# Herança

## SCC0604 - Programação Orientada a Objetos

Prof. Fernando V. Paulovich
http://www.icmc.usp.br/~paulovic
*paulovic@icmc.usp.br*

Instituto de Ciências Matemáticas e de Computação(ICMC)
Universidade de São Paulo (USP)

23 de agosto de 2010

# Sumário

# Sumário

# Introdução

- Aqui vamos aprofundar o que foi dado anteriormente, apresentando o mecanismo necessário para derivar classes existentes, chamado de **Herança**

# Introdução

- Aqui vamos aprofundar o que foi dado anteriormente, apresentando o mecanismo necessário para derivar classes existentes, chamado de **Herança**
- Lembre-se de que o mecanismo de herança serve para **reutilizar ou alterar os métodos de classes já existentes**, bem como adicionar novos atributos e métodos a fim de **adaptar as classes a novas situações**

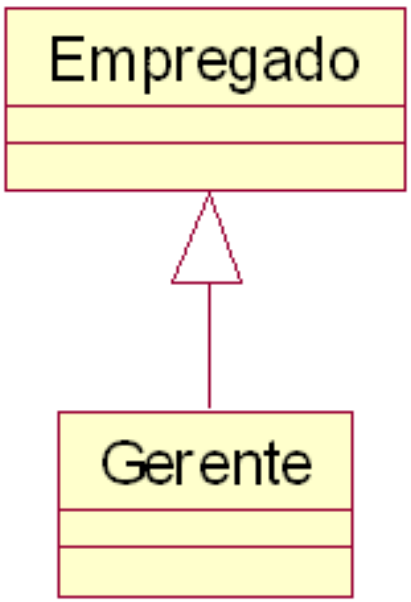
# Primeiros Passos com Herança

- Para a implementação de uma herança em Java, a palavra-reservada **extends** deve ser usada

# Primeiros Passos com Herança

- Para a implementação de uma herança em Java, a palavra-reservada **extends** deve ser usada
- O **extends** indica que uma nova classe (classe-derivada) está sendo derivada (criada) de uma classe existente (classe-base)

# Exemplo de Codificação




```
1  public class Empregado {
2  ...
3  }
4
5  public class Gerente extends Empregado {
6  ...
7  }
```

# Sumário

# Herança e Construtores

- Como já foi dito, quando um objeto é instanciado (**new**), um de seus construtores é chamado automaticamente

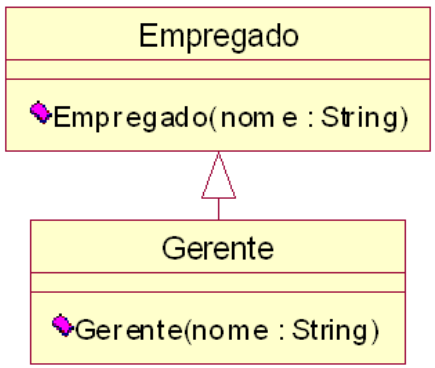
# Herança e Construtores

- Como já foi dito, quando um objeto é instanciado (**new**), um de seus construtores é chamado automaticamente
- Quando uma classe-base possui construtores com parâmetros de entrada, e se deseja passar valores a esses construtores quando um objeto de uma classe-derivada é criado, é necessário usar a palavra-chave **super**

# Herança e Construtores

- Como já foi dito, quando um objeto é instanciado (**new**), um de seus construtores é chamado automaticamente
- Quando uma classe-base possui construtores com parâmetros de entrada, e se deseja passar valores a esses construtores quando um objeto de uma classe-derivada é criado, é necessário usar a palavra-chave **super**
- O **super** é simplesmente uma referência aos elementos da classe-base

# Herança e Construtores




```
1  public class Empregado {
2    private String nome;
3    ...
4    public Empregado(String nome) {
5      this.nome = nome;
6    }
7  }
8
9  public class Gerente extends Empregado {
10   ...
11   public Gerente(String nome) {
12     super(nome);
13   }
14 }
```

# Herança e Construtores

- É possível passar argumentos extra no construtor da classe-derivada

```
1  public class Empregado {
2    private String nome;
3    ...
4    public Empregado(String nome) {
5      this.nome = nome;
6    }
7  }
8
9  public class Gerente extends Empregado {
10   private double bonus,
11   ...
12   public Gerente(String nome, double bonus) {
13     super(nome);
14     this.bonus = bonus;
15   }
16 }
```

# Observações sobre Herança

- A chamada a **super** deve ser a primeira instrução do construtor da classe-derivada

# Observações sobre Herança

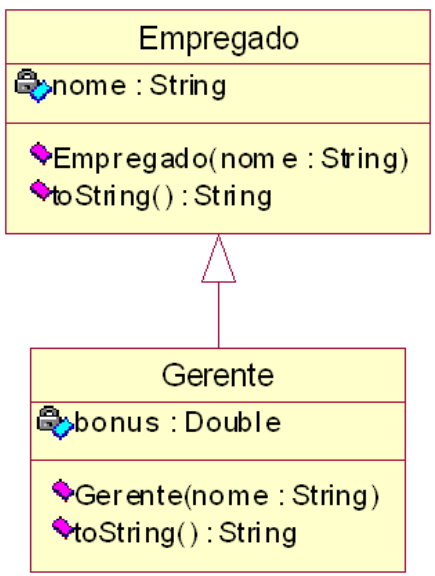
- A chamada a **super** deve ser a primeira instrução do construtor da classe-derivada
- Se o construtor da classe-derivada não chamar o construtor da classe-base explicitamente, então a classe-base usa seu construtor padrão (sem argumentos)

# Observações sobre Herança

- A chamada a **super** deve ser a primeira instrução do construtor da classe-derivada
- Se o construtor da classe-derivada não chamar o construtor da classe-base explicitamente, então a classe-base usa seu construtor padrão (sem argumentos)
- Se a classe-base **não tiver construtor padrão**, então o **compilador acusará um erro**.

# Uso do **super**

- O **super** pode ser usado para referenciar tanto atributos quanto métodos da classe-base




```
1  public class Empregado {
2    private String nome;
3    ...
4    public String toString() {
5       return nome;
6    }
7  }
8
9  public class Gerente extends Empregado {
10   private double bonus;
11   ...
12   public String toString() {
13      String aux = super.toString() + " - " + bonus;
14      return aux;
15   }
16 }
```

# Sumário

# Trabalhando com Subclasses

- Qualquer **objeto** de uma **classe-derivada** pode ser **utilizado no lugar** de um objeto da **classe-base**

# Trabalhando com Subclasses

- Qualquer **objeto** de uma **classe-derivada** pode ser **utilizado no lugar** de um objeto da **classe-base**
- Assim, podemos **atribuir** um objeto de uma **classe-derivada** a uma variável da **classe-base**

# Trabalhando com Subclasses

```
1  Empregado[] emp = new Empregado[3];
2  emp[0] = new Empregado("Empregado 1");
3  emp[1] = new Empregado("Empregado 2");
4  emp[2] = new Gerente("Empregado 3",20);
5
6  for(int i = 0; i < emp.length; i++) {
7      System.out.println(emp[i]);
8  }
```

# Trabalhando com Subclasses

- De forma geral, o inverso é falso: **um objeto de classe-base não pode ser atribuído a um objeto de sub-classe**

# Trabalhando com Subclasses

- De forma geral, o inverso é falso: **um objeto de classe-base não pode ser atribuído a um objeto de sub-classe**
- A seguinte instrução causa um erro

# Trabalhando com Subclasses

- De forma geral, o inverso é falso: **um objeto de classe-base não pode ser atribuído a um objeto de sub-classe**
- A seguinte instrução causa um erro

# Trabalhando com Subclasses

- De forma geral, o inverso é falso: **um objeto de classe-base não pode ser atribuído a um objeto de sub-classe**
- A seguinte instrução causa um erro

```
1 Gerente chefe = emp[2];
```

# Conversão de Tipo Explícita (*Cast*)

- Para se converter um tipo em outro, Java provê um mecanismo chamado *casting*

# Conversão de Tipo Explícita (*Cast*)

- Para se converter um tipo em outro, Java provê um mecanismo chamado *casting*
- Para isso, coloque o tipo ao qual se queira converter entre parênteses e coloque-o a frente do tipo que se queira converter

# Conversão de Tipo Explícita (*Cast*)

- Para se converter um tipo em outro, Java provê um mecanismo chamado *casting*
- Para isso, coloque o tipo ao qual se queira converter entre parênteses e coloque-o a frente do tipo que se queira converter

# Conversão de Tipo Explícita (*Cast*)

- Para se converter um tipo em outro, Java provê um mecanismo chamado *casting*
- Para isso, coloque o tipo ao qual se queira converter entre parênteses e coloque-o a frente do tipo que se queira converter

```
1 Gerente chefe = (Gerente)emp[2];
```

# Conversão de Tipo Explícita (*Cast*)

- Na verdade, o comando anterior pode ser executado porque **emp[2]** faz referência a um objeto da classe **Gerente**. Caso contrário o sistema em tempo de execução geraria uma exceção

# Conversão de Tipo Explícita (*Cast*)

- Na verdade, o comando anterior pode ser executado porque **emp[2]** faz referência a um objeto da classe **Gerente**. Caso contrário o sistema em tempo de execução geraria uma exceção
- Assim, uma boa prática de programação é **testar qual tipo de objeto** uma variável faz referência antes de fazer o *casting*

# Conversão de Tipo Explícita (*Cast*)

- Na verdade, o comando anterior pode ser executado porque **emp[2]** faz referência a um objeto da classe **Gerente**. Caso contrário o sistema em tempo de execução geraria uma exceção
- Assim, uma boa prática de programação é **testar qual tipo de objeto** uma variável faz referência antes de fazer o *casting*
- Isso pode ser feito usando o operador **instanceof**

# Aplicando o **instanceof**

```
1 if(emp[2] instanceof Gerente) {
2   Gerente chefe = (Gerente)emp[2];
3   System.out.println(g)
4 }
```

# Observações sobre *Casting*

- O *casting* é necessário quando um **objeto da classe-derivada for atribuído a um objeto da classe-base**

# Observações sobre *Casting*

- O *casting* é necessário quando um **objeto da classe-derivada for atribuído a um objeto da classe-base**
- Pode-se fazer um *casting* somente **dentro de uma hierarquia de heranças**

# Observações sobre *Casting*

- O *casting* é necessário quando um **objeto da classe-derivada for atribuído a um objeto da classe-base**
- Pode-se fazer um *casting* somente **dentro de uma hierarquia de heranças**
- Deve-se **evitar o uso** de *casting* em um programa, normalmente ele só deve ser empregado com um contêiner

# Polimorfismo

## Polimorfismo

- Quando um método da classe-derivada é solicitado para ser executado, primeiro a classe-derivada verifica se ela tem um método com esse nome e com exatamente os mesmo parâmetros. Se tiver, o mesmo é usado, caso contrário essa solicitação é passada para a classe-base. Se a classe-base tiver tal método, esse método é usado, de outra forma um erro em tempo de compilação é retornado

# Polimorfismo

## Polimorfismo

- Quando um método da classe-derivada é solicitado para ser executado, primeiro a classe-derivada verifica se ela tem um método com esse nome e com exatamente os mesmo parâmetros. Se tiver, o mesmo é usado, caso contrário essa solicitação é passada para a classe-base. Se a classe-base tiver tal método, esse método é usado, de outra forma um erro em tempo de compilação é retornado
- Assim, temos que um método definido em uma classe-derivada com o **mesmo nome e lista de parâmetros** que um método da classe-base, **oculta o método da classe-base**

# Exemplo de Polimorfismo

```
1 ...
2 Gerente chefe = new Gerente("CHEFE");
3 chefe.toString(); //chama método da classe-derivada
4 ...
```

# Sumário

# Como Evitar Herança

## Classes e Métodos Finais (**final**)

- Devido ao **polimorfismo**, um programa pode definir em **tempo de execução** qual será seu comportamento - isso se chama **ligação dinâmica**;

# Como Evitar Herança

## Classes e Métodos Finais (**final**)

- Devido ao **polimorfismo**, um programa pode definir em **tempo de execução** qual será seu comportamento - isso se chama **ligação dinâmica**;
- Esse processo de **determinar dinamicamente** qual método executar torna o processo de execução de um método, se comparado a uma determinação estática, **mais lento**

# Como Evitar Herança

## Classes e Métodos Finais (**final**)

- Devido ao **polimorfismo**, um programa pode definir em **tempo de execução** qual será seu comportamento - isso se chama **ligação dinâmica**;
- Esse processo de **determinar dinamicamente** qual método executar torna o processo de execução de um método, se comparado a uma determinação estática, **mais lento**
- Assim, se um **método** de uma classe-base **nunca for sobreposto** por um método de uma classe-derivada, nós podemos **indicar isso ao compilador**

# Como Evitar Herança

## Classes e Métodos Finais (**final**)

- Devido ao **polimorfismo**, um programa pode definir em **tempo de execução** qual será seu comportamento - isso se chama **ligação dinâmica**;
- Esse processo de **determinar dinamicamente** qual método executar torna o processo de execução de um método, se comparado a uma determinação estática, **mais lento**
- Assim, se um **método** de uma classe-base **nunca for sobreposto** por um método de uma classe-derivada, nós podemos **indicar isso ao compilador**
- Para indicar que um método **nunca será sobreposto**, o mesmo deve ser definido como **final**

# Indicando que um Método é **final**

```
1  public class Empregado {
2    private String nome;
3
4    public final void setNome(String nome){
5      this.nome = nome;
6    }
7  }
```

# Criando uma Classe **final**

- Para indicar que uma **classe** como um todo **não terá métodos sobrecarregados**, determinando que essa classe **não terá classes-derivadas**, a mesma deve ser declarada como **final**

```
1 public final class Cliente {
2 ...
3 }
```

# Classes Abstratas

- Ao **subir na hierarquia** de heranças, as classe tornam-se mais **genéricas** a tal ponto que não representem algo tangível ou visível. Elas acabam se tornando **modelos para classes**

# Classes Abstratas

- Ao **subir na hierarquia** de heranças, as classe tornam-se mais **genéricas** a tal ponto que não representem algo tangível ou visível. Elas acabam se tornando **modelos para classes**
- Nesses casos, pode ser desejável que **não se permita que objetos sejam instanciados** a partir dessas classes

# Classes Abstratas

- Ao **subir na hierarquia** de heranças, as classe tornam-se mais **genéricas** a tal ponto que não representem algo tangível ou visível. Elas acabam se tornando **modelos para classes**
- Nesses casos, pode ser desejável que **não se permita que objetos sejam instanciados** a partir dessas classes
- Por exemplo, **não faz sentido** que **objetos** sejam declarados a partir de classes como **Forma ou Pessoa**

# Classes Abstratas

- Ao **subir na hierarquia** de heranças, as classe tornam-se mais **genéricas** a tal ponto que não representem algo tangível ou visível. Elas acabam se tornando **modelos para classes**
- Nesses casos, pode ser desejável que **não se permita que objetos sejam instanciados** a partir dessas classes
- Por exemplo, **não faz sentido** que **objetos** sejam declarados a partir de classes como **Forma ou Pessoa**
- Classes que não adimitem objetos devem ser definidas como **abstratas**

# Classes Abstratas

- Em Java usa-se a palavra-chave **abstract** para indicar que uma classe é abstrata

```
1 public abstract class Forma {
2 ...
3 }
4
5 public abstract class Pessoa {
6 ...
7 }
```

# Métodos Abstratos

- Na verdade, uma **classe** normalmente (mas nem sempre) é declarada como **abstrata** quando a mesma **apresenta métodos abstratos**

# Métodos Abstratos

- Na verdade, uma **classe** normalmente (mas nem sempre) é declarada como **abstrata** quando a mesma **apresenta métodos abstratos**
- Um **método abstrato** é um método que **não apresenta implementação**, sendo responsabilidade das classes-derivadas a implementação do mesmo

# Métodos Abstratos

- Na verdade, uma **classe** normalmente (mas nem sempre) é declarada como **abstrata** quando a mesma **apresenta métodos abstratos**
- Um **método abstrato** é um método que **não apresenta implementação**, sendo responsabilidade das classes-derivadas a implementação do mesmo
- Por exemplo, na classe **Forma** podemos declarar o método **desenhar()** como **abstrato**, indicando que as **classes-derivadas deverão implementá-lo** - não faz sentido esse método ser implementado em Forma

# Métodos Abstratos

- Na verdade, uma **classe** normalmente (mas nem sempre) é declarada como **abstrata** quando a mesma **apresenta métodos abstratos**
- Um **método abstrato** é um método que **não apresenta implementação**, sendo responsabilidade das classes-derivadas a implementação do mesmo
- Por exemplo, na classe **Forma** podemos declarar o método **desenhar()** como **abstrato**, indicando que as **classes-derivadas deverão implementá-lo** - não faz sentido esse método ser implementado em Forma
- Um método é declarado como abstrato usando-se a palavra-chave **abstract**.

# Métodos Abstratos

```
1  public abstract class Forma {
2      ...
3     public abstract void desenhar();
4     ...
5  }
6
7  public class Quadrado extends Forma {
8    public void desenhar() {
9      //implementação de desenhar um quadrado
10     ...
11   }
12 }
```

# Exemplo

```
1  public abstract class Empregado {
2     protected String nome;
3     ...
4
5     public abstract String toString();
6
7     public void print() {
8         System.out.println(this.toString());
9     }
10 }
```

# Exemplo

```
1  public class Gerente extends Empregado {
2     ....
3
4     public String toString() {
5         String aux = this.nome + " - " + this.bonus;
6         return aux;
7     }
8  }
```

# Exemplo

```
1  public class Principal {
2    public static void main(String[] args) {
3      Empregado[] emp = new Empregado[3];
4      emp[0] = new Gerente("Empregado 1", 10);
5      emp[1] = new Gerente("Empregado 2", 20);
6      emp[2] = new Gerente("Empregado 3", 30);
7
8      for(int i = 0; i < emp.length; i++) {
9        emp[i].print();
10     }
11   }
12 }
```

# Acesso Protegido

- Por motivos já discutidos, os **atributos** de uma classe **devem ser declarados como privados**

# Acesso Protegido

- Por motivos já discutidos, os **atributos** de uma classe **devem ser declarados como privados**
- Porém, nesses casos as **classes-derivadas** também **não terão acesso a tal atributo**

# Acesso Protegido

- Por motivos já discutidos, os **atributos** de uma classe **devem ser declarados como privados**
- Porém, nesses casos as **classes-derivadas** também **não terão acesso a tal atributo**
- De forma a **prover acesso e ainda restringir** o mesmo a outras partes do sistema, os **atributos** de uma classe podem ser declarados como protegidos (**protected**)

# Acesso Protegido

- Por motivos já discutidos, os **atributos** de uma classe **devem ser declarados como privados**
- Porém, nesses casos as **classes-derivadas** também **não terão acesso a tal atributo**
- De forma a **prover acesso e ainda restringir** o mesmo a outras partes do sistema, os **atributos** de uma classe podem ser declarados como protegidos (**protected**)
- Na verdade, declarando um atributo como **protected**, ele **também será visível a todas classe do mesmo pacote**

# Acesso Protegido

- Por motivos já discutidos, os **atributos** de uma classe **devem ser declarados como privados**
- Porém, nesses casos as **classes-derivadas** também **não terão acesso a tal atributo**
- De forma a **prover acesso e ainda restringir** o mesmo a outras partes do sistema, os **atributos** de uma classe podem ser declarados como protegidos (**protected**)
- Na verdade, declarando um atributo como **protected**, ele **também será visível a todas classe do mesmo pacote**
- Os **métodos** de uma classe também podem ser declarado com visibilidade **protected**

# Resumo dos Modificadores de Acesso

## Tipos de Modificadores em Java

- **private** - visibilidade somente para a classe

# Resumo dos Modificadores de Acesso

## Tipos de Modificadores em Java

- **private** - visibilidade somente para a classe
- **public** - visibilidade para qualquer entidade do sistema

# Resumo dos Modificadores de Acesso

## Tipos de Modificadores em Java

- **private** - visibilidade somente para a classe
- **public** - visibilidade para qualquer entidade do sistema
- **protected** - visibilidade para o pacote e todas as classes-derivadas

# Resumo dos Modificadores de Acesso

## Tipos de Modificadores em Java

- **private** - visibilidade somente para a classe
- **public** - visibilidade para qualquer entidade do sistema
- **protected** - visibilidade para o pacote e todas as classes-derivadas
- Nenhum modificador - visibilidade para o pacote

# Sumário

# **Object**: A Superclasse Cósmica

- Se uma classe não for derivada explicitamente de nenhuma classe, a mesma estará **derivando implicitamente** da classe **Object**, não sendo necessário indicar essa herança no código

# Object: A Superclasse Cósmica

- Se uma classe não for derivada explicitamente de nenhuma classe, a mesma estará **derivando implicitamente** da classe **Object**, não sendo necessário indicar essa herança no código
- Assim, implicitamente **toda classe é derivada de Object**, de forma que uma variável do tipo **Object pode referenciar qualquer objeto**

# Object: A Superclasse Cósmica

- Se uma classe não for derivada explicitamente de nenhuma classe, a mesma estará **derivando implicitamente** da classe **Object**, não sendo necessário indicar essa herança no código
- Assim, implicitamente **toda classe é derivada de Object**, de forma que uma variável do tipo **Object pode referenciar qualquer objeto**
- Quando for necessário trabalhar com **programação genérica**, **Object** é a escolha

# Object: A Superclasse Cósmica

## Alguns métodos oferecidos pela classe **Object**

- public final Class getClass()

# **Object**: A Superclasse Cósmica

## Alguns métodos oferecidos pela classe **Object**

- public final Class getClass()
- public boolean equals(Object obj)

# **Object**: A Superclasse Cósmica

## Alguns métodos oferecidos pela classe **Object**

- public final Class getClass()
- public boolean equals(Object obj)
- public int hashCode()

# Object: A Superclasse Cósmica

## Alguns métodos oferecidos pela classe **Object**

- public final Class getClass()
- public boolean equals(Object obj)
- public int hashCode()
- protected void finalize() throws Throwable

# **Object**: A Superclasse Cósmica

## Alguns métodos oferecidos pela classe **Object**

- public final Class getClass()
- public boolean equals(Object obj)
- public int hashCode()
- protected void finalize() throws Throwable
- protected Object clone() throws CloneNotSupportedException

# Object: A Superclasse Cósmica

## Alguns métodos oferecidos pela classe **Object**

- public final Class getClass()
- public boolean equals(Object obj)
- public int hashCode()
- protected void finalize() throws Throwable
- protected Object clone() throws CloneNotSupportedException
- public String toString()

# **Object**: A Superclasse Cósmica

## Alguns métodos oferecidos pela classe **Object**

- public final Class getClass()
- public boolean equals(Object obj)
- public int hashCode()
- protected void finalize() throws Throwable
- protected Object clone() throws CloneNotSupportedException
- public String toString()
- ...

# Exemplo: Programação Genérica

```
1  public class Util {
2      public static int find(Object[] array, Object key) {
3          for(int i = 0; i < array.length; i++) {
4              if(array[i].equals(key)) {
5                return i;
6              }
7          }
8
9          return -1;
10     }
11 }
```

# Exemplo: Programação Genérica

```
1  public class Gerente extends Empregado {
2      ...
3      @Override
4      public boolean equals(Object obj) {
5       if(obj instanceof Gerente) {
6         Gerente aux = (Gerente)obj;
7         return (this.nome.equals(aux.nome) &&
8                         this.bonus == aux.bonus);
9
10      } else {
11        return false;
12      }
13     }
14 }
```

# Exemplo: Programação Genérica

```
1  public class Principal {
2      public static void main(String[] args) {
3          Empregado[] emp = new Empregado[3];
4          emp[0] = new Gerente("Empregado 1", 10);
5          emp[1] = new Gerente("Empregado 2", 10);
6          emp[2] = new Gerente("Empregado 3", 30);
7
8          Gerente chefe = new Gerente("Empregado 3", 30);
9          System.out.println(Util.find(emp,chefe));
10     }
11 }
```

# Sumário

# Recomendações de Projeto com Herança

- Coloque métodos e atributos comuns nas classes-base

# Recomendações de Projeto com Herança

- Coloque métodos e atributos comuns nas classes-base
- Use herança para modelar uma relação “é-um-tipo-de”

# Recomendações de Projeto com Herança

- Coloque métodos e atributos comuns nas classes-base
- Use herança para modelar uma relação "é-um-tipo-de"
- Não use herança a menos que todos os métodos herdados façam sentido

# Recomendações de Projeto com Herança

- Coloque métodos e atributos comuns nas classes-base
- Use herança para modelar uma relação "é-um-tipo-de"
- Não use herança a menos que todos os métodos herdados façam sentido
- Use polimorfismo, não informação de tipo

# Resumo

| | **Objetos** | **Herança** | **Métodos** | **Atributos** |
|---|---|---|---|---|
| **Classe Abstrata** | Não pode ter instâncias | Pode ser estendida (**extends**) | Métodos concretos e abstratos | Constantes e atributos |
| **Classe Final** | Pode ter instâncias | Não pode ser estendida | Somente métodos concretos | Constantes e atributos |