Programação Orientada a objetos - Java

Os temas abordados serão:

- O conceito de classe e objeto
- Acesso aos atributos e métodos de uma classe
- Memoria em Java (Stack e Heap)
- Encapsulamento
- Contexto Estático
- Herança
- Sobrecarga de Métodos
- Sobreposição de métodos
- Polimorfismo

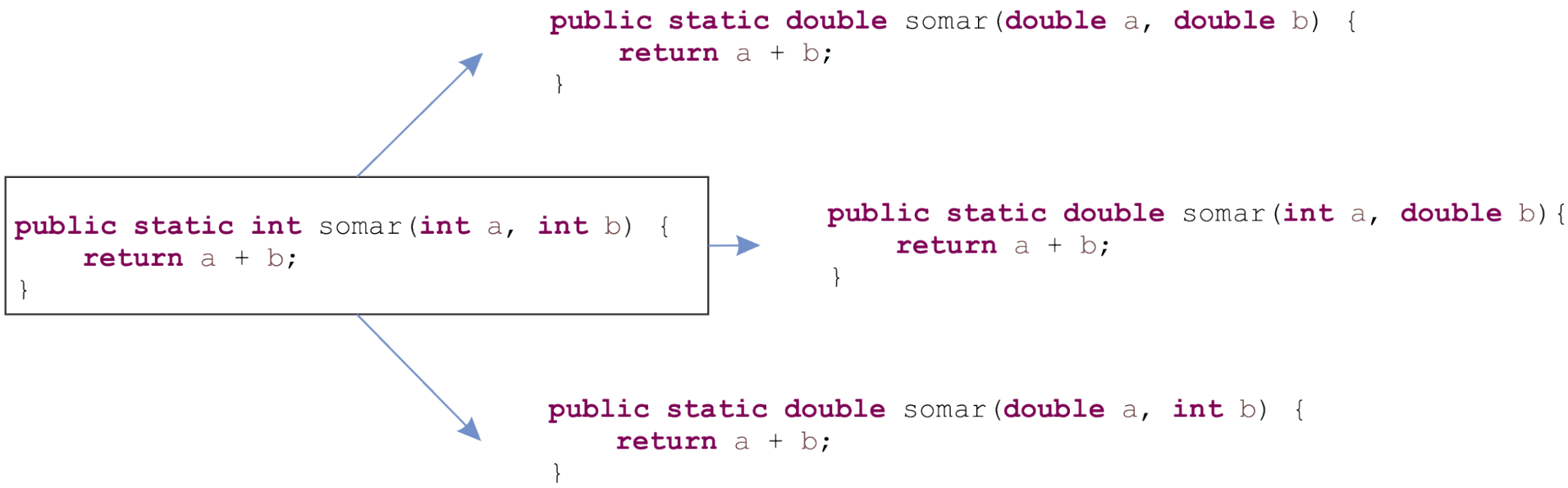
.

Sobrecarga de métodos

Consiste em permitir dentro da mesma classe, mais de um método com o mesmo nome, desde que a assinatura do método seja diferente a cada vez.

Assinatura = nome_Método + lista_de_argumentos

Sobrecarga de métodos



Sobrecarga de métodos

- Consiste em criar métodos com o mesmo nome, sempre que tenham tipo de parâmetros diferentes.
- A escolha de qual método ir ser invocado pelo programa principal é tomada apenas no tempo de execução.
- Tanto o nome dos argumentos como o tipo de retorno do método não interferem na sobrecarga de métodos

Sobrecarga - Operacoes.java

```java
package sobrecargametodo;
public class Operacoes {
    public static int somar(int a, int b) {
        System.out.println("metodo somar (int, int)");
        return a + b;
    }
    public static double somar(double a, double b) {
        System.out.println("metodo somar (double, double)");
        return a + b;
    }
    public static double somar(int a, double b) {
        System.out.println("metodo somar (int, double)");
        return a + b;
    }
    public static double somar(double a, int b) {
        System.out.println("metodo somar (double, int)");
        return a + b;
    }
}
```

Sobrecarga - SobrecargaMetodos.java

```
package sobrecargametodo;
public class SobrecargaMetodos {
    public static void main(String[] args) {
        System.out.println("Resultado 1: " + Operacoes.somar(3, 4));
        System.out.println("Resultado 2: " + Operacoes.somar(5, 4.1));
        System.out.println("Resultado 3: " + Operacoes.somar(7.1, 3));
        System.out.println("Resultado 4: " + Operacoes.somar(2.2, 6.8));
        System.out.println("Resultado 5: " + Operacoes.somar(3, 1L));
        System.out.println("Resultado 6: " + Operacoes.somar(3F, 'A'));
    }
}
```

```
metodo somar (int, int)
Resultado 1: 7
metodo somar (int, double)
Resultado 2: 9.1
metodo somar (double, int)
Resultado 3: 10.1
metodo somar (double, double)
Resultado 4: 9.0
metodo somar (int, double)
Resultado 5: 4.0
metodo somar (double, int)
Resultado 6: 68.0
```

Sobrecarga de métodos

Tanto polimorfismo quanto herança são referências no ramo de reutilização de código, pois trabalham em conjunto. Existem dois tipos de polimorfismo que são conhecidos como sobrecarga (*overload*) e sobreposição (*override*).

- A sobrecarga (*overload)* consiste em permitir, dentro da mesma classe, mais de um método com o mesmo nome, mas devem possuir argumentos diferentes para funcionar.
- A sobreposição (*override*) possibilita reescrever na classe filha os métodos implementados previamente na classe pai, ou seja, uma classe filha pode redefinir métodos herdados de suas descendentes, mantendo o nome e a assinatura.

Sobreposição de métodos

- Superposição consiste em usar um método que foi definido em uma superclasse, com funções diferentes dentro de uma subclasse.

- Uma subclasse pode modificar o comportamento herdado de uma superclasse, sempre que a assinatura (nome, tipo de retorno, lista de argumentos) do método sobreposto pela subclasse seja igual que a contida na superclasse.

- O único que pode variar na sobreposição de um método é o modificador de acesso, mas não pode ser definido como menos accessível.

Sobreposição de métodos

```
public class Empregado {
    protected String nome;
    protected double salario;

    public String obterdetalhes(){
        return "Nome: " + nome +
               ", salario: " + salario;
    }
```

Uma subclasse pode modificar o comportamento herdado de uma superclasse sempre que sua assinatura do método sobreposto for igual à assinatura da superclasse.

```
public class Gerente extends Empregado {
    private String departamento;

    public String obterdetalhes(){
        return "Nome: " + nome +
               ", salario: " + salario +
               ", departamento: " + departamento;
    }
```

O único que pode mudar na sobreposição do método é o modificador de acesso.

Sobreposição de métodos

Métodos similares devem ter:

- Os mesmos nomes.
- A mesma ordem de argumentos.
- Mesmo tipo para valores retornados.

Sobreposição de métodos - Empregado.java

```
package sobreposicao;
public class Empregado {
    protected String nome;
    protected double salario;
    protected Empregado(String nome, double
salario){
        this.nome = nome;
        this.salario = salario;
    }
    public String obterdetalhes(){
        return "Nome: " + nome +
                ", salario: " + salario;
    }
```

```
    public String getNome() {
        return nome;
    }
    public void setNome(String
nome) {
        this.nome = nome;
    }
    public double getSalario()
{
        return salario;
    }
    public void
setSalario(double salario) {
        this.salario = salario;
    }
}
```

Sobreposição de métodos - Gerente.java

```
package sobreposicao;
public class Gerente extends Empregado {

   private String departamento;

    public Gerente(String nome, double salario, String departamento){
            super(nome, salario);
            this.departamento = departamento;
          }
        public String obterdetalhes(){
            return "Nome: " + nome +
                    ", salario: " + salario +
                    ", departamento: " + departamento;
        }

        public String getDepartamento() {
            return departamento;
        }

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

Sobreposição de métodos - TestSobreposicao.java

```
package sobreposicao;
 public class TestSobreposicao {
    public static void main(String[] args) {
    Empregado empregado = new Empregado("Fernanda",10000);
        System.out.println( empregado.obterdetalhes() );
         Gerente gerente = new Gerente("Karla",2000,"Finanzas");
         System.out.println( gerente.obterdetalhes() );
    }
}
```

```
Nome: Fernanda, salario: 10000.0
Nome: Karla, salario: 2000.0, departamento: Finanzas
```

Polimorfismo

- Polimorfismo é habilidade de executar métodos sintaticamente iguais em tipos distintos com o objetivo de alterar seu comportamento.

- Um objeto criado só pode ter uma forma e um tipo porem uma variável de um tipo pode referenciar objetos diferentes, sempre em quando exista uma relação de herança.

- O método a ser executado será do objeto cuja referência este apontando no tempo de execução.

Polimorfismo

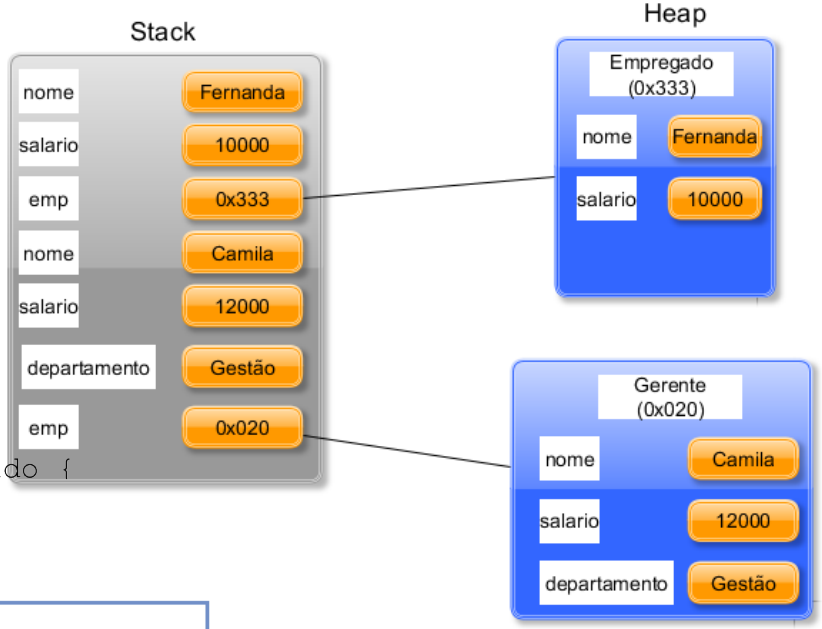
```
public class Empregado {
   protected String nome;
   protected double salario;
```

```
public String obterdetalhes(){
    return "Nome: " + nome +
           ", salario: " + salario;
}
```

```
public class Gerente extends Empregado {

    private String departamento;
```

```
public String obterdetalhes(){
    return "Nome: " + nome +
           ", salario: " + salario +
           ", departamento: " + departamento;
}
```

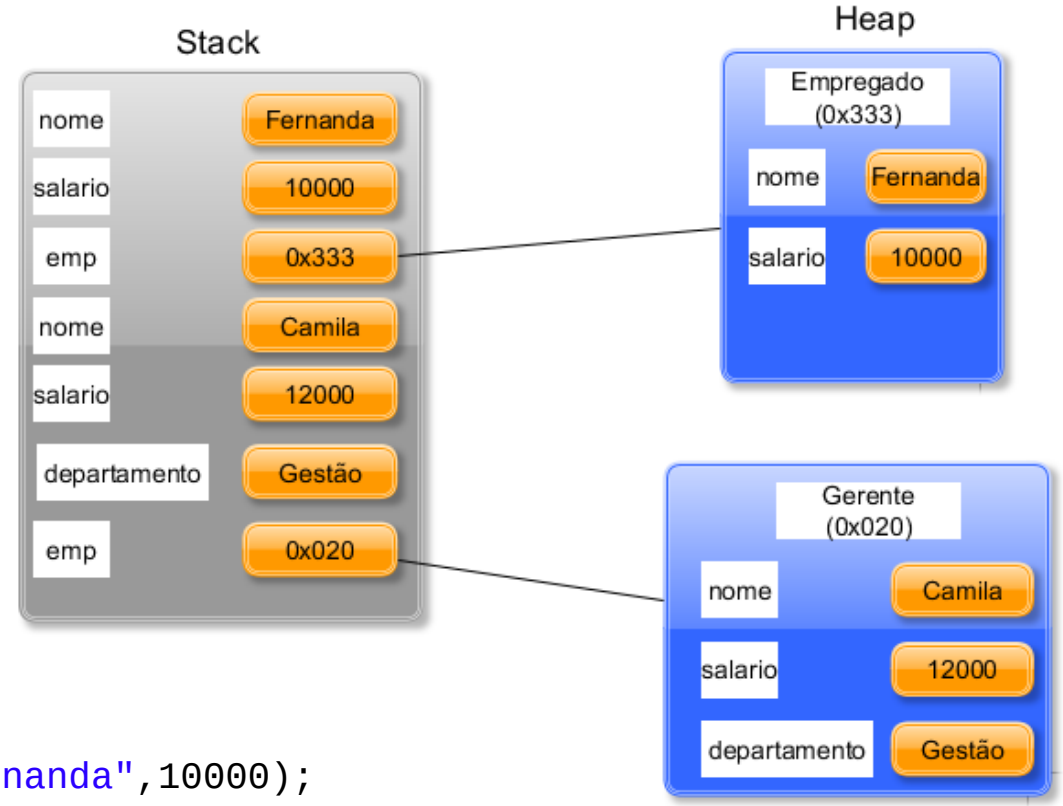



```
Empregado emp = new Empregado("Fernanda",10000);
System.out.println( emp.obterdetalhes() );

emp = new Gerente("Camila",12000,"Gestão");
System.out.println( emp.obterdetalhes() );
```

Polimorfismo

O método executado pertence ao tipo cuja referência este apontando em tempo de execução.



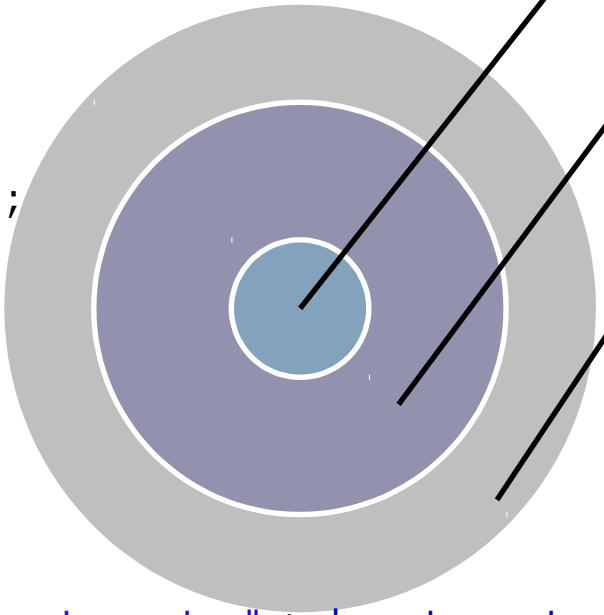

```
Empregado emp = new Empregado("Fernanda",10000);
System.out.println( emp.obterdetalhes() );

emp = new Gerente("Camila",12000,"Gestão");
System.out.println( emp.obterdetalhes() );
```

Polimorfismo

- O método herdado de empregado esta oculto

```
public String obterdetalhes(){
    return "Nome: " + nome +
    ", salario: " + salario +
    ", departamento: " + departamento;
     }
```




```
public String obterdetalhes(){
      return super.obterdetalhes()+ ", departamento: " + departamento;
   }
```

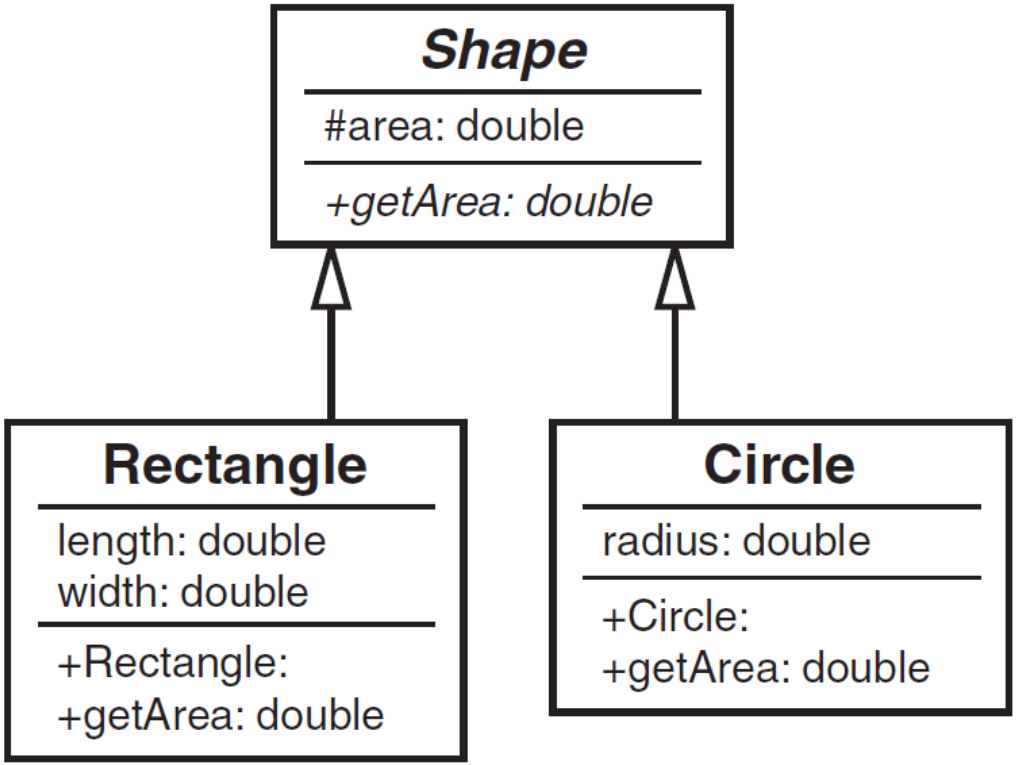
Polimorfismo

- Polimorfismo é habilidade de executar métodos sintaticamente iguais em tipos distintos com o objetivo de alterar seu comportamento.

- Um objeto criado só pode ter uma forma e um tipo porem uma variável de um tipo pode referenciar objetos de diferentes, sempre em quando exista uma relação de herança.

- O método a ser executado será do objeto cuja referência este apontando no tempo de execução.

Polimorfismo



Polimorfismo

```
package area;
public abstract class Shape {
    public abstract void draw();
}
```

```
package area;
public class Circle extends Shape {
    public void draw() {
        System.out.println("I am drawing a Circle");
    }
}
```

Polimorfismo

```
package area;
public class Star extends Shape{
    public void draw() {
        System.out.println("I am drawing a Star");
        }
}
package area;
public class Rectangle extends Shape{

    public void draw() {
        System.out.println("I am drawing a Rectangle");
        }
}
```

Polimorfismo

```
public class TestShape {
    public static void main(String[] args) {
        Circle circle = new Circle();
        Rectangle rectangle = new Rectangle();
        Star star = new Star();
        circle.draw();
        rectangle.draw();
        star.draw();
        drawMe(circle);
        drawMe(rectangle);
        drawMe(star);
    }

    static void drawMe(Shape s) {
        s.draw();
        }
}
```

```
I am drawing a Circle
I am drawing a Rectangle
I am drawing a Star
I am drawing a Circle
I am drawing a Rectangle
I am drawing a Star
```

Polimorfismo

Por médio do polimorfismo podemos implementar sistemas facilmente extensíveis:

- Novas classes podem se adicionadas.
- As únicas partes que precisam ser alteradas são aquelas que exigem um conhecimento das novas classes para elas ser acomodadas.

No exemplo anterior demostramos que um programa pode determinar o tipo de um objeto em tempo de execução.

O polimorfismo permite tratar as generalidades e deixar que o ambiente do tempo de execução trate com as especialidades.

Polimorfismo

O polimorfismo promove extensibilidade:

- O software que invoca o comportamento polimórfico é independente dos tipos de objetos que recebem as mensagens.
- Os objetos podem ser instruídos para se comportarem de maneiras apropriadas.
- Novos tipos de objetos que podem responder ao chamado de métodos existentes podem ser incorporados ao sistema.

Exercício

Empresa tem dois tipos de empregados:

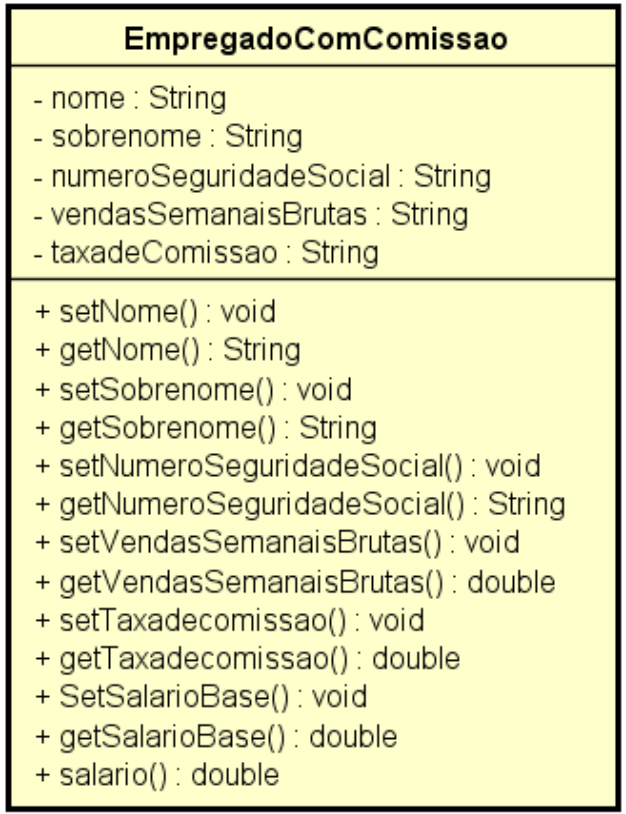
- Empregado por comissão

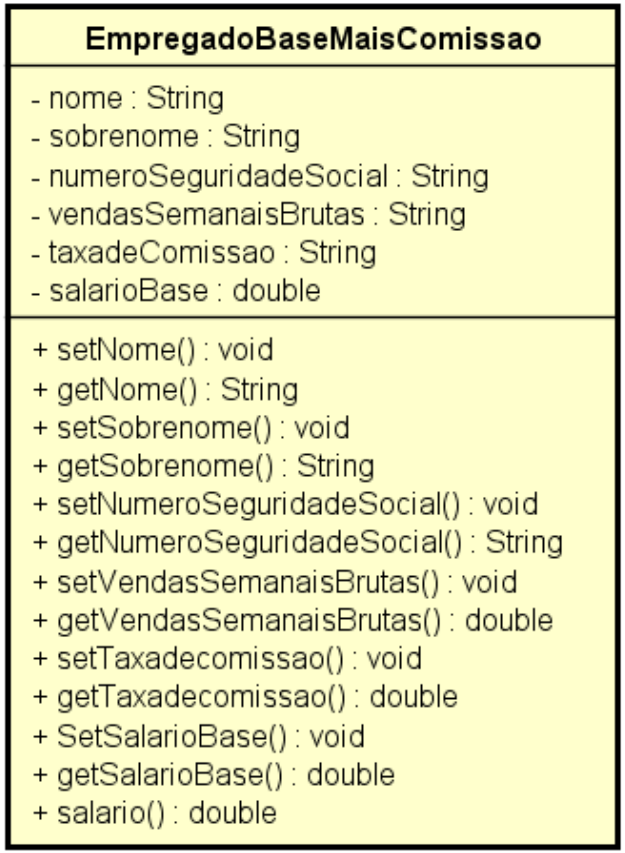
  salário =taxa de comissão * vendas Semanais

- Empregado com salario base e comissão

  salário = Salario Base +(taxa de comissão * vendas)

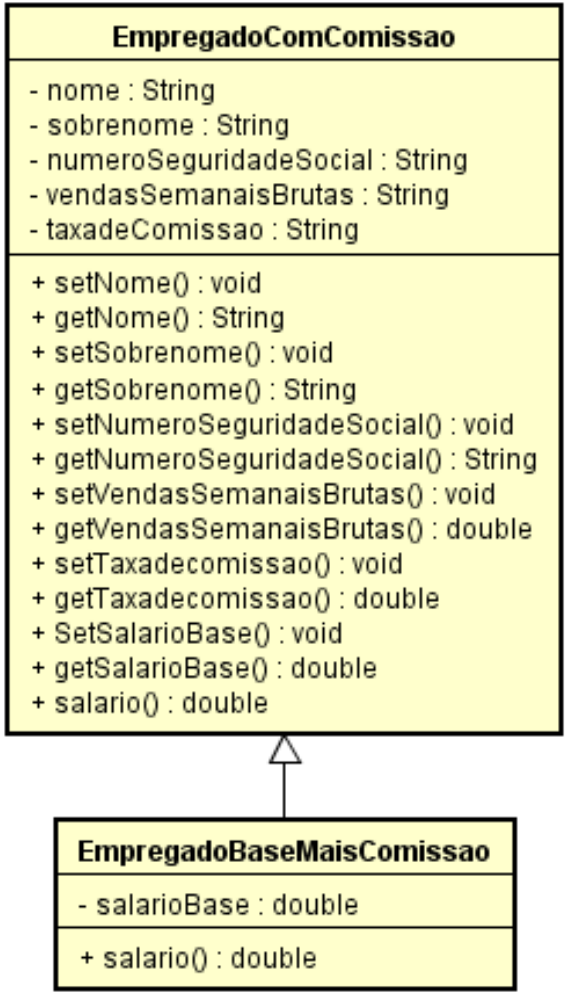
Exercício


EmpregadoComComissao
- nome : String
- sobrenome : String
- numeroSeguridadeSocial : String
- vendasSemanaisBrutas : String
- taxadeComissao : String
+ setNome() : void
+ getNome() : String
+ setSobrenome() : void
+ getSobrenome() : String
+ setNumeroSeguridadeSocial() : void
+ getNumeroSeguridadeSocial() : String
+ setVendasSemanaisBrutas() : void
+ getVendasSemanaisBrutas() : double
+ setTaxadecomissao() : void
+ getTaxadecomissao() : double
+ SetSalarioBase() : void
+ getSalarioBase() : double
+ salario() : double


EmpregadoBaseMaisComissao
- nome : String
- sobrenome : String
- numeroSeguridadeSocial : String
- vendasSemanaisBrutas : String
- taxadeComissao : String
- salarioBase : double
+ setNome() : void
+ getNome() : String
+ setSobrenome() : void
+ getSobrenome() : String
+ setNumeroSeguridadeSocial() : void
+ getNumeroSeguridadeSocial() : String
+ setVendasSemanaisBrutas() : void
+ getVendasSemanaisBrutas() : double
+ setTaxadecomissao() : void
+ getTaxadecomissao() : double
+ SetSalarioBase() : void
+ getSalarioBase() : double
+ salario() : double

Exercício

Vemos que é muito mais eficiente criar uma extensão da classe `EmpregadoComComissão` para herdar atributos e construir uma subclasse de `EmpregadocomBaseMaisComissao`



EmpregadoComComissao.java

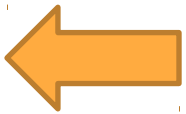
```
package polimorfismo;
public class EmpregadoComComissao {
    private String nome;
    private String sobrenome;
    private String numeroSeguridadeSocial;
    private double vendasSemanaisBrutas;
    private double taxadeComissao;
```

`extends Object` do pacote java.lang

Todas as classes, sem exceção, herdam de `Object`, seja direta ou indiretamente. Então, qualquer objeto possui todos os métodos declarados na classe `Object`.

EmpregadoComComissao.java

```
public EmpregadoComComissao(String nome,
 String sobrenome,
 String numeroSeguridadeSocial,
 double vendasSemanaisBrutas,
double taxadeComissao) {
    this.nome = nome;
    this.sobrenome =sobrenome;
    this.numeroSeguridadeSocial=numeroSeguridadeSocial;
    this.vendasSemanaisBrutas =vendasSemanaisBrutas;
    this.taxadeComissao =taxadeComissao;
}
```



Chamada implícita

Os construtores não são herdados, mais os construtores de uma superclasse estão disponíveis para as subclasses.
Toda subclasse, explicita ou implicitamente chama o construtor da superclasse para assegurar que as variáveis de instância herdadas da superclasse sejam inicializadas adequadamente.

EmpregadoComComissao.java

```
public void setNome(String nome) {
    this.nome = nome;
    }
public String getNome() {
    return nome;
    }
public void setSobrenome(String sobrenome) {
    this.sobrenome = sobrenome;
    }
public String getSobrenome() {
    return sobrenome;
    }
public String getNumeroSeguridadeSocial() {
    return numeroSeguridadeSocial;
    }
```

Declaram métodos *get* e *set public* para todas as variáveis de instancia da classe declaradas.

EmpregadoComComissao.java

```
public void setNumeroSeguridadeSocial(String numeroSeguridadeSocial) {
    this.numeroSeguridadeSocial = numeroSeguridadeSocial;
    }
public double getVendasSemanaisBrutas() {
    return vendasSemanaisBrutas;
    }
public void setVendasSemanaisBrutas(double vendasSemanaisBrutas) {
    this.vendasSemanaisBrutas = (vendasSemanaisBrutas <0.0 )? 0.0 :vendasSemanaisBrutas;
    }
public double getTaxadeComissao() {
        return taxadeComissao;
    }
public void setTaxadecomissao(double taxadeComissao) {
    this.taxadeComissao = taxadeComissao;
    }
```

EmpregadoComComissao.java

```
@Override
public String toString() {

return String.format("Empregado Com Comissão: \nNome: "+ this.nome + "\n
Sobrenome: " + this.sobrenome + "\n Número da Segurança Social: "
+this.numeroSeguridadeSocial + "\n Vendas brutas: " + this.vendasSemanaisBrutas +
"\n Taxa de comissão: " + this.taxadeComissao);
}
}
```

Compara a assinatura do método com as assinaturas de método da superclasse.

O método **toString** retorna uma **String** representando um objeto. Ele é chamado implicitamente sempre que um objeto precisa ser convertido em uma representação **String**. Exemplo:

```
ContaCorrente cc = new ContaCorrente();
System.out.println(cc.toString());

ContaCorrente cc = new ContaCorrente();
System.out.println(cc); // O toString é chamado
```

Considerações

- Utilizar uma assinatura de método incorreta ao tentar sobrescrever um método de superclasses, resulta em uma sobrecarga que pode levara a erros.
- Sobrescrever um método com um modificador de acesso mais restrito, resulta em um erro de sintaxes.
- Declarar os métodos sobrescritos com a notação *@Override* para assegurar em tempo de compilação que as assinaturas estão corretamente definidas.

EmpregadoBaseMasComissao.java

```
package polimorfismo;
public class EmpregadoBaseMasComissao extends EmpregadoComComissao{
	private double salarioBase;
	public EmpregadoBaseMasComissao(String nome,
		String sobrenome,
		String numeroSeguridadeSocial,
		double vendasSemanaisBrutas,
		double taxadeComissao,
		double salarioBase) {

		super(nome, sobrenome, numeroSeguridadeSocial, vendasSemanaisBrutas,
		taxadeComissao);
		setSalarioBase(salarioBase);
}
```

Cada construtor de subclasse deve chamar implícita ou explicitamente seu construtor de superclasse para inicializar as variáveis de instancia herdadas da superclasse.

EmpregadoBaseMasComissao.java

```
public double getSalarioBase() {
	return salarioBase;
	}
public void setSalarioBase(double salarioBase) {
	this.salarioBase = salarioBase;
	}
@Override
public double salario()
	{
	return this.salarioBase + (super.getTaxadeComissao()*
super.getVendasSemanaisBrutas());
	}
@Override
public String toString() {
	 return super.toString() + "\nSalario Base: " + this.salarioBase;
	}
}
```

EmpregadoBaseMasComissaoTest.java

Para realizar a comprovação se cria uma classe executora que reúne os seguintes requisitos.

- Instanciar um objeto e invocar o construtor.
- Usar o método `get` da classe `EmpregadoBaseMasComissao` para recuperar os valores de variável de instância do objeto para saída.
- Usar os métodos `SetSalarioBase` para alterar os valores das variáveis de instância.
- Gerar a representação `String` do objeto atualizada.

EmpregadoBaseMasComissaoTest.java

```
package polimorfismo;
public class EmpregadoBaseMasComissaoTest {
	public static void main(String[] args) {
	EmpregadoBaseMasComissao empregado = new EmpregadoBaseMasComissao("Maria",
"Ramirez","17-2235", 2500, 0.04, 300);
	System.out.println("Informacao obtida usando Metodos gets:");
	System.out.println("Nome: " + empregado.getNome());
	System.out.println("Sobrenome: " + empregado.getSobrenome());
	System.out.println("Número de segurança social: "+
empregado.getNumeroSeguridadeSocial());
	System.out.println("Vendas brutas: " + empregado.getVendasSemanaisBrutas());
	System.out.println("A taxa de comissão: "+ empregado.getTaxadeComissao());
	System.out.println("O salario base: "+ empregado.getSalarioBase());

	empregado.setSalarioBase(1000);
	System.out.println("\nAtualizar informação: \n"+ empregado.toString());
	}
}
```

Considerações para Herança

- Com a herança as variáveis de instância e os métodos de todas as classes na hierarquia são declarados em uma superclasse.
- Quando são feitas modificações em nessas características comuns na superclasse, as subclasses herdam por tanto as modificações.
- Um erro de compilação ocorre se um construtor de subclasse chamar um construtor de superclasse com argumentos que não coincidem com o número e os tipos de parâmetros dos construtores definidos na superclasse.

PolimorfismoTest.java

```
package polimorfismo;
public class PolimorfismoTest {
    public static void main(String[] args) {
        //Criam um objeto EmpregadoComComissao e atribuem sua referencia a uma variável.
        EmpregadoComComissao EmpComissao =new
EmpregadoComComissao("Camila","Gomez", "11-111-222", 10000, 0.06);
        //Criam um objeto EmpregadoBaseMasComissao e atribuem sua referencia a uma variável.
        EmpregadoBaseMasComissao EmpBase = new    EmpregadoBaseMasComissao("Fernanda",
"Ramirez", "33-444-555“,15000,0.04,300);
        //Os métodos da superclasse y e subclasse são invocados explicitamente.
        System.out.println("Chamada toString de (Superclasse)"+ EmpComissao.toString());
        System.out.println("Chamada toString de (Subclasse)"+ EmpBase.toString());
        //Invoca toString no objeto de subclasse utilizando a variável de superclasse EmpregadoComComissao
EmpComissao2 = EmpBase;
        //Chama o  método toString da Classe EmpregadoBaseMasComissao.
        System.out.println("Chamada toString de EmpBase (subclasse) usando variavel de
superclasse "+ EmpComissao2.toString());
        }
}
```

Considerações polimorfismo

- Quando uma variável de superclasse contém uma referência a um objeto de subclasse, e essa referência é utilizada para chamar um método, a versão de subclasse do método é chamada.
- O tipo de objeto referenciado real, não o tipo de variável, determina qual é chamado.
- Um objeto de uma subclasse pode ser tratado como um objeto de sua superclasse.

Classes e Métodos abstratos

- O proposito de uma classe abstrata é fornecer uma superclasse apropriada a partir da qual outras classes podem herdar e assim podem compartilhar um design próprio.
- Superclasses abstratas são excessivamente gerais para criar objetos (só especificam o que é comum entre subclasses). As classes concretas fornecem os aspetos específicos para instanciar objetos.
- Uma classe abstrata contém um ou mais métodos abstratos que as subclasses devem sobrescrever para ser concretas. Variáveis de instância e métodos concretos de uma classe abstrata estão sujeitos às regras de herança.