

# 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** ou classe **Base**
  - ▶ A **nova classe** que herdará os membros é chamada de **subclasse** ou classe **Derivada**.
- ▶ 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 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 sobreescreve 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 <b>Object</b>	
<i>clone()</i>	<i>getClass()</i>
<i>equals()</i>	<i>hashCode()</i>
<i>finalize()</i>	<i>notify()</i> , <i>notifyAll()</i>
<i>toString()</i>	<i>wait()</i> 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
- ▶ Cada classe que implemente *clone()* deve chamar a *super.clone()*.
- ▶ Devem também implementar a interface *Cloneable*, incluir a chamada da exceção *CloneNotSupportedException*
- ▶ A implementação padrão realiza uma **cópia rasa** (*shallow copy*)
- ▶ Uma implementação sobrescrita normalmente realiza uma **cópia profunda** (*deep copy*).

## Classe *Object* (cont.)

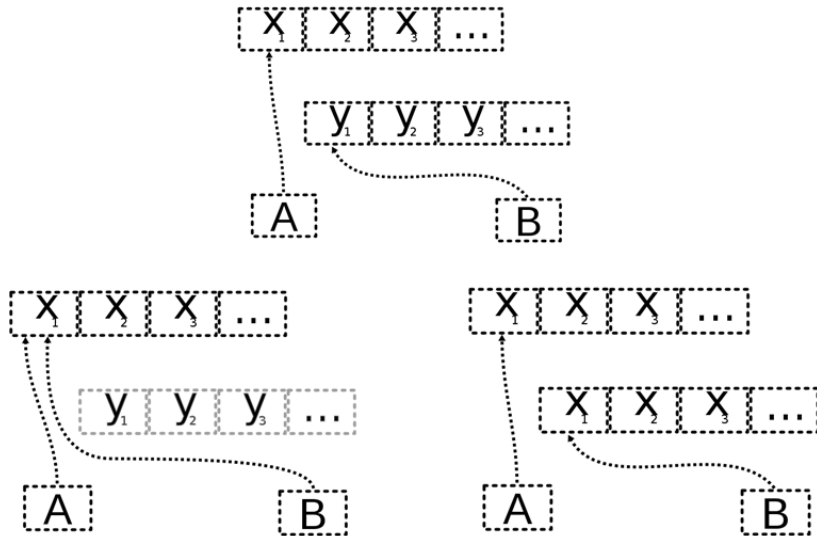
- ▶ A cópia rasa (*shallow copy*) realiza o mínimo de duplicação possível
  - ▶ É uma cópia de referência, não dos elementos
  - ▶ Não existe para classes que possuem apenas tipos primitivos.
  - ▶ Dois objetos compartilham os mesmos membros.

## 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.println("Classe: " + p.getClass().getSimpleName());
```

Classe Person

## 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()` (quando não é sobrecarregado o método: `className@hexadecimal hash`)

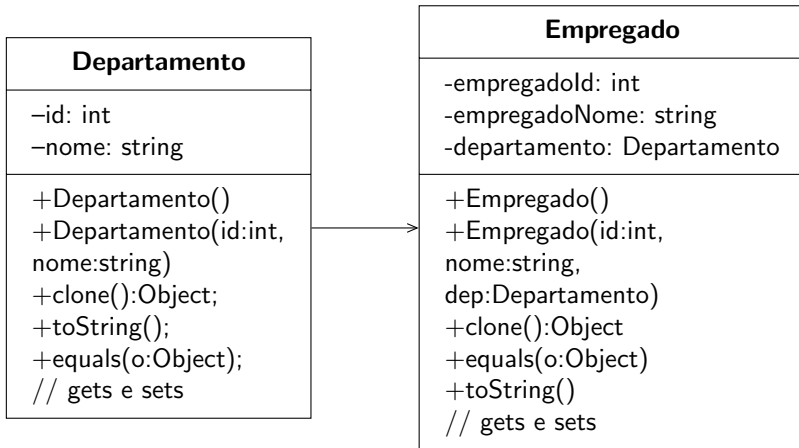
```
public String toString() {  
    return getClass().getName()+"@"+Integer.toHexString(  
        hashCode());  
}
```

- 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

```
import java.util.Objects;
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 int getId() {
        return id;
    }

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

## Departamento.java (cont.)

```
}

public String getNome() {
    return nome;
}

public void setNome(String nome) {
    this.nome = nome;
}

@Override
public String toString() {
    return "Departamento{" + "id=" + id + ", nome=" + nome + '}';
}
```

## Departamento.java (cont.)

@Override

```
public int hashCode() {  
    int hash = 7;  
    hash = 61 * hash + this.id;  
    hash = 61 * hash + Objects.hashCode(this.nome);  
    return hash;  
}
```

@Override

```
public boolean equals(Object obj) {  
    if (this == obj) return true;  
    if (obj == null) return false;  
    if (getClass() != obj.getClass()) return false;  
    final Departamento other = (Departamento) obj;  
    if (this.id != other.id) return false;  
    return Objects.equals(this.nome, other.nome);  
    //return ( id == cp.getId() nome.equals(cp.getNome()) );  
}
```



## Departamento.java (cont.)

```
@Override  
public Object clone() throws CloneNotSupportedException {  
    return super.clone();  
}  
}
```

# Empregado.java

```
import java.util.Objects;
public class Empregado implements Cloneable{
    private int id;
    private String nome;
    private Departamento departamento;

    public Empregado() {
    }

    public Empregado(int id, String nome, Departamento departamento) {
        this.id = id;
        this.nome = nome;
        this.departamento = departamento;
    }

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

## Empregado.java (cont.)

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

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

```
public void setNome(String nome) {  
    this.nome = nome;  
}
```

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

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

## Empregado.java (cont.)

```
@Override
public String toString() {
    return "Empregado{" + "id=" + id + ", nome=" + nome + ",
        departamento=" + departamento + '}';
}
```

```
@Override
public int hashCode() {
    int hash = 7;
    hash = 83 * hash + this.id;
    hash = 83 * hash + Objects.hashCode(this.nome);
    hash = 83 * hash + Objects.hashCode(this.departamento);
    return hash;
}
```

## Empregado.java (cont.)

```
@Override
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null) return false;
    if (getClass() != obj.getClass()) return false;
    final Empregado other = (Empregado) obj;
    if (this.id != other.id) return false;
    if (!Objects.equals(this.nome, other.nome)) return false;
    return Objects.equals(this.departamento, other.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;
    }
}
```

# 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");  
  
        System.out.println("Mostrando os dados iniciais ...");  
        System.out.println("novo: " + novo);  
        System.out.println("emp: " + emp);  
    }  
}
```

## DriverEmpregado.java (cont.)

```
novo.setNome("Gisele");  
System.out.println("Modificando nome de novo ...");  
System.out.println("novo: " + novo);  
System.out.println("emp: " + emp);  
  
System.out.println("clonando emp em novo2 e modificando dados  
    de novo2 ...");  
var novo2 = (Empregado)emp.clone();  
novo2.getDepartamento().setNome("AEDS I");  
novo2.setNome("Mario");  
System.out.println(novo2);  
System.out.println(emp);
```



## DriverEmpregado.java (cont.)

```
        System.out.println("clonando emp em novo3 e comparando conteú  
        dos");  
        var novo3 = (Empregado)emp.clone();  
        if (emp.equals(novo3))  
            System.out.println("Conteudos Iguais");  
        else  
            System.out.println("Conteudos Diferentes");  
    }  
}
```

## DriverEmpregado.java (cont.)

Iguais

Mostrando os dados iniciais ...

```
novo: Empregado{id=101, nome=Mario, dep.={id=1, nome=Computacao}}
```

```
emp: Empregado{id=101, nome=Mario, dep.={id=1, nome=Computacao}}
```

Modificando nome de novo ...

```
novo: Empregado{id=101, nome=Gisele, dep.={id=1, nome=Computacao}}
```

```
emp: Empregado{id=101, nome=Gisele, dep.={id=1, nome=Computacao}}
```

clonando emp em novo2 e modificando dados de novo2 ...

```
Empregado{id=101, nome=Mario, dep.={id=1, nome=AEDS I}}
```

```
Empregado{id=101, nome=Gisele, dep.={id=1, nome=Computacao}}
```

clonando emp em novo3 e comparando c o n t e d o s

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
    - ▶ Implícita 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

- ▶ 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.



## Exercícios (cont.)

```
public class Equipamento {  
    private boolean ligado;  
    public Equipamento() {  
        ligado = false;  
    }  
    public void liga() {  
        ligado = true;  
    }  
    public void desliga() {  
        ligado = false;  
    }  
    @Override  
    public String toString() {  
        return "Equipamento{" + "ligado=" + ligado + '}';  
    }  
}
```

## Exercícios (cont.)

```
public class EquipamentoSonoro extends Equipamento{
    private short volume;
    private boolean stereo;
    public EquipamentoSonoro() {
    }
    public short getVolume() {
        return volume;
    }
    public void setVolume(short volume) {
        this.volume = volume;
    }
    public boolean isStereo() {
        return stereo;
    }
    public void setStereo(boolean stereo) {
        this.stereo = stereo;
    }
    public void mono(){
        stereo = false;
    }
}
```

## Exercícios (cont.)

```
    }  
    public void stereo(){  
        stereo = true;  
    }  
    @Override  
    public String toString() {  
        return "EquipamentoSonoro{" + "volume=" + volume + ", stereo=" + stereo + '}';  
    }  
}
```

## Exercícios (cont.)

```
public class EquipamentoSonoroDriver {  
  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        // TODO code application logic here  
        EquipamentoSonoro eq = new EquipamentoSonoro();  
        eq.liga();  
        System.out.println(eq);  
    }  
}
```

# Exercícios

- ▶ Classe Ponto
  - ▶ variaveis:  $x$ ,  $y$
  - ▶ métodos: getters/setters e sobrescrever toString
- ▶ Classe Círculo (herda de Ponto)
  - ▶ variaveis: raio
  - ▶ métodos: getters/setters, area e sobrescrever toString
- ▶ Classe Cilindro (herda de Circulo)
  - ▶ variaveis: altura
  - ▶ métodos: getters/setters, area ( $2\pi r^2 + 2\pi rh$ ), volume ( $\pi r^2 h$ ) e sobrescrever toString

## Exercícios (cont.)

```
public class Ponto {  
    protected float x, y;  
    public Ponto(float x, float y) {  
        this.x = x;  
        this.y = y;  
    }  
    public Ponto() {  
        this(0,0);  
    }  
    public void setPonto(float x, float y){  
        this.x = x;  
        this.y = y;  
    }  
    public float getX() {  
        return x;  
    }  
    public float getY() {  
        return y;  
    }  
}
```

## Exercícios (cont.)

```
@Override  
public String toString() {  
    return "Ponto{" + "x=" + x + ", y=" + y + '}';  
}  
}
```

## Exercícios (cont.)

```
public class Circulo extends Ponto{
    protected double raio;
    public Circulo() {
    }
    public Circulo(float x, float y, double raio) {
        super(x, y);
        this.raio = raio;
    }
    public Circulo(double raio) {
        this.raio = raio;
    }
    public double getRaio() {
        return raio;
    }
    public void setRaio(double raio) {
        this.raio = raio;
    }
}
```



## Exercícios (cont.)

```
public double area(){  
    return Math.PI * raio * raio;  
}  
@Override  
public String toString() {  
    return "Circulo{" + "centro=" + super.toString() + "raio=" +  
        raio + '}';  
}  
}
```

## Exercícios (cont.)

```
public class Cilindro extends Circulo{
    protected double altura;
    public Cilindro() {
        this(0,0,0,0);
    }
    public Cilindro(float x, float y, double raio, double altura) {
        super(x, y, raio);
        this.altura = altura;
    }
    public double getAltura() {
        return altura;
    }
    public void setAltura(double altura) {
        this.altura = altura;
    }
}
```

## Exercícios (cont.)

```
public double area(){
    return 2 * super.area() + 2 * Math.PI * raio * altura;
}
public double volume(){
    return super.area() * altura;
}
@Override
public String toString() {
    return "Cilindro{" + super.toString() + "altura=" + altura + '}'
        + ' ';
}
}
```

## Exercícios (cont.)

```
public class PontoCirculoCilindro {  
  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        // TODO code application logic here  
        Cilindro c = new Cilindro(3,4,10,15);  
        System.out.println("Area: " + c.area());  
        System.out.println("Volume: " + c.volume());  
    }  
}
```

FIM