o){
        if (this == o) return true;
        if (o == null) return false;
        if (getClass() != o.getClass()) return false;
        final Empregado cp = (Empregado)o;

        return (empregadoId == cp.getEmpregadoId() &&
           empregadoNome.equals(cp.getEmpregadoNome())
               &&
           departamento.equals(cp.getDepartamento()));
    }
}
```

DriverEmpregado.java

```
public class DriverEmpregado {

    public static void main(String[] args) {
        Departamento dep = new Departamento(1,
            "Computacao");
        Empregado emp = new Empregado(101,"Mario",dep);
        Empregado novo;
        novo = emp;
        if (novo == emp)
            System.out.println("Iguais");
        else
            System.out.println("Diferentes");

        novo.setEmpregadoNome("Gisele");

        System.out.println(novo);
        System.out.println(emp);
```

DriverEmpregado.java (cont.)

```
        Empregado novo2 = (Empregado)emp.clone();
        novo2.setDepartamentoNome("AEDS I");
        novo2.setEmpregadoNome("Mario");

        System.out.println(novo2);
        System.out.println(emp);

        Empregado novo3 = (Empregado)emp.clone();
        if (emp.equals(novo3))
            System.out.println("Conteudos Iguais");
        else
            System.out.println("Conteudos Diferentes");
    }
}
```

DriverEmpregado.java (cont.)

```
Iguais

Dados Empregado
Identificacao: 101
Nome: Gisele
Dados Departamento
Identificacao: 1
Nome:: Computacao

Dados Empregado
Identificacao: 101
Nome: Gisele
Dados Departamento
Identificacao: 1
Nome:: Computacao
```

DriverEmpregado.java (cont.)

```
Dados Empregado
Identificacao: 101
Nome: Mario
Dados Departamento
Identificacao: 1
Nome:: AEDS I

Dados Empregado
Identificacao: 101
Nome: Gisele
Dados Departamento
Identificacao: 1
Nome:: Computacao


Conteudos Iguais
```

Exemplo Herança

- Consideremos novamente o exemplo de uma empresa que possui dois tipos de empregados
  - Comissionados (superclasse)
    - Recebem uma comissão sobre vendas.
  - Assalariados Comissionados (subclasse)
    - Recebem salário fixo e comissão sobre vendas

Exemplo Herança (cont.)

```
public class ComissionEmployee {
    private String firstName;
    private String lastName;
    private String socialSecurityNumber;
    private double grossSales;
    private double comissionRate;
    public ComissionEmployee(String firstName,
        String lastName, String socialSecurityNumber,
        double grossSales, double comissionRate)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.socialSecurityNumber =
            socialSecurityNumber;
        setGrossSales(grossSales);
        setComissionRate(comissionRate);
    }
```

Exemplo Herança (cont.)

```
public String getFirstName() {
    return firstName;
}

public String getLastName() {
    return lastName;
}

public String getSocialSecurityNumber() {
    return socialSecurityNumber;
}

public double getGrossSales() {
    return grossSales;
}

public double getComissionRate() {
    return comissionRate;
}
```

Exemplo Herança (cont.)

```
public void setFirstName(String firstName) {
    this.firstName = firstName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}

public void setSocialSecurityNumber(String
    socialSecurityNumber) {
    this.socialSecurityNumber =
        socialSecurityNumber;
}

public void setGrossSales(double grossSales) {
    this.grossSales = grossSales < 0.0 ? 0.0 :
        grossSales;
}
```

Exemplo Herança (cont.)

```
public void setComissionRate(double comissionRate)
    {
     this.comissionRate = (comissionRate > 0.0 &&
         comissionRate < 1.0) ? comissionRate : 0.0;
}

public double earnings(){
    return getComissionRate() * getGrossSales();
}
```

Exemplo Herança (cont.)

```
    public String toString(){
        return String.format("%s: %s %s\n %s: %s\n %s:
            %.2f\n %s: %.2f",
            "comission employee", getFirstName(),
                getLastName(),
            "social security number ",
                getSocialSecurityNumber(),
            "gross sales", getGrossSales(),
            "comission rate", getComissionRate());
    }
}
```

Exemplo Herança (cont.)

```
public class ComissionEmployeeDriver {

    public static void main(String[] args) {
        ComissionEmployee employee = new
            ComissionEmployee(
            "Sue", "Jones", "222-22-2222",
            10000,0.06);
        System.out.println(employee);
    }
}
```

Exemplo Herança (cont.)

```
comission employee: Sue Jones
social security number : 222-22-2222
gross sales: 10000.00
comission rate: 0.06
```

Exemplo Herança (cont.)

- Construtores não são herdados
  - A primeira tarefa de qualquer construtor é invocar o construtor da superclasse direta
    - Implicita ou explicitamente.
  - Se não houver uma chamada explícita, o compilador invoca o construtor padrão
    - Sem argumentos;
    - Não efetua nenhuma operação.

Exemplo Herança (cont.)

- O método toString(), herdado da classe Object é sobrescrito na classe de exemplo
  - Retorna uma String que representa um objeto;
  - Este método é chamado implicitamente quando tentamos imprimir um objeto com *println()*
- O exemplo ainda utiliza o método *format* da classe *String*

BasePlusComissionEmployee.java

```
public class BasePlusComissionEmployee extends
    ComissionEmployee {
    private double baseSalary;

    public BasePlusComissionEmployee(double baseSalary,
        String firstName,
        String lastName,
        String socialSecurityNumber,
        double grossSales,
        double comissionRate) {
        super(firstName, lastName,socialSecurityNumber,
              grossSales, comissionRate);
        setBaseSalary(baseSalary);
    }
```

BasePlusComissionEmployee.java (cont.)

```
    public double getBaseSalary() {
        return baseSalary;
    }

    public void setBaseSalary(double baseSalary) {
        this.baseSalary = baseSalary < 0.0 ? 0.0 :
            baseSalary;
    }

    public double earnings(){
        return getBaseSalary() + super.earnings();
    }

    public String toString(){
        return String.format("%s \n%s\n %s: %.2f",
                "based-salaried", super.toString(),
                "base salary", getBaseSalary());
    }
}
```

Herança

- A herança é definida pela palavra reservada **extends**;
- A subclasse invoca o construtor da superclasse explicitamente através da instrução

```
super( first , last , ssn , sales , rate );
```

- Esta deve ser a primeira ação em um construtor.

Herança (cont.)

- Se um método realiza as operações necessárias em outro método, é preferível que ele seja chamado, ao invés de duplicarmos o código
  - Reduz a manutenção no código;
  - Boa prática de Engenharia de Software.
- No exemplo, invocamos o método *earnings()* da superclasse, já que ele é sobrescrito na subclasse

```
super.earnings();
```

BasePlusComissionEmployeeDriver.java

```
public class ComissionEmployeeDriver {
    public static void main(String[] args) {
        BasePlusComissionEmployee employee = new
            BasePlusComissionEmployee( 8000, "Sue",
                "Jones", "222-22-2222",10000,0.06);
        System.out.println(employee);
        System.out.println("\nTotal salary: " +
            employee.earnings());
    }
}
```

BasePlusComissionEmployeeDriver.java (cont.)

```
based-salaried
comission employee: Sue Jones
social security number : 222-22-2222
gross sales: 10000.00
comission rate: 0.06
base salary: 8000.00

Total salary: 8600.0
```

Redefinição de Métodos

- Subclasses podem redefinir métodos das superclasses
  - A assinatura pode até mudar, embora o nome do método permaneça;
  - A precedência é do método redefinido na classe derivada
    - Na verdade, este substitui o método da classe base na classe derivada.

Redefinição de Métodos (cont.)

- É comum que métodos redefinidos chamem o método original dentro de sua redefinição e acrescentem funcionalidades
  - Como no exemplo anterior, em que frases adicionais são impressas na redefinição do método *toString()*

Métodos e Classes *final*

- Uma variável ou atributo declarado com o modificador final é constante
  - Ou seja, depois de inicializada não pode ser modificada
- Um método declarado com o modificador *final* não pode ser **sobrescrito**
- Uma classe declarada com o modificador *final* não pode ser **estendida**
  - Embora possa ser utilizada em composições

Engenharia de *Software* com Herança

- Em uma hierarquia de herança, uma **subclasse não necessita** ter **acesso** ao **código fonte da superclasse**
  - Java exige apenas acesso ao arquivo **.class** da superclasse para que possamos compilar e executar uma subclasse

Engenharia de *Software* com Herança (cont.)

- Esta característica é útil para *software* proprietário
  - Basta distribuí-lo em formato *bytecode*, não é necessário fornecer o código fonte;
  - No entanto, deve haver **documentação precisa sobre o funcionamento da classe**, para que outros programadores a compreendam.

Exercícios

Crie uma classe **Equipamento** com o atributo ligado (tipo *boolean*) e com os métodos *liga* e *desliga*. O método *liga()* torna o atributo ligado *true* e o método *desliga()* torna o atributo ligado *false*.
Crie também uma classe **EquipamentoSonoro** que herda as características de Equipamento e que possui os atributos *volume* (tipo *short*) que varia de 0 a 10 e *stereo* (tipo *boolean*). A classe ainda deve possuir métodos *getters* e *setters*, além dos métodos *mono()* e *stereo()*. O método *mono()* torna o atributo *stereo* falso e o método *stereo()* torna o atributo *stereo* verdadeiro. Ao ligar o **EquipamentoSonoro** através do método liga, seu volume é automaticamente ajustado para 5.

FIM