



# BCC221

# Programação Orientada a Objetos

Prof. Marco Antonio M. Carvalho  
2014/2



UFOP

# Endereços Importantes

- Site da disciplina:  
<http://www.decom.ufop.br/marco/>
- Moodle:  
[www.decom.ufop.br/moodle](http://www.decom.ufop.br/moodle)
- Lista de e-mails:  
[bcc221-decom@googlegroups.com](mailto:bcc221-decom@googlegroups.com)
- Para solicitar acesso:  
<http://groups.google.com/group/bcc221-decom>



UFOP



# Avisos



# Avisos

UFOP



# Na aula Passada

UFOP

- Herança
- Especificadores de Acesso
- Classe *Object*
- Exemplo
- Construtores em Subclasses
- Compilação
- Redefinição de Métodos
- Engenharia de Software com Herança



# Na aula de Hoje

UFOP

- Polimorfismo
  - Exemplo
- Classes e métodos abstratos
  - Exemplo Polimorfismo e Downcast
  - Resolução Dinâmica
  - Operador *instanceof*
  - *Downcast*
  - Método *getClass*
- Métodos e Classes *final*
- Criando e Utilizando Interfaces
- Declarando Constantes em Interfaces
- Interfaces Comuns da API Java



# Polimorfismo

UFOP

- O **Polimorfismo** nos *permite programar genericamente* ao invés de *programar especificamente*
  - O ambiente de tempo de execução se encarrega das especificidades;
  - Nos permite criar programar que processam objetos que compartilham uma mesma superclasse como se todos fossem daquela classe;
  - Simplifica a programação
    - Extensibilidade/plugabilidade.



# Polimorfismo

UFOP

- É possível fazer com que objetos se comportem de maneira adequada automaticamente
  - Sem conhecer o tipo do objeto;
  - Desde que os objetos pertençam a uma mesma hierarquia de herança.
- O mesmo nome e assinatura de métodos podem ser utilizados para causar diferentes ações
  - Dependendo do tipo do objeto.

# Exemplo



# TestePolimorfismo.java

UFOP

```
class Super
{
    public void print()
    {
        System.out.printf("Chamada da superclasse\n");
    }
}

class Sub extends Super
{
    //sobrescreve o metodo da superclasse
    public void print()
    {
        System.out.printf("Chamada da subclasse\n");
    }
}
```



# TestePolimorfismo.java

UFOP

```
public class TestePolimorfismo
{
    public static void main(String args[])
    {
        //associa uma referencia da superclasse a uma variavel da superclasse
        Super sup = new Super();
        //associa uma referencia da subclasse a uma variavel da subclasse
        Sub sub = new Sub();
        //associa uma referencia da subclasse a uma variavel da superclasse
        Super poli = new Sub();

        //invoca o metodo da superclasse usando uma variavel da superclasse
        sup.print();
        //invoca o metodo da subclasse usando uma variavel da subclasse
        sub.print();
        //invoca o metodo da subclasse usando uma variavel da superclasse
        poli.print();
    }
}
```



# Saída

UFOP

Chamada da superclasse

Chamada da subclasse

Chamada da subclasse



# Polimorfismo

UFOP

- Quando uma variável da superclasse contém uma referência a um objeto da subclasse, e esta referência é utilizada para invocar um método, a versão da subclasse é utilizada
  - O compilador Java permite isto por causa da relação de herança;
  - Um objeto da subclasse é um objeto da superclasse
    - O contrário não é verdadeiro.
  - Quando o compilador encontra uma chamada a um método através de uma variável, é verificado se o método pode ser chamado, de acordo com a classe da variável;



# Polimorfismo

UFOP

## ■ Continuação:

- Se a classe contiver (ou herdar) uma declaração do método, a chamada é compilada;
- Em tempo de execução, a classe do objeto referido pela variável determina qual método será utilizado.

# Classes e Métodos Abstratos



# Classes e Métodos Abstratos

UFOP

- Algumas classes são utilizadas apenas em níveis altos de hierarquias de herança, sem a necessidade de instanciá-las em nossos programas
  - São as **abstratas**;
  - São classes incompletas, “faltando peças”;
  - Mais genéricas.
- O objetivo das superclasses abstratas é fornecer uma superclasse a partir da qual outras subclasses possam herdar e compartilhar o projeto
  - Preenchendo as “peças faltantes”.



# Classes e Métodos Abstratos

UFOP

- Classes que podem ser implementadas para cada método declarado  
■ São chamadas de classes concretas
- Mais específicas que as classes abstratas.
- Pode-se escrever um método que receba como parâmetro um objeto da superclasse abstrata;
- Este método pode receber qualquer outro objeto da hierarquia de herança e funcionará corretamente.



# Classes e Métodos Abstratos

UFOP

- Para tornar uma classe abstrata, utilizamos a palavra **abstract**:

**public abstract class Classe();** // classe abstrata

- As classes abstratas normalmente contém um ou mais métodos abstratos:

**public abstract void metodo();** // metodo abstrato



# Classes e Métodos Abstratos

UFOP

- Um método abstrato não possui implementação
  - Uma classe que possui um método abstrato deve ser abstrata, mesmo que possua outros métodos concretos;  
Cada subclasse concreta deve fornecer implementações para cada método abstrato da superclasse.
- Construtores e métodos *static* não podem ser declarados abstratos
  - Construtores não podem ser herdados, logo, nunca seriam implementados;
  - Métodos *static* não são associados a objetos, logo, podem ser invocados a partir da classe abstrata.



# Classes e Métodos Abstratos

UFOP

- Embora não possamos instanciá-las, podemos utilizar as classes abstratas para declarar variáveis que mantêm referências a objetos das subclasses concretas
  - Comportamento polimórfico.
- Novamente, as classes abstratas também podem ser utilizadas para invocar seus membros *static*.



# Classes e Métodos Abstratos

UFOP

- Uma subclasse pode herdar a interface ou a implementação de uma superclasse
  - Hierarquias projetadas para **herança de implementação** tendem a ter a funcionalidade nos níveis mais altos da hierarquia
    - Cada nova subclasse herda um ou mais métodos que foram implementados na superclasse.
  - Hierarquias projetadas para **herança de interface** tendem a ter a funcionalidade nos níveis mais baixos da hierarquia
    - Uma superclasse especifica um ou mais métodos abstratos que devem ser sobreescritos especificamente por cada subclasse.

# Exemplo Polimorfismo e *Downcast*



# Exemplo

UFOP

- Vamos revisitar o exemplo de uma empresa que possui diferentes tipos de empregados:
  - Horista
    - Recebe um valor fixo por hora trabalhada;
    - Após 40 horas, recebe 50% a mais nas horas extras.
  - Assalariado
    - Recebe um salário fixo.
  - Comissionado
    - Recebe uma comissão aplicada sobre o valor das vendas realizadas.
  - Comissionado Assalariado
    - Recebe um salário fixo mais uma comissão aplicada sobre o valor das vendas realizadas;



# Exemplo

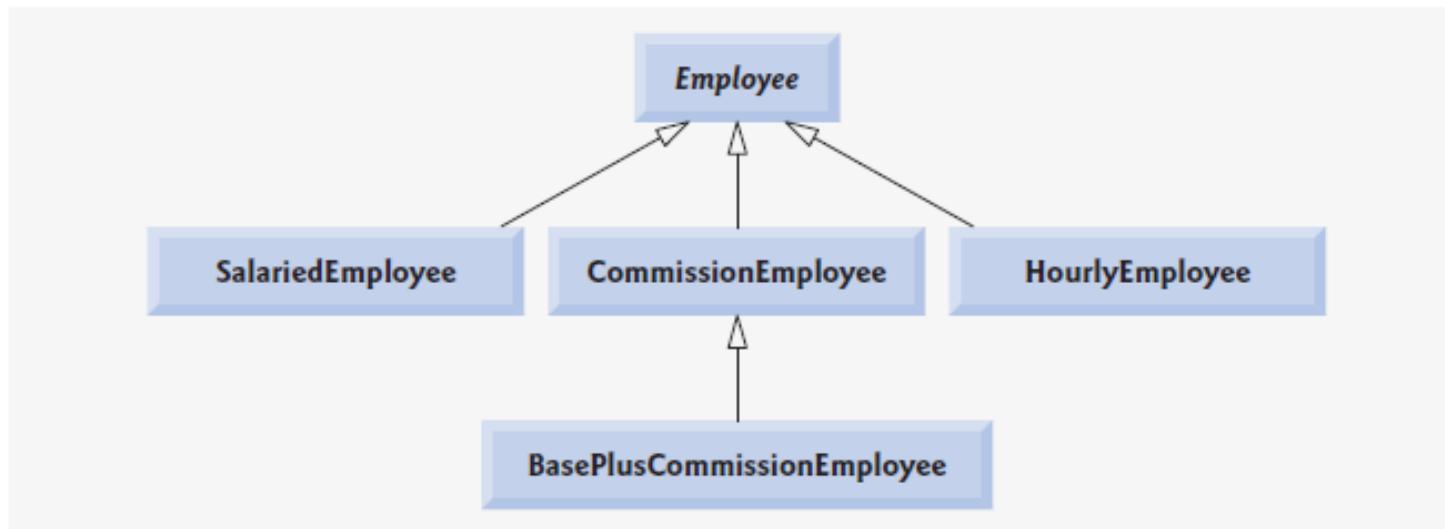
UFOP

- Especialmente para este período, deseja-se aplicar um bônus de 10% para os funcionários assalariados comissionados;
- É um requisito que esta aplicação processe os funcionários **polimorficamente**
  - Ou seja, sem que saibamos a classe de cada um.
- Vamos criar uma classe abstrata **Empregado**, que possua um método abstrato **salario()**
  - Calcula o salário de acordo com a implementação de cada subclasse;
  - Cada tipo de empregado será representado por uma subclasse;
  - A classe **EmpregadoComissionado** será superclasse direta da classe **EmpregadoComissionadoAssalariado**.



# Exemplo

UFOP





# Empregado.java

```
public abstract class Empregado
{
    private String nome;
    private String sobrenome;
    private String documento;

    public Empregado(String n, String s, String d)
    {
        setEmpregado(n, s, d);
    }

    public void setEmpregado(String n, String s, String d)
    {
        nome = n;
        sobrenome = s;
        documento = d;
    }
}
```



# Empregado.java

UFOP

```
//retorna a representacao em String, sobrescrito da classe Object
public String toString()
{
    return String.format("Nome: %s\nSobrenome: %s\nDocumento: %s",
                         nome, sobrenome, documento);
}
//metodo abstrato
//nao possui implementacao
//deve ser sobreescrito pelas subclasses
public abstract double salario();
}
```



# EmpregadoAssalariado.java

UFOP

```
public class EmpregadoAssalariado extends Empregado
{
    private double salarioSemanal;

    public EmpregadoAssalariado (String n, String s, String d, double ss)
    {
        super(n, s, d);
        setSalarioSemanal(ss);
    }

    public void setSalarioSemanal(double ss)
    {
        salarioSemanal = ss;
    }
}
```



# EmpregadoAssalariado.java

UFOP

```
//implementa o metodo abstrato da superclasse
public double salario()
{
    return salarioSemanal;
}
//sobrescreve o metodo da classe Object e da superclasse
public String toString()
{
    return String.format("%s\nSalario Semanal:%s", super.toString(),
        salarioSemanal);
}

}
```



# EmpregadoHorista.java

UFOP

```
public class EmpregadoHorista extends Empregado
{
    double valor;
    double horas;

    public EmpregadoHorista(String n, String s, String d, double v, double h)
    {
        super(n, s, d);
        setEmpregadoHorista(v, h);
    }

    public void setEmpregadoHorista(double v, double h)
    {
        valor = v;
        horas = h;
    }
}
```



# EmpregadoHorista.java

UFOP

```
//implementa o metodo abstrato da superclasse
public double salario()
{
    if(horas <=40)
        return valor * horas;
    else
        return 40*valor+(horas-40)*valor*1.5;
}
//sobrescreve o metodo da classe Object e da superclasse
public String toString()
{
    return String.format("%s\nHoras Trabalhadas:%.2f\nValor por hora:
                           %.2f", super.toString(), horas, valor);
}
```



# EmpregadoHorista.java

UFOP

```
public class EmpregadoComissionado extends Empregado
{
    double vendas;
    double comissao;

    public EmpregadoComissionado(String n, String s, String d, double v,
                                  double c)
    {
        super(n, s, d);
        setEmpregadoComissionado(v, c);
    }

    public void setEmpregadoComissionado(double v, double c)
    {
        vendas = v;
        comissao = c;
    }
}
```



# EmpregadoHorista.java

UFOP

```
//implementa o metodo abstrato da superclasse
public double salario()
{
    return vendas * comissao;
}
//sobrescreve o metodo da classe Object e da superclasse
public String toString()
{
    return String.format("%s\nTotal em vendas: %.2f\nComissao: %.2f",
        super.toString(), vendas, comissao);
}
```



# EmpregadoComissionadoAssalariado.java

UFOP

```
//herda da classe EmpregadoComissionado
public class EmpregadoComissionadoAssalariado extends
EmpregadoComissionado
{
    double salarioMensal;

    public EmpregadoComissionadoAssalariado(String n, String s, String d,
                                              double v, double c, double sm)
    {
        super(n, s, d, v, c);
        setSalarioMensal(sm);
    }

    public void setSalarioMensal(double sm)
    {
        salarioMensal = sm;
    }
}
```



# EmpregadoComissionadoAssalariado.java

UFOP

```
public double getSalarioMensal()
{
    return salarioMensal;
}
//implementa o metodo abstrato da superclasse
public double salario()
{
    return getSalarioMensal() + super.salario();
}
//sobrescreve o metodo da classe Object e da superclasse
public String toString()
{
    return String.format("%s\nSalario Mensal: %.2f", super.toString(),
        getSalarioMensal());
}
```



# TestePagamento.java

```
public class TestePagamento
{
    public static void main(String args[])
    {
        EmpregadoAssalariado ea = new EmpregadoAssalariado("John",
                                                               "Smith", "1111111-1", 800.00);
        EmpregadoHorista eh = new EmpregadoHorista("Karen", "Price",
                                                    "2222222-2", 16.75, 40);
        EmpregadoComissionado ec = new EmpregadoComissionado("Sue",
                                                               "Jones", "3333333-3", 10000, 0.06);
        EmpregadoComissionadoAssalariado eca = new
        EmpregadoComissionadoAssalariado("Bob", "Lewis", "44444444-4",
                                         5000, 0.04, 300);

        System.out.println("Processando individualmente\n");

        System.out.printf("%.2f\n", ea.salario());
        System.out.printf("%.2f\n", eh.salario());
        System.out.printf("%.2f\n", ec.salario());
        System.out.printf("%.2f\n", eca.salario());
    }
}
```



# TestePagamento.java

UFOP

```
Empregado vetor[] = new Empregado[4];  
  
vetor[0] = ea;  
vetor[1] = eh;  
vetor[2] = ec;  
vetor[3] = eca;  
  
System.out.println("\nProcessando polimorficamente\n");  
  
for(Empregado atual: vetor)  
{  
    //invoca o toString de acordo com o objeto  
    System.out.println(atual);  
    System.out.println();  
  
    //determina se o objeto é um EmpregadoComissionadoAssalariado
```



# TestePagamento.java

UFOP

```
if(atual instanceof EmpregadoComissionadoAssalariado)
{
    //realiza o dowcast
    EmpregadoComissionadoAssalariado temp =
        (EmpregadoComissionadoAssalariado) atual;
    //aplica o bônus de 10%
    temp.setSalarioMensal(temp.getSalarioMensal()*1.1);
    //imprime o novo salario
    System.out.println(temp);
}
//calcula o salario de acordo com o objeto
System.out.printf("Salario: %.2f\n", atual.salario());
System.out.println();
}

for(int j = 0; j<vetor.length; j++)
{
    System.out.printf("O empregado %d é %s\n", j,
                      vetor[j].getClass().getName());
}

}
```



# Saída

UFOP

Processando individualmente

800.00  
670.00  
600.00  
60000.00

Processando polimorficamente

Nome: John  
Sobrenome: Smith  
Documento: 1111111-1  
Salario Semanal: 800.0

Salario: 800.00  
Nome: Karen  
Sobrenome: Pric  
Documento: 2222222-2  
Horas Trabalhadas: 40,00  
Valor por hora: 16,75

Salario: 670.00  
Nome: Sue  
Sobrenome: Jones  
Documento: 3333333-3  
Total em vendas: 10000,00  
Comissao: 0,06

Salario: 600.00

Nome: Bob  
Sobrenome: Lewis  
Documento: 44444444-4  
Total em vendas: 5000,00  
Comissao: 0,04  
Salario Mensal: 300,00

Nome: Bob  
Sobrenome: Lewis  
Documento: 44444444-4  
Total em vendas: 5000,00  
Comissao: 0,04  
Salario Mensal: 30,00  
Salario: 6000.00

O empregado 0 é EmpregadoAssalariado  
O empregado 1 é EmpregadoHorista  
O empregado 2 é EmpregadoComissionado  
O empregado 3 é EmpregadoComissionadoAssalariado



# Exemplo

UFOP

	earnings	toString
Employee	abstract	<i>firstName lastName social security number: SSN</i>
Salaried-Employee	weeklySalary	<i>salaried employee: firstName lastName social security number: SSN weekly salary: weeklySalary</i>
Hourly-Employee	<pre>if hours &lt;= 40     wage * hours else if hours &gt; 40     40 * wage +     ( hours - 40 ) *     wage * 1.5</pre>	<i>hourly employee: firstName lastName social security number: SSN hourly wage: wage; hours worked: hours</i>
Commission-Employee	commissionRate * grossSales	<i>commission employee: firstName lastName social security number: SSN gross sales: grossSales; commission rate: commissionRate</i>
BasePlus-Commission-Employee	<pre>( commissionRate * grossSales ) + baseSalary</pre>	<i>base salaried commission employee: firstName lastName social security number: SSN gross sales: grossSales; commission rate: commissionRate; base salary: baseSalary</i>

# Resolução Dinâmica



# Resolução Dinâmica

UFOP

- Todas as chamadas aos métodos *toString()* e *salario()* são adequadas ao objeto que as chama
  - Resolvido em tempo de execução, baseado na classe do objeto;
  - **Resolução Dinâmica (*Dynamic Binding*)**.

# Operador *instanceof*



# Operador *instanceof*

UFOP

- O operador `instanceof` é utilizado para determinar se um objeto é de uma determinada classe

`currentEmployee instanceof BasePlusCommissionEmployee`

- O valor retornado é *true* caso o objeto pertença à classe ou caso herde direta ou indiretamente da classe;
- Retorna *false* caso contrário.

# *Downcast*



# Downcast

UFOP

- Os métodos `getSalario()` e `setSalario()` são definidos apenas na subclasse

*EmpregadoComissionadoAssalariado*

- Não é possível invocá-lo em outras classes;
- Uma vez determinado que se trata de um objeto da classe *EmpregadoComissionadoAssalariado*, é necessário realizar a conversão para o tipo adequado
  - De superclasse para subclasse;
  - **Downcast.**



# Downcast

UFOP

- Atribuir uma variável de uma superclasse a uma variável de uma subclasse sem realizar *cast* explícito resulta em erro de compilação;
- Antes de realizar um *downcast*, sempre realize um teste antes com o operador *instanceof*;
- Durante a realização de um *downcast*, se o objeto não possuir um relacionamento tem um com o objeto do *cast* será lançada uma exceção
  - *ClassCastException*.

# Método *getClass*



# Método *getClass*

UFOP

- Todos os objetos em Java sabem qual é a sua própria classe e podem acessar esta informação através do método *getClass()*
  - Todas as classes o herdam da classe *Object*;
  - Retorna um objeto;
  - Neste objeto, invocamos o método *getName()*, que retorna o nome da classe que o objeto representa.

# Métodos e Classes *final*



# Métodos e Classes *final*

UFOP

- Como vimos anteriormente, variáveis podem ser declaradas como *final* para indicar que não podem ser modificadas depois de inicializadas
  - Valor constante.
- Também é possível declarar classes e métodos com o modificador *final*;
- Métodos declarados como *final* não podem ser sobrescritos em subclasses
  - Métodos privados e *static* são declarados *final* implicitamente;
  - Desta forma, a implementação não mudará.



# Métodos e Classes *final*

UFOP

- Chamadas a métodos declarados como *final* são resolvidas em tempo de compilação
  - Resolução Estática (*Static Binding*).
- Uma vez que o compilador sabe que a implementação não mudará, ele pode substituir a chamada do método pelo código do próprio método
  - *Inlining*;
  - Reduz o *overhead* da chamada de métodos.



# Métodos e Classes *final*

- Uma classe que é declarada como *final* não pode ser uma superclasse
  - Ou seja, uma classe *final* não pode ser estendida;
  - Todos os métodos de uma classe *final* são implicitamente declarados como *final*.
- Por exemplo, a classe *String* é declarada como *final*
  - Imutável;
  - Os programas que utilizam *Strings* confiam apenas na implementação da API.
- Tornar uma classe *final* também previne que programadores burlem restrições de segurança



# Métodos e Classes *final*

UFOP

```
class ChessAlgorithm {  
    enum ChessPlayer { WHITE, BLACK }  
    ...  
    final ChessPlayer getFirstPlayer() {  
        return ChessPlayer.WHITE;  
    }  
    ...  
}
```

# Criando e Utilizando Interfaces



# Criando e Utilizando Interfaces

UFOP

- Vamos estender nosso exemplo anterior:
  - Suponha que a mesma empresa deseja realizar pagamentos diversos utilizando uma mesma aplicação
    - Pagamento de empregados e também de notas fiscais de fornecedores;
    - Cada pagamento depende do tipo do empregado ou do conteúdo da nota fiscal.
  - Apesar de aplicado a elementos distintos (empregados e notas fiscais), ambas as operações envolvem o pagamento de algo
    - Seria possível processar elementos distintos polimorficamente?



# Criando e Utilizando Interfaces

UFOP

- Continuação
  - Seria possível implementar um conjunto de métodos comuns em classes não relacionadas?
- As **interfaces** Java oferecem exatamente esta capacidade
  - Padronizam as formas em que *elementos* como sistemas e pessoas podem interagir entre si;
  - Por exemplo, os controles de um rádio definem a interface entre o usuário e os componentes internos
    - Permite que os usuários realizem um conjunto restrito de operações;
    - Diferentes rádios implementam os controles de forma diferente.



# Criando e Utilizando Interfaces

UFOP

- A interface especifica **o que** deve ser feito
  - Quais operações.
  - Porém, não especifica **como** deve ser feito.
- Objetos de *software* também se comunicam via interfaces
  - Uma interface descreve um conjunto de métodos que podem ser invocados a partir de um objeto
    - Dizem ao objeto que uma tarefa deve ser realizada, ou que deve ser retornada uma porção de informação.



# Criando e Utilizando Interfaces

UFOP

- A declaração de uma interface começa com a palavra **interface** e contém apenas constantes e métodos abstratos
  - Todos os membros devem ser públicos;
  - Não devem especificar nenhum detalhe de implementação
    - Como variáveis ou métodos concretos.



# Criando e Utilizando Interfaces

UFOP

- Logo, todos os métodos são **implicitamente** *public abstract*;
  - Todos os atributos são **implicitamente** *public, static* e *final*.
- De acordo com a especificação da linguagem Java
  - Não se declara métodos de uma interface com os modificadores *public* e *abstract*;
  - Constantes não são declaradas com os modificadores *public, static* e *final*;
  - São redundantes em interfaces.



# Criando e Utilizando Interfaces

UFOP

- Para utilizar uma interface, uma classe concreta deve especificar que a implementa
  - Definido pela palavra **implements**;
  - Deve declarar cada método da interface, com a mesma assinatura
    - Uma classe que não implementa todos os métodos de uma interface é **abstrata**, e assim deve ser declarada.



# Criando e Utilizando Interfaces

UFOP

- Interfaces são utilizadas quando classes não relacionadas precisam compartilhar métodos e constantes
  - Objetos destas classes podem ser processados polimorficamente
    - Objetos de classes que implementam uma mesma interface respondem aos mesmos métodos.
- Outra aplicação de interfaces é na substituição de classes abstratas em que não há implementação padrão a ser herdada
  - Métodos ou atributos
  - Interfaces normalmente são públicas, logo, podem ser declaradas em arquivos próprios, com extensão .java

# Exemplo



# Exemplo

UFOP

- Vejamos o exemplo de uma interface chamada *Pagavel*
  - Descreve a funcionalidade de qualquer objeto capaz de ser pago e, portanto, fornece um método para determinar a quantidade a ser paga.



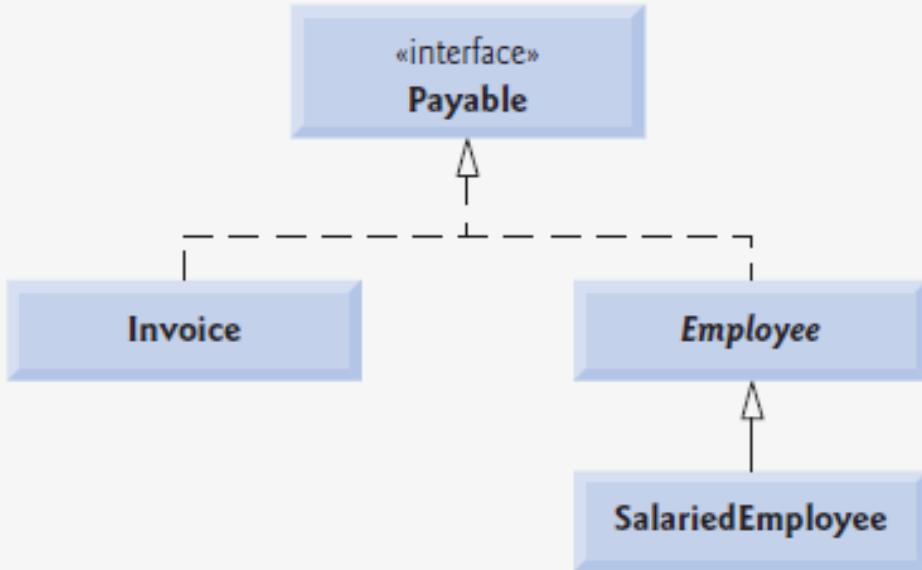
# Exemplo

- A interface **Pagavel** determina a quantia a ser paga a um empregado ou uma nota fiscal
  - Método **double getValorAPagar();**
  - Uma versão genérica do método **salario()**
    - Pode ser aplicada a diferentes classes.
- Depois de declararmos esta interface, definiremos a classe **NotaFiscal**, que a implementará;
- Também modificaremos a classe **Empregado** para implementar a interface
  - A subclasse **EmpregadoAssalariado** também será modificada.



# Exemplo

UFOP



- O relacionamento entre uma classe e uma interface em UML é chamado de realização
  - Uma classe *realiza* os métodos de uma interface;
  - Representado por uma linha pontilhada com seta vazada.



# Pagavel.java

UFOP

```
public interface Pagavel
{
    //calcula o pagamento, nao possui implementacao
    double getValorAPagar();
}
```



# NotaFiscal.java

```
public class NotaFiscal implements Pagavel
{
    private String numeroPeca;
    private String descricaoPeca;
    private int quantidade;
    private double precoPorItem;

    public NotaFiscal( String peca, String descricao, int numero, double preco )
    {
        numeroPeca = peca;
        descricaoPeca = descricao;
        setQuantidade( numero );
        setPrecoPorItem( preco );
    }

    // setter
    public void setNumeroPeca( String peca )
    {
        numeroPeca = peca;
    }
}
```



# NotaFiscal.java

UFOP

```
//getter
public String getNumeroPeca()
{
    return numeroPeca;
}

// setter
public void setDescricaoPeca( String descricao )
{
    descricaoPeca = descricao;
}

// getter
public String getDescricaoPeca()
{
    return descricaoPeca;
}
```



# NotaFiscal.java

UFOP

```
// setter
public void setQuantidade( int count )
{
    //a quantidade nao pode ser negativa
    quantidade = ( count < 0 ) ? 0 : count;
}

// getter
public int getQuantidade()
{
    return quantidade;
}

//setter
public void setPrecoPorItem( double preco )
{
    precoPorItem = ( preco < 0.0 ) ? 0.0 : preco;
}
```



# NotaFiscal.java

UFOP

```
//getter
public double getPrecoPorItem()
{
    return precoPorItem;
}

// retorna a representacao de um objeto NotaFiscal em formato String
public String toString()
{
    return String.format( "%s: \n%s: %s (%s) \n%s: %d \n%s: $%,.2f",
        "Nota", "peca numero", getNumeroPeca(),
        getDescricaoPeca(), "quantidade",
        getQuantidade(), "preco por item",
        getPrecoPorItem() );
}

//metodo exigido para cumprir o contrato com a interface Pagavel
public double getValorAPagar()
{
    //calcula o custo total
    return getQuantidade() * getPrecoPorItem();
}
```



# Empregado.java

UFOP

```
//Classe abstrata Empregado nao implementa o metodo getValorASerPago() da interface
Pagavel
public abstract class Empregado implements Pagavel
{
    private String nome;
    private String sobrenome;
    private String documento;

    public Empregado( String n, String s, String d )
    {
        nome = n;
        sobrenome = s;
        documento = d;
    }

    //setter
    public void setNome( String n )
    {
        nome = n;
    }
}
```



# Empregado.java

UFOP

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

//setter
public void setSobrenome( String s )
{
    sobrenome = s;
}

//getter
public String getSobrenome()
{
    return sobrenome;
}
```



# Empregado.java

UFOP

```
//setter
public void setDocumento( String d )
{
    documento = d;
}

//getter
public String getDocumento()
{
    return documento;
}

// retorna a representacao de um objeto Empregado em formato String
public String toString()
{
    return String.format( "%s %s\nDocumento: %s", getNome(),
                           getSobrenome(), getDocumento() );
}
```



# EmpregadoAssalariado.java

```
public class EmpregadoAssalariado extends Empregado
{
    private double salarioMensal;

    public EmpregadoAssalariado( String n, String s, String d, double salario )
    {
        super( n, s, d );
        setSalarioMensal( salario );
    }

    //setter
    public void setSalarioMensal( double salario )
    {
        salarioMensal = salario < 0.0 ? 0.0 : salario;
    }

    //getter
    public double getSalarioMensal()
    {
        return salarioMensal;
    }
}
```



# EmpregadoAssalariado.java

UFOP

```
//calcula o salario, implementa o metodo que e  
//abstrato na superclasse Empregado  
public double getValorAPagar()  
{  
    return getSalarioMensal();  
}  
  
// retorna a representacao de um objeto EmpregadoAssalariado em formato  
//String  
public String toString()  
{  
    return String.format( "Empregado Assalariado: %s\n%s: $%,.2f",  
                           super.toString(), "Salario", getSalarioMensal() );  
}
```



# TesteInterfacePagavel.java

UFOP

```
public class TesteInterfacePagavel
{
    public static void main( String args[] )
    {
        //Crie um vetor de 4 elementos da interface Pagavel
        Pagavel pagavel[] = new Pagavel[ 4 ];

        //Preenche o vetor com objetos de classes que implementam a interface
        //Pagavel
        pagavel[ 0 ] = new NotaFiscal( "01234", "banco", 2, 375.00 );
        pagavel[ 1 ] = new NotaFiscal( "56789", "pneu", 4, 79.95 );
        pagavel[ 2 ] =new EmpregadoAssalariado( "John", "Smith", "111-11-
            1111", 800.00 );
        pagavel[ 3 ] =new EmpregadoAssalariado( "Lisa", "Barnes", "888-88-
            8888", 1200.00 );
    }
}
```



# TesteInterfacePagavel.java

UFOP

```
System.out.println("Notas Fiscais e Empregados processados polimorficamente:\n");

//Processa os elementos do vetor automaticamente
for ( Pagavel atual : pagavel )
{
    System.out.printf( "%s \n%s: $%,.2f\n\n", atual.toString(),
                      "Pagamento Devido", atual.getValorAPagar());
}
}
```



# Saída

UFOP

Notas Fiscais e Empregados processados polimorficamente:

Nota:

peça numero: 01234 (banco)

quantidade: 2

preço por item: \$375,00

Pagamento Devido: \$750.00

Nota:

peça numero: 56789 (pneu)

quantidade: 4

preço por item: \$79,95

Pagamento Devido: \$319.80

Empregado Assalariado: John Smith

Documento: 111-11-1111

Salario: \$800,00

Pagamento Devido: \$800.00

Empregado Assalariado: Lisa Barnes

Documento: 888-88-8888

Salario: \$1.200,00

Pagamento Devido: \$1,200.00



# Exemplo

- O relacionamento **é um** existe entre superclasses e subclasses e entre interfaces e classes que as implementam
  - Quando um método recebe por parâmetro uma variável de superclasse ou de interface, o método processa o objeto recebido polimorficamente.
- Através de uma referência a uma superclasse ou interface, podemos polimorficamente invocar qualquer método definido na superclasse ou interface
  - Através de uma referência a uma interface, podemos invocar polimorficamente qualquer método definido na interface.



# Exemplo

UFOP

- Todos os métodos da classe *Object* podem ser invocados a partir de uma referência a uma interface.

# Declarando Constantes em Interfaces

# Declarando Constantes em Interfaces



UFOP

- Como dito anteriormente, uma interface pode ser utilizada para declarar constantes
  - Implicitamente, são *public*, *static* e *final*, o que não precisa ser declarado no código.
- Um uso popular é declarar um conjunto de constantes que possa ser utilizado em várias declarações de classes
  - Uma classe pode utilizar as constantes importando a interface;
  - Refere-se ao nome da interface e ao nome da constante, separados por `;`
  - Se for realizado *static import*, o nome da interface pode ser omitido.

# Declarando Constantes em Interfaces



UFOP

```
public interface Constants
{
    int ONE = 1;
    int TWO = 2;
    int THREE = 3;
}
```

# Herança Múltipla e Interfaces



# Herança Múltipla e Interfaces

UFOP

- Java não fornece suporte à herança múltipla
  - No entanto, uma classe pode implementar mais do que uma interface;
  - Desta forma, é possível determinar que uma classe “absorva” o comportamento de diferentes interfaces;
  - Ainda é necessário implementar as interfaces.
- Para nosso exemplo, suponha que possuímos interfaces **Pesado** e **Colorido**, uma classe abstrata **Animal** e queremos implementar uma classe **Porco**
  - Os objetos da classe Porco devem saber seu peso e sua cor, além de serem animais;
  - Desta forma, podemos herdar diretamente da classe animal e implementar as duas interfaces.



# Herança Múltipla e Interfaces

UFOP

```
abstract class Animal
{
    public abstract void fazerBarulho();
}

interface Pesado
{
    double obterPeso();
}

interface Colorido
{
    String obterCor();
}
```



# Herança Múltipla e Interfaces

UFOP

```
class Porco extends Animal implements Pesado, Colorido
{
    public void fazerBarulho()
    {
        System.out.println("Óinc!");
    }

    //Implementação da interface Pesado
    public double obterPeso()
    {
        return 50.00;
    }

    //Implementação da interface Colorido
    public String obterCor()
    {
        return "Preto";
    }
}
```



# Herança Múltipla e Interfaces

UFOP

```
//Uma propriedade só do porco
public boolean enlameado()
{
    return true;
}

public String toString()
{
    return String.format("Cor: %s, Peso: %f, Enlameado? %s.",
        obterCor(), obterPeso(), enlameado());
}
```



# Herança Múltipla e Interfaces

UFOP

```
public class DriverInterfacesMultiplas
{
    public static void main (String args[])
    {
        Porco missPiggy = new Porco();
        System.out.println(missPiggy);
        missPiggy.fazerBarulho();
    }
}
```

# Interfaces Comuns da API Java



# Interfaces Comuns da API Java

UFOP

- As interfaces da API java permite que usemos nossas próprias classes dentro de *frameworks* fornecidos pelo Java
  - Tais como comparar objetos de suas próprias classes e criar múltiplas linhas de execução.
- Algumas das mais populares são:
  - *Comparable*;
  - *Serializable*;
  - *Runnable*;
  - *GUI event-listener interfaces*;
  - *Swing Constants*.



# Interfaces Comuns da API Java

UFOP

## ■ *Comparable*

- Permite que objetos de classes que a implementam sejam comparados;
- Contém apenas um método:
  - *compareTo();*
- Compara o objeto que o invocou com um objeto passado como argumento;
- Deve retornar:
  - Um número negativo, caso o primeiro objeto seja menor;
  - Zero, caso os objetos sejam iguais;
  - Um número positivo, caso o primeiro objeto seja maior.



# Interfaces Comuns da API Java

UFOP

## ■ *Serializable*

- Utilizada para identificar classes cujos objetos podem ser escritos (serializados) ou lidos (desserializados) de algum tipo de armazenamento (arquivo, banco de dados), ou transmitidos através da rede.



# Interfaces Comuns da API Java

UFOP

## ■ *Runnable*

- Implementada por qualquer classe cujos objetos devem estar aptos a executarem em paralelo utilizando uma técnica chamada de *multithreading*;
- Contém um método:
  - `run();`
- Descreve o comportamento do objeto quando executado.



# Interfaces Comuns da API Java

UFOP

## ■ *GUI event-listener interfaces*

- Toda interação com uma interface gráfica é chamada de **evento**
  - O código utilizado para responder é chamado de **manipulador de evento** (*event handler*).
- Os manipuladores de eventos são declarados em classes que implementam uma interface *event-listener* adequada
  - Cada uma delas especifica um ou mais métodos que devem ser implementados para responder às interações do usuário.



# Interfaces Comuns da API Java

UFOP

## ■ *Swing Constants*

- Contém um conjunto de constantes utilizado em programação de interfaces gráficas
  - Posicionamento de elementos GUI na tela.



UFOP



# Perguntas?



# Na próxima aula

UFOP

- Introdução ao Java Reflections API



UFOP



**FIM**