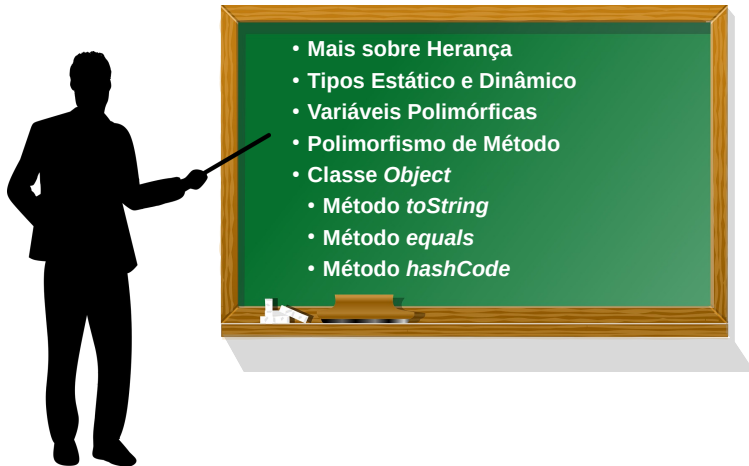


## Herança e Polimorfismo - Parte 2

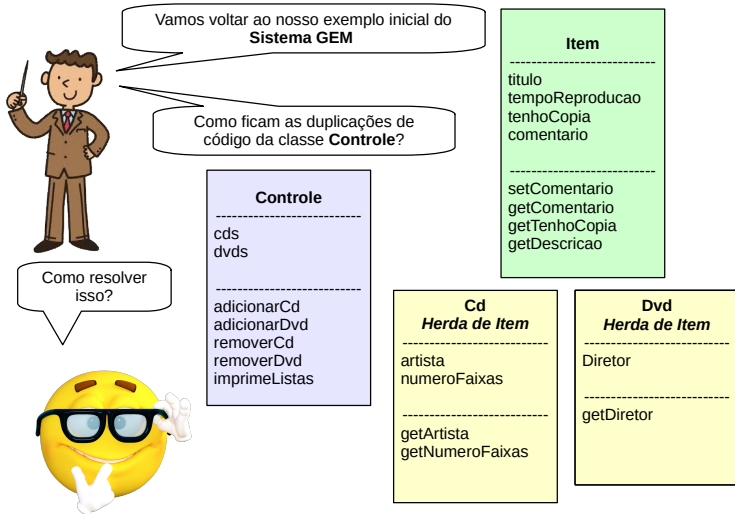
Luiz Henrique de Campos Merschmann  
Departamento de Computação Aplicada  
Universidade Federal de Lavras

luiz.hcm@ufla.br

# Na Aula de Hoje



# Mais sobre Herança



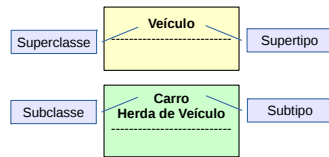
# Mais sobre Herança

- ▶ Quando queremos atribuir um objeto a uma variável, o tipo do objeto deve corresponder ao tipo da variável. Exemplo:  
`Carro meuCarro = new Carro( );`

## A Hierarquia de Herança define uma Hierarquia de Tipos

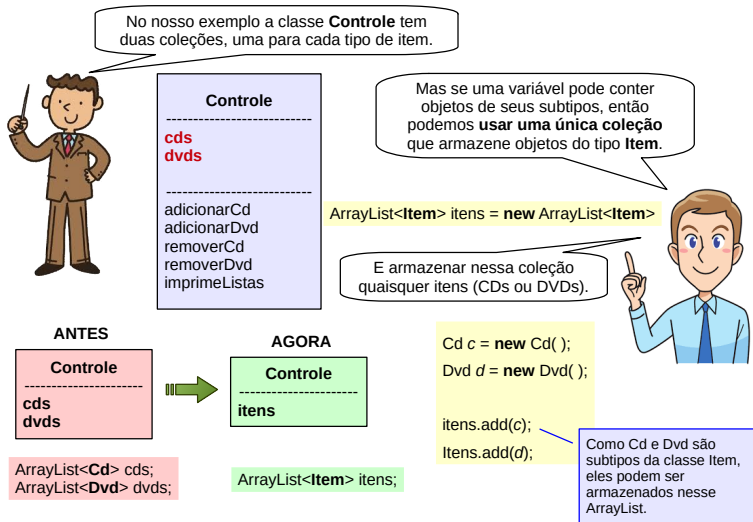
Mas agora que conhecemos herança, a regra é:

- ▶ Uma variável pode conter objetos de seu tipo declarado ou de qualquer um dos seus subtipos.
- ▶ Ou seja, podemos ter: `Veiculo meuVeiculo = new Carro( );`
- ▶ Isso é válido porque um objeto da classe **Carro** é um objeto da classe **Veiculo**.



E qual diferença isso faz?

# Mais sobre Herança



# Mais sobre Herança



Vamos analisar agora o método da classe **Controle** utilizado para imprimir a descrição de seus objetos.

## ANTES

```
public void imprimeListas(){  
    for (Cd cd: cds){  
        System.out.println(cd.getDescricao());  
    }  
    for (Dvd dvd: dvds){  
        System.out.println(dvd.getDescricao());  
    }  
}
```

## AGORA

```
public void imprimeListas(){  
    for (Item item: itens){  
        System.out.println(item.getDescricao());  
    }  
}
```

Mas isso funciona?

Qual método *getDescricao* será chamado?



Para entendermos o que está ocorrendo precisamos conhecer os conceitos de **tipo estático** e **tipo dinâmico**.

# Tipo Estático e Tipo Dinâmico

Chamamos de **tipo estático** de uma variável o **tipo utilizado no código** ao declarar a variável.

- ▶ Por exemplo, a partir da declaração  
**Item** *i*;  
podemos afirmar que o tipo estático da variável *i* é **Item**.

Chamamos de **tipo dinâmico** de uma variável o **tipo do objeto** que **atualmente está vinculado** à variável.

- ▶ Por exemplo, a partir da declaração  
*i* = **new** Cd( );  
podemos afirmar que o tipo dinâmico da variável *i* **agora** é **Cd**.

# Variáveis Polimórficas



Quais são os tipos estático e dinâmico de cada uma das variáveis a seguir?

```
Veículo v1 = new Carro();  
Veículo v2 = new Bicicleta();  
Caminhao c1 = new Caminhao();  
v2 = c1;  
v1 = v2;
```

Como o **tipo estático** é aquele declarado no código, então:

- **v1** e **v2** são do tipo *Veículo*.
- **c1** é do tipo *Caminhao*.

Como o **tipo dinâmico** é o do objeto que está vinculado num certo momento:

- **c1** é do tipo *Caminhao*.
- **v1**, da 1ª a 4ª linha, é do tipo *Carro* e, na 5ª linha, passa a ser *Caminhao*.
- **v2**, nas 2ª e 3ª linhas é do tipo *Bicicleta* e, a partir da 4ª linha passa a ser *Caminhao*.

Essas variáveis que podem ser de tipos dinâmicos diferentes em momentos distintos são denominadas **variáveis polimórficas**. De fato, elas podem assumir diferentes “formas” (quer dizer, referenciar objetos diferentes).



# Pesquisa Dinâmica de Método

Durante a execução do programa, a JVM verifica o tipo dinâmico da variável.

- ▶ Considere o seguinte exemplo:

```
Veiculo v1 = new Carro( );  
v1.deslocar( );  
v1 = new Bicicleta( );  
v1.deslocar( );
```

- ▶ A primeira chamada ao método *deslocar* irá executar o método da classe **Carro**. Já a segunda executará o método da classe **Bicicleta**.
- ▶ Isso ocorre porque a JVM procura o método primeiro na classe associada ao tipo dinâmico.
  - ▶ Se não encontrar, a busca sobe na hierarquia de herança.
- ▶ Isso é chamado de **pesquisa dinâmica de método**.

# Polimorfismo de Método

É por isso que aquela solução apresentada anteriormente (código abaixo) funciona!



```
public void imprimeListas(){  
    for (Item item: itens){  
        System.out.println(item.getDescricao());  
    }  
}
```

Lembre-se que a coleção *itens* possui objetos do tipo Cd e Dvd.

Durante a **compilação** do código, o compilador verificará se a classe **Item** possui o método *getDescricao*, pois ele utiliza o tipo estático da variável *item*.

Por outro lado, durante a **execução** do código, a JVM procurará o método *getDescricao* na classe do objeto que estiver sendo referenciado naquele momento, pois ela utiliza o tipo dinâmico da variável *item*.

Esse processo é conhecido como **polimorfismo de método**.

# A Importância do Polimorfismo



Além de termos que escrever menos código, qual seria outra vantagem da solução à direita?

```
public void imprimeListas(){  
    for (Cd cd: cds){  
        System.out.println(cd.getDescricao());  
    }  
    for (Dvd dvd: dvds){  
        System.out.println(dvd.getDescricao());  
    }  
}
```

```
public void imprimeListas(){  
    for (Item item: itens){  
        System.out.println(item.getDescricao());  
    }  
}
```

Suponha que desejamos acrescentar o item Videogame no nosso sistema GEM.

Considerando a solução à esquerda, teríamos que **acrescentar** no método `imprimeListas` o seguinte trecho de código:

```
for (VideoGame vg: vgs){  
    System.out.println(vg.getDescricao());  
}
```

E na solução à direita?

Não preciso mudar **nada!**

Ou seja, tornei a extensão do sistema muito mais fácil.

# Princípio da Substituição

Suponha que alguém lhe pediu emprestado uma caneta.

Você pode atender a essa solicitação entregando a essa pessoa uma caneta tinteiro ou uma caneta esferográfica.

- ▶ Observe que tanto a caneta tinteiro quanto a caneta esferográfica são subclasses de caneta.
- ▶ Portanto, fornecer qualquer uma delas quando um objeto da classe caneta é esperado é normal!




# Cuidados com o Princípio da Substituição

Vamos considerar essa hierarquia de classes/tipos para analisar algumas situações.

```
graph TD; Veiculo[Veículo] --> Carro[Carro  
Herda de Veículo]; Veiculo --> Bicicleta[Bicicleta  
Herda de Veículo];
```

O código abaixo irá compilar?


```
Carro c1 = new Veiculo( );
```



Apesar de todo carro ser um veículo, o oposto não necessariamente é verdade. Portanto, o Java não permitirá isso emitindo erro durante a compilação.

O código abaixo irá compilar?


```
Carro c2 = new Bicicleta( );
```



Apesar de carro e bicicleta serem veículos, uma bicicleta não é um carro. Portanto, o Java não permitirá isso emitindo erro durante a compilação.

O código abaixo irá compilar?

```
Veiculo v;  
Carro c = new Carro( );  
v = c;  
c = (Carro) v;
```



Estamos usando conversão de tipos (casting), para forçar que o compilador aceite que **v** é do tipo Carro. Em tempo de execução será verificado se **v** realmente é um carro.

# Exercício



O que aconteceria se tentássemos compilar e executar o código a seguir?

Analise os comandos das linhas 5, 6 e 7.

**Veículo**

**Carro**  
Herda de Veículo

**Bicicleta**  
Herda de Veículo

Correto! Estamos usando uma variável polimórfica (um carro **é** um veículo).

Compila, mas dará erro durante a execução! Isso ocorre porque **v** está referenciando um carro e não uma bicicleta.

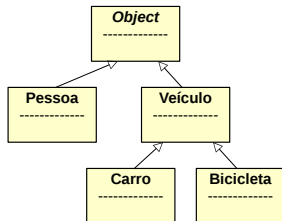
```
1 Veiculo v;  
2 Carro c;  
3 Bicicleta b;  
4 c = new Carro();  
5 v = c;  
6 b = (Bicicleta) c;  
7 b = (Bicicleta) v;
```

Erro de compilação!  
Um carro não pode ser convertido em uma bicicleta.

# A Classe *Object*

Em Java, todas as classes têm uma superclasse!

- ▶ Todas as classes que não tem uma declaração explícita de superclasse herdam de uma classe chamada *Object*.
  - ▶ Portanto, todas as classes em Java herdam **direta** ou **indiretamente** da classe *Object*.



Qual a utilidade disso?

1. Podemos declarar variáveis polimórficas do tipo *Object* para referenciar objeto de qualquer tipo.
  - ▶ `Object obj = new QualquerClasse( );`
2. A classe *Object* possui métodos que estão automaticamente disponíveis para todos os objetos existentes.

# A Classe *Object*

Dada a importância dos mesmos, discutiremos sobre os métodos **toString**, **equals** e **hashCode** definidos na classe **Object**.



All Methods	Instance Methods	Concrete Methods
Modifier and Type	Method and Description	
protected Object	<b>clone()</b>	Creates and returns a copy of this object.
boolean	<b>equals(Object obj)</b>	Indicates whether some other object is "equal to" this one.
protected void	<b>finalize()</b>	Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	<b>getClass()</b>	Returns the runtime class of this Object.
int	<b>hashCode()</b>	Returns a hash code value for the object.
void	<b>notify()</b>	Wakes up a single thread that is waiting on this object's monitor.
void	<b>notifyAll()</b>	Wakes up all threads that are waiting on this object's monitor.
String	<b>toString()</b>	Returns a string representation of the object.



# Método *toString*

- ▶ O propósito do método *toString* é retornar uma representação do estado do objeto como String.

O que acontece se você chamar o método *toString* para um objeto da classe **Cd** (sem que o método tenha sido sobrescrito)?

- ▶ O valor de retorno será algo como: **Cd@6acdd1** que é formado agregando-se o **nome da classe do objeto** + **@** + a **representação do código de hash** do objeto em hexadecimal.

Por que não foi retornada uma String com informações sobre o estado do objeto?

- ▶ Porque a classe *Object* não possui atributos, portanto ela não tem um estado.
  - ▶ Por isso é exibido o nome da classe do objeto seguido pela representação do código de hash do mesmo.

# Método *toString*

Mas se o método *toString* não foi sobrescrito na classe **Cd**, então por que o retorno do método não foi *Object@6acdd1*?

- ▶ Porque é exibido o tipo dinâmico da variável.

Desse modo, para tornar *toString* um método mais útil, precisamos sobrescrevê-lo em nossas classes.

- ▶ Ao sobrescrever o método podemos escolher aquilo que julgamos importante para descrever o estado do objeto.

# Exemplos de Uso do Método *toString*

Na classe **Cd**:

**@Override**

```
public String toString() {  
    return "Artista:" + artista + "Número de Faixas:" + numeroFaixas;  
}
```

Ou então:

**@Override**

```
public String toString() {  
    return super.toString() + "Artista:" + artista;  
}
```

## Dica



Ao se usar *System.out.println* e *System.out.print*, se o parâmetro para um desses métodos não for um objeto *String*, eles automaticamente invocam o método *toString* do objeto passado como parâmetro. Assim, não precisamos escrever a chamada ao método *toString* explicitamente. Podemos, por exemplo, escrever somente: *System.out.println(cd)*;

# Método *equals*

Já vimos que o método *equals* serve para indicar se dois objetos são iguais em termos de conteúdo (e não em termos de referência!)

- ▶ Ele é declarado na classe *Object* e tem como parâmetro um objeto do tipo *Object*.

```
public boolean equals(Object obj)
```

Indicates whether some other object is "equal to" this one.

O que ocorre se chamarmos o método *equals* para um objeto de uma classe que não o sobrescreveu?

- ▶ Exemplo: `if(cd1.equals(cd2))`
- ▶ Como o método não foi sobrescrito na classe **Cd**, será chamado o *equals* da classe *Object*.

# Método *equals*

Como a classe *Object* não possui atributos, na sua implementação do método *equals* não há como comparar estados de objetos.

- ▶ Portanto, o que está implementado é apenas um teste de igualdade de referência.

```
public boolean equals(Object obj){  
    return this == obj;  
}
```

- ▶ Então, no exemplo anterior, **cd1.equals(cd2)** será verdadeiro apenas se as variáveis **cd1** e **cd2** fizerem referência ao mesmo objeto.

# Método *equals*

Como se verifica a “igualdade de conteúdo” entre dois objetos?

- ▶ A “igualdade de conteúdo” entre dois objetos só pode ser definida comparando-se os atributos definidos na classe dos mesmos.
- ▶ Desse modo, para que o método *equals* possa cumprir o seu papel na comparação entre objetos, precisamos sobrescrevê-lo na classe de interesse.

# Método *equals*

Vamos implementar a sobrescrição do método *equals* na classe **Cd**, a qual possui os atributos *numeroFaixas* (int) e *artista* (String).

## @Override

```
public boolean equals(Object obj){  
    return this.numeroFaixas == obj.??? && ...;  
}
```

- ▶ Como conseguiremos pegar os atributos do **cd** que foi passado por parâmetro?
- ▶ Além disso, como o parâmetro é do tipo *Object*, se o método for chamado passando um objeto que não é da classe **Cd**, não dará erro de compilação.
  - ▶ Como tratar isso?

# Operador *instanceof*



Para resolver os problemas levantados anteriormente, utilizaremos o operador **instanceof**.

Ele é um operador booleano que indica se um objeto é de uma determinada classe.

```
if(umObjeto instanceof UmaClasse)
```

Portanto, vejamos como fica a sobrescrição do método *equals* na classe **Cd**:

```
public boolean equals(Object obj){  
    if(this == obj){  
        return true;  
    }  
    else if(!(obj instanceof Cd)){  
        return false;  
    }  
    else{  
        Cd outro = (Cd) obj;  
        return numeroFaixas == outro.numeroFaixas &&  
            artista.equals(outro.artista);  
    }  
}
```

Qual o motivo desta comparação? **Resp.: Eficiência, pois esse é um teste de igualdade de referência.**

Qual o motivo desta comparação? **Resp.: Para certificar que estamos comparando objetos da mesma classe.**

Por que podemos fazer esse *casting* aqui? Qual a utilidade? **Resp.: A conversão de tipo e a variável *outro* nos permitem acessar os atributos de Cd.**



# Método *equals* e *hashCode*

Para que serve o método *hashCode*?

- ▶ É utilizado por estruturas de dados como *HashMap* e *HashSet* para fornecer uma pesquisa eficiente dos objetos nessas coleções.
- ▶ Essencialmente, ele retorna um valor inteiro que representa um objeto.

Sempre que o método *equals* é sobrescrito, o método *hashCode* também deve ser sobrescrito.

Por que?

- ▶ O vínculo entre *equals* e *hashCode* está no fato de que **sempre que dois objetos são considerados iguais pelo método *equals*, eles devem retornar valores idênticos a partir do método *hashCode*<sup>1</sup>.**

---

<sup>1</sup>Não é essencial que objetos não-idênticos sempre retornem códigos de *hash* distintos.

# Método *equals* e *hashCode*

## Como calcular códigos de *hash*?

- ▶ Uma maneira de produzir esse valor inteiro seria calculá-lo usando os valores dos atributos que são comparados pelo método *equals* sobrescrito<sup>2</sup>.
- ▶ Exemplo:

**@Override**

```
public int hashCode( ){  
    int resultado = 13; //Um valor inicial arbitrário.  
    resultado = 37 * resultado + numeroFaixas + artista.hashCode( );  
    return resultado;  
}
```

---

<sup>2</sup>Foge do escopo desta disciplina o estudo de implementações de *hashCode*'s eficientes.

# Perguntas?

