



# Curso Java Básico

Uma introdução prática usando  
BlueJ



# Aperfeiçoando estruturas com o uso da herança

# Principais conceitos a serem abordados

- Herança
- Subtipagem
- Substituição
- Variáveis polimórficas
- Classes abstratas

# DoME - Database of Multimedia Entertainment

- Modelaremos um banco de dados de entretenimento multimídia:
  - Armazena detalhes sobre CDs e vídeos:
    - CD: título, artista, número de faixas, duração, comentário, recomendação de compra (“got-it”).
    - Vídeo: título, diretor, duração, comentário, got-it.
  - Permite (posteriormente) pesquisar informações ou imprimir listas.
  - Não trataremos de armazenamento, interface com usuário e outros aspectos de uma aplicação completa.

# Objetos do DoME

**: CD**

title	<input type="text"/>
artist	<input type="text"/>
#tracks	<input type="text"/>
playing time	<input type="text"/>
got it	<input type="text"/>
comment	<input type="text"/>

**: DVD**

title	<input type="text"/>
director	<input type="text"/>
playing time	<input type="text"/>
got it	<input type="text"/>
comment	<input type="text"/>

# Classes do DoME

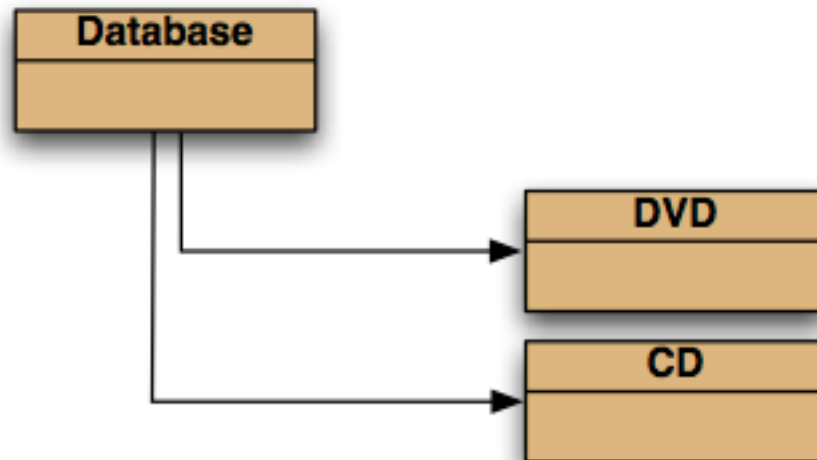
CD
title artist numberOfTracks playingTime gotIt comment
setComment getComment setOwn getOwn print

DVD
title director playingTime gotIt comment
setComment getComment setOwn getOwn print

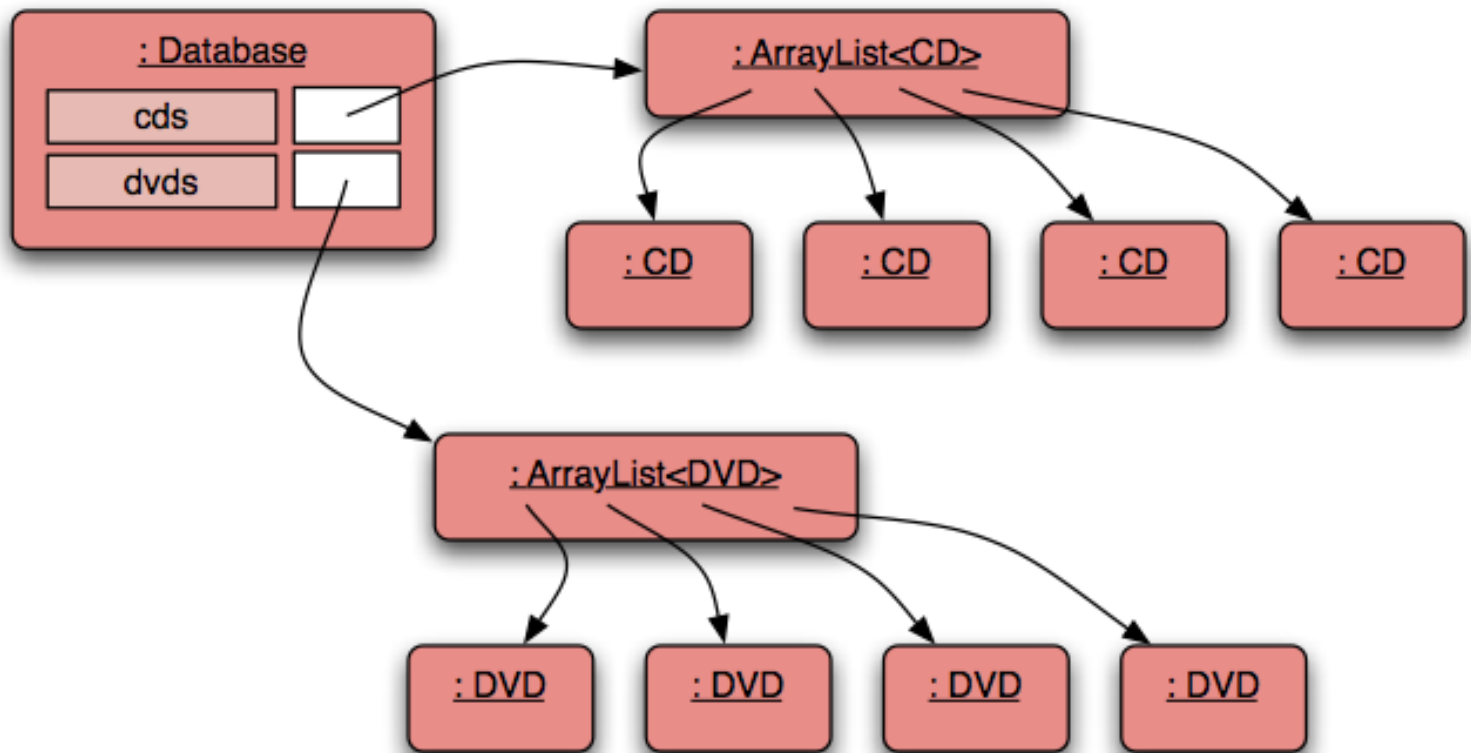
*Metade superior  
exibe campos*

*Metade inferior  
exibe métodos*

# Diagrama de classe



# Modelo de objeto do DoME





# Exercício

- **Database multimídia**

- Inicie o **BlueJ**, abra o projeto *dome-v1* e crie uma instância de *Database*. Crie algumas instâncias de *CD* e de *DVD* e as adicione ao *Database*.
- Liste o conteúdo do *Database*. Insira comentários em um ou mais *CD* e *DVD* e volte a listar o *Database*. Explique o comportamento observado.

# Código-fonte: CD

```
public class CD
{
    private String title;
    ...
    private int playingTime;
    private boolean gotIt;
    private String comment;

    public CD(String theTitle, ..., int time) {
        title = theTitle;
        ...
        playingTime = time;
        gotIt = false;
        comment = "<no comment>";
    }
    ...
}
```

# Código-fonte: DVD

```
public class DVD
{
    private String title;
    ...
    private int playingTime;
    private boolean gotIt;
    private String comment;

    public DVD(String theTitle, ..., int time) {
        title = theTitle;
        ...
        playingTime = time;
        gotIt = false;
        comment = "<no comment>";
    }
    ...
}
```

# Código-fonte: CD

```
public class CD
{
    ...
    public void setComment(String newComment) { ... }
    public String getComment() { ... }
    public void setOwn(boolean ownIt) { ... }
    public void getOwn(boolean ownIt) { ... }
    public void print() { ... }
}
```

# Código-fonte: DVD

```
public class DVD
{
    ...
    public void setComment(String newComment) { ... }
    public String getComment() { ... }
    public void setOwn(boolean ownIt) { ... }
    public void getOwn(boolean ownIt) { ... }
    public void print() { ... }
}
```

# Código-fonte: Database

```
class Database
{
    private ArrayList<CD> cds;
    private ArrayList<DVD> dvds;
    ...
    public void list()
    {
        for(CD cd : cds) {
            cd.print();
            System.out.println();//empty line between items
        }

        for(DVD dvd : dvds) {
            dvd.print();
            System.out.println();//empty line between items
        }
    }
}
```

# Exercício

- **Database multimídia**

- Imagine que o **DoME** agora precise armazenar dados sobre videogames. O que seria necessário implementar ?
- E se fosse necessário tratar mais itens multimídia ? Isto é fácil atualmente no **DoME** ?

# Crítica ao DoME

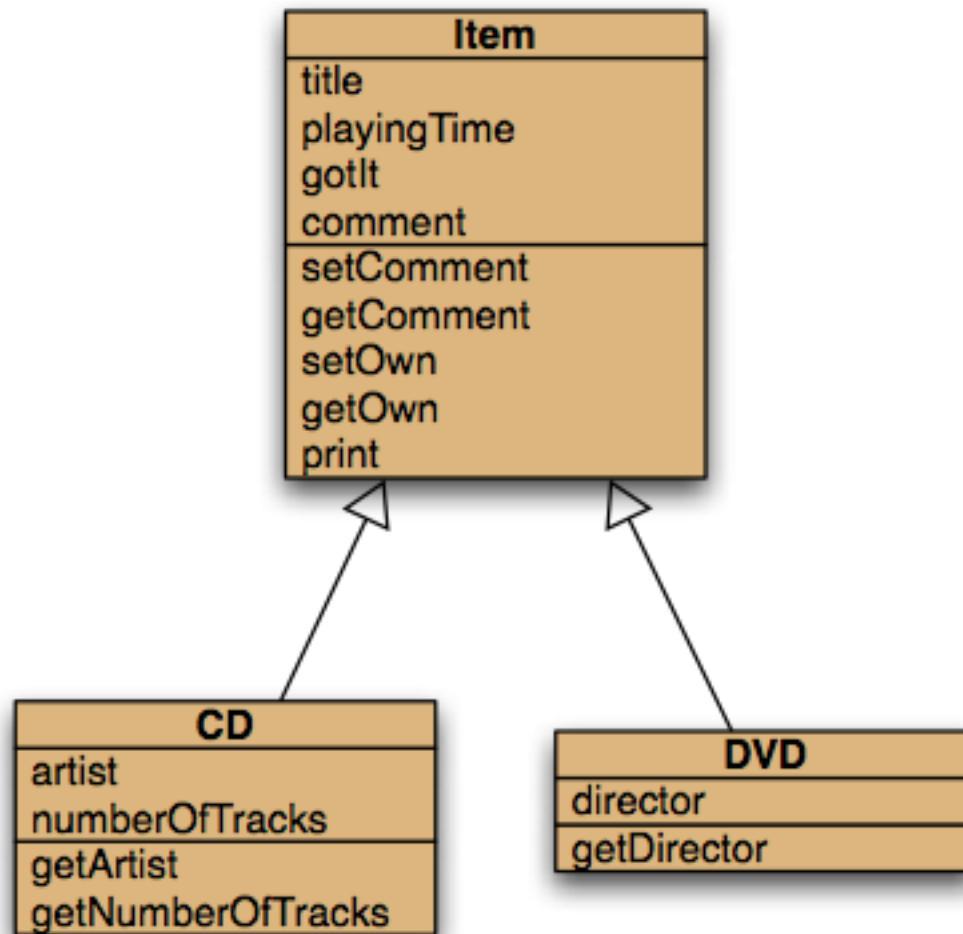
- Duplicação de código:
  - entre as classes **CD** e **DVD**;
  - dentro da classe **Database**.
- Duplicação de código:
  - torna a manutenção mais difícil/trabalhosa;
  - introduz o perigo de bugs devidos a uma manutenção incorreta (inconsistências);
  - é um indicador de design ruim.



# Utilizando herança

- A *herança* é um mecanismo que fornece uma solução para nosso problema de duplicação de código.
- Como a duplicação entre as classes **CD** e **DVD** é eliminada com herança ?

# Utilizando herança



# Utilizando herança

- Define uma **superclasse**: **Item**.
- Define **subclasses** para **CD** e **DVD**.
- A superclasse define os membros (campos e métodos) comuns.
- As subclasses **herdam** os membros da superclasse.
- As subclasses definem os membros específicos.

# Utilizando herança

- A herança é a implementação de um relacionamento conceitual do tipo *‘é um’*.
- Assim, podemos dizer que:
  - ‘um **CD** *é um* **Item**’;
  - ‘um **DVD** *é um* **Item**’.

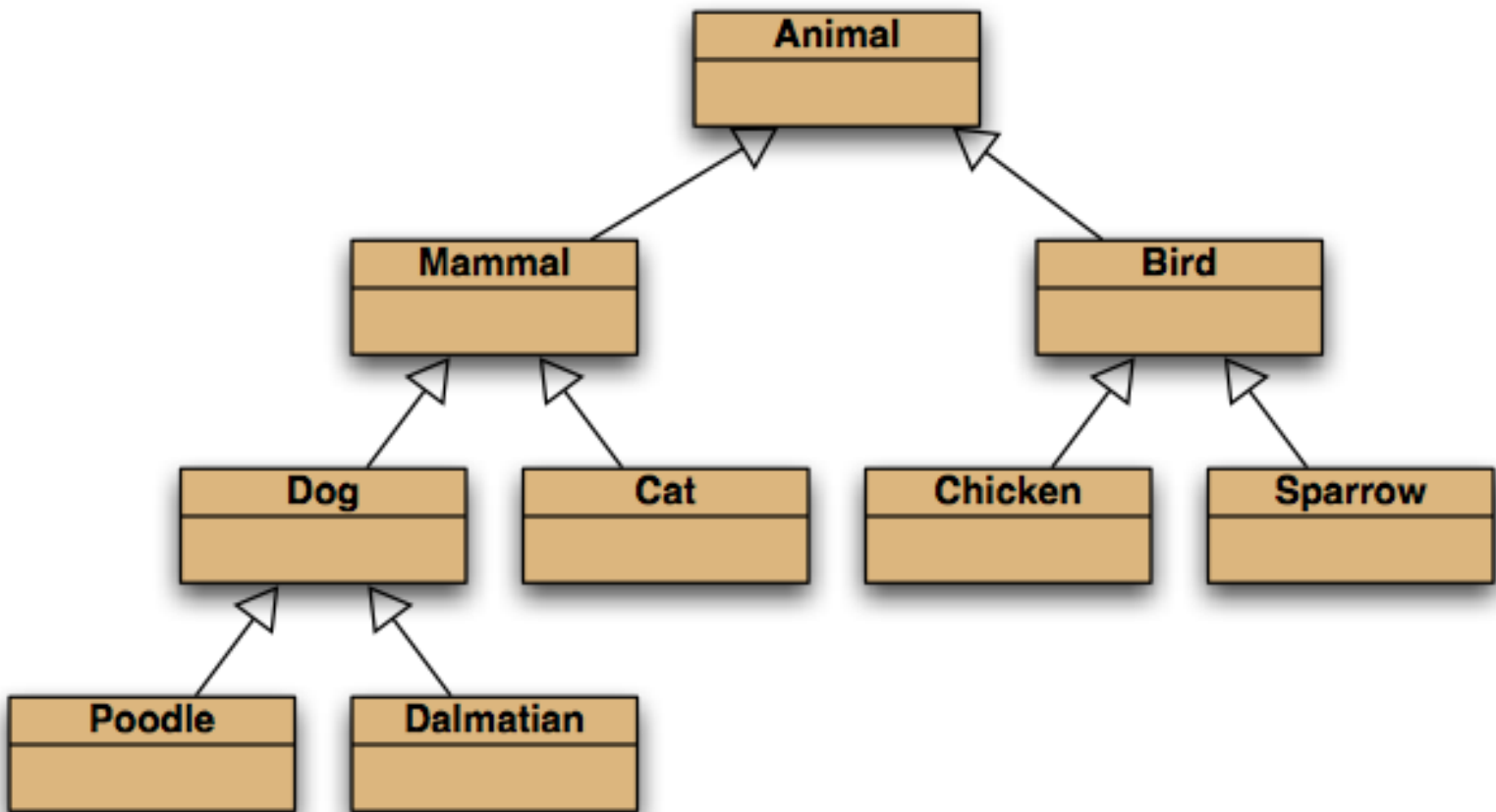
# Análise conceitual (análise de domínio)

- **Generalização** é a atividade de identificar o que há de comum entre conceitos e definir relacionamentos entre superclasses (conceito geral) e subclasses (conceitos especializados).
- Isso expressa a noção de que a subclasse conceitual *é uma espécie* (ou *especialização*) da superclasse conceitual.

# Análise conceitual (análise de domínio)

- As superclasses e subclasses conceituais estão relacionadas em termos de pertinência a conjuntos:
  - todos os membros de um conjunto de subclasse conceitual são membros do conjunto da superclasse.

# Hierarquias de herança



# Herança em Java

```
public class Item  
{  
    ...  
}
```

Aqui  
nenhuma  
alteração

```
public class DVD extends Item  
{  
    ...  
}
```

```
public class CD extends Item  
{  
    ...  
}
```

Aqui existe  
alteração



# Superclasse

```
public class Item
{
    private String title;
    private int playingTime;
    private boolean gotIt;
    private String comment;

    // Construtores e métodos omitidos.
}
```

# Subclasses

```
public class CD extends Item
{
    private String artist;
    private int numberOfTracks;

    // Construtores e métodos omitidos.
}
```

---

```
public class DVD extends Item
{
    private String director;

    // Construtores e métodos omitidos.
}
```

# Herança e construtores

```
public class Item
{
    private String title;
    private int playingTime;
    private boolean gotIt;
    private String comment;

    public Item(String theTitle, int time) {
        title = theTitle;
        playingTime = time;
        gotIt = false;
        comment = "<no comment>";
    }
}
```

# Herança e construtores

```
public class CD extends Item
```

```
{
```

```
    private String artist;
```

```
    private int numberOfTracks;
```

```
    public CD(String theTitle, String theArtist,
```

```
               int tracks, int time) {
```

```
        super(theTitle, time);
```

```
        artist = theArtist;
```

```
        numberOfTracks = tracks;
```

```
}
```

Chamada ao  
construtor da  
supeclasse

# Palavra-chave *super*

- A palavra-chave *super* é uma referência à superclasse imediata.
- Usada (seguida de ponto) para se referir a membros (não privados) da superclasse:

`super.print()` ;  
                  ↑  
Operador ponto      método

- Usada (seguida de parêntesis) em um construtor da subclasse para chamar um da superclasse:

`super(theTitle, time) ;`

# Palavra-chave *super*

- Construtores da subclasse sempre devem conter uma chamada *super()*, que deve ser sua primeira instrução:

**`super(theTitle, time);`**

- Se não existir uma chamada *super()* o compilador inserirá uma sem parâmetros (só funciona se a superclasse tiver um construtor sem parâmetros).

# Exercício

- **Database multimídia**

- Feche o projeto anterior, abra o projeto *dome-v2*. Observe que o diagrama de classes exibe o relacionamento de herança (tente comentar `extends Item`).
- Crie uma instância de *CD*. Quais são os campos do objeto ? (use *Inspect*).
- Verifique como usar um método herdado de *Item* para inserir um comentário no objeto *CD*.

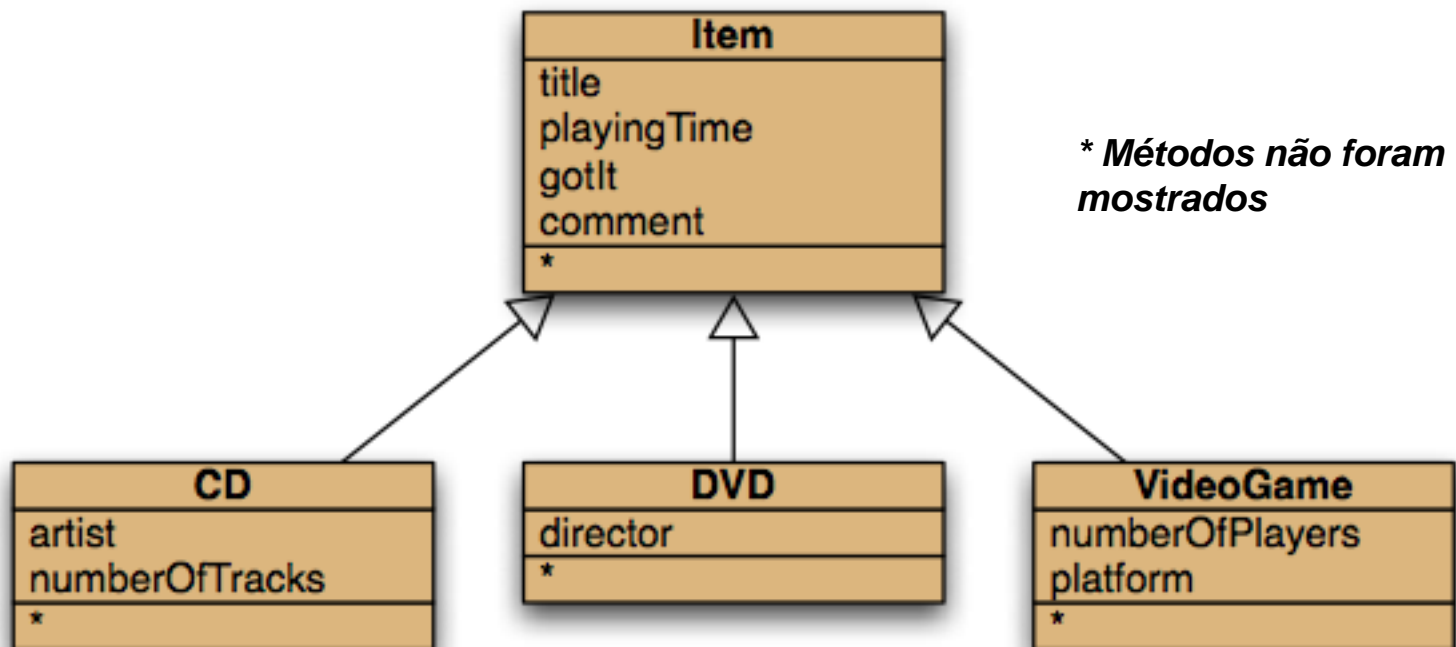
# Exercício

- **Database multimídia**

- Configure um ponto de interrupção no início do construtor da classe *CD*.
- Crie um objeto *CD* e utilize o comando *Step Into* do *Debugger* para percorrer o construtor da superclasse. Observe os campos e sua inicialização.
- Retire o ponto de interrupção na classe *CD*.



# Adicionando outros tipos de item



# Hierarquias mais profundas



# Utilizando herança

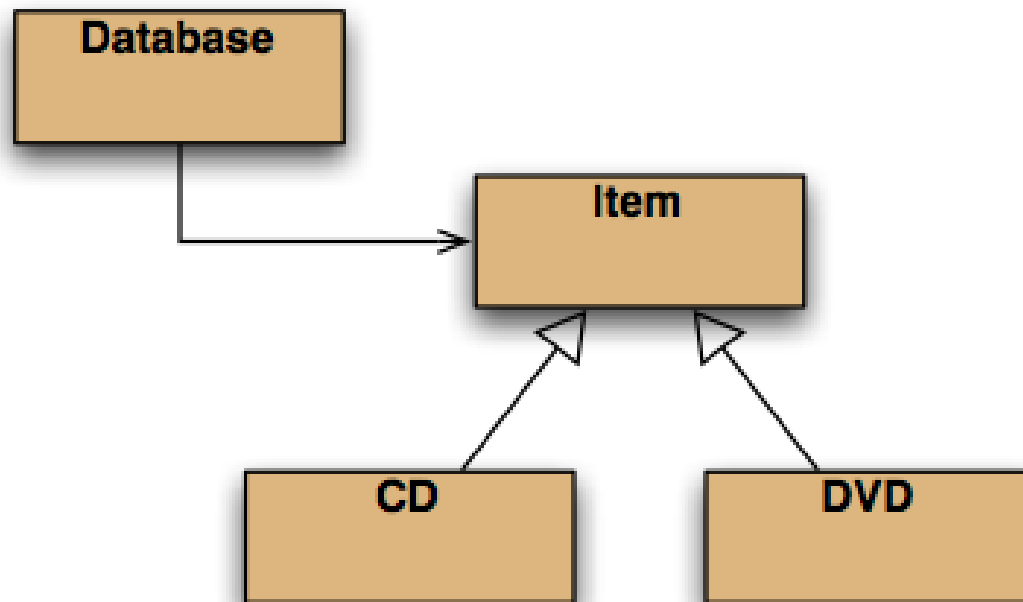
Herança (até aqui) ajuda a:

- evitar duplicação do código;
- reutilizar o código;
- simplificar a manutenção; e
- a extensibilidade.

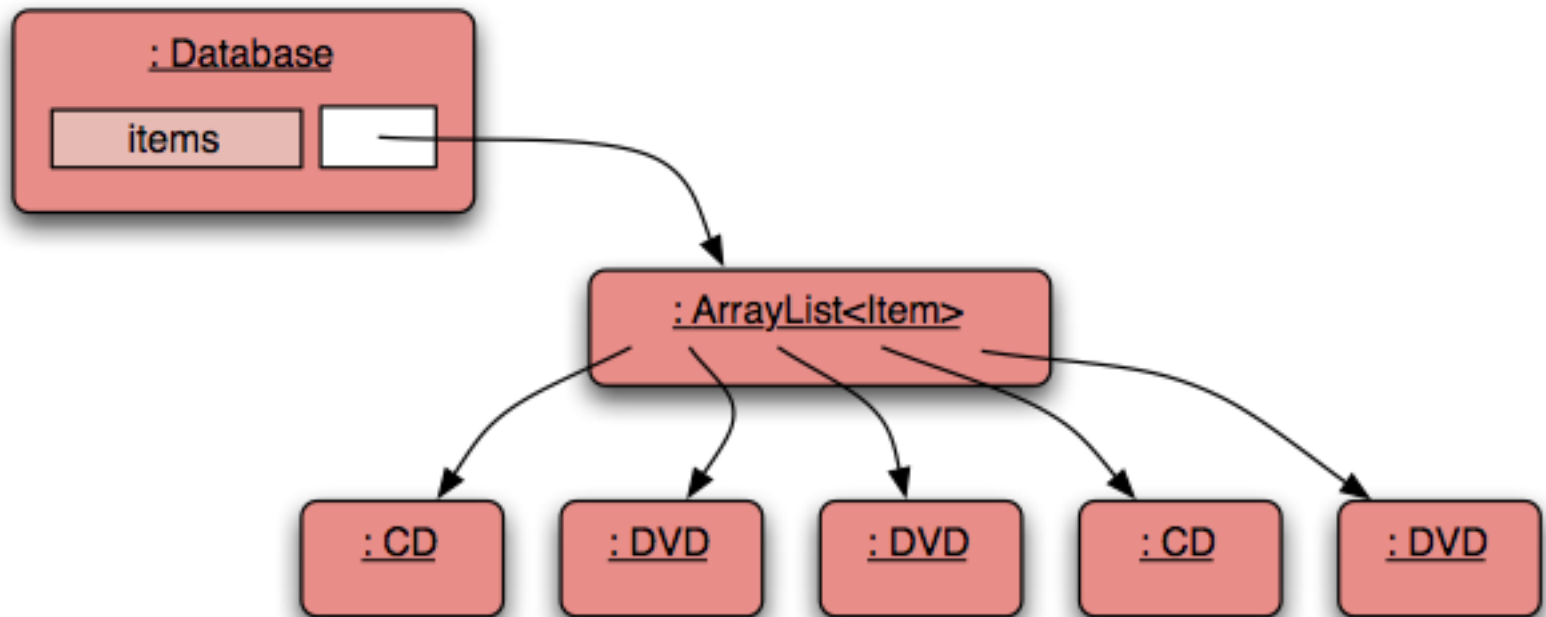
# Utilizando herança

- A *herança* é um mecanismo que fornece uma solução para nosso problema de duplicação de código.
- Como a duplicação dentro da classe **Database** é eliminada com herança ?

# Diagrama de classe




# Diagrama de objeto



# Código-fonte: Database

```
public class Database
{
    private ArrayList<Item> items;


    public Database()
    {
        items = new ArrayList<Item>();
    }
    ...
}
```



Sem  
duplicação  
de código

# Código-fonte: Database

```
public class Database
{
    ...
    public void addItem(Item theItem)
    {
        items.add(theItem);
    }
    ...
    public void list()
    {
        for(Item item : items) {
            item.print();
            // Print an empty line between items
            System.out.println();
        }
    }
}
```



Sem  
duplicação  
de código



# Subtipagem

Primeiro, nós tínhamos:

```
public void addCD (CD theCD)  
public void addVideo (DVD theDVD)
```

Agora, nós temos:

```
public void addItem (Item theItem)
```

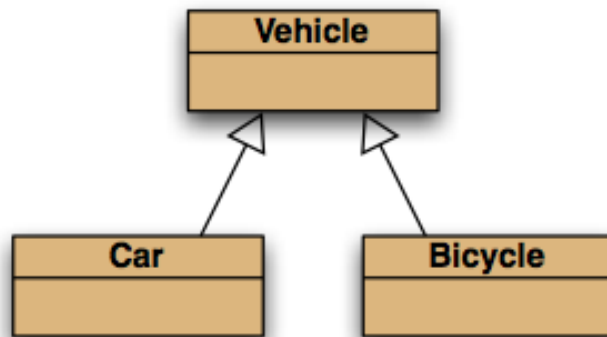
Chamamos este método com:

```
DVD myDVD = new DVD (...);  
database.addItem (myDVD);
```

# Subclasses e subtipos

- Classes definem tipos.
- Subclasses definem subtipos.
- Objetos das subclasses podem ser utilizados onde os objetos dos supertipos são requeridos (isso é chamado **substituição**).

# Subtipagem e atribuição



*Objetos da  
subclasse podem  
ser atribuídos a  
variáveis da  
superclasse*

```
Vehicle v1 = new Vehicle();  
Vehicle v2 = new Car();  
Vehicle v3 = new Bicycle();
```

# Subtipagem e transmissão de parâmetros

```
public class Database
{
    public void addItem(Item theItem)
    {
        ...
    }
}
```

```
DVD dvd = new DVD (...);
CD cd = new CD (...);
```

```
database.addItem(dvd);
database.addItem(cd);
```

*Objetos da subclasse  
podem ser passados  
para os parâmetros  
da superclasse*

# Variáveis polimórficas

- Variáveis de objeto Java são **polimórficas** (podem referenciar objetos de mais de um tipo).
- Elas podem referenciar objetos do tipo declarado ou de seus subtipos.

# Exercício

- **Database multimídia**

- Nesta versão do **DoME** que usa herança, o que é necessário mudar na classe *Database* quando uma nova subclasse de *Item* é adicionada ?

# Exercício

- **Database multimídia**

- Adicione uma nova classe ao projeto *dome-v2* chamada *VideoGame*, definida como subclasse de *Item* (use a opção *New Classe* e *New Inheritance Arrow*).
- Edite o código de *VideoGame*, inclua atributos específicos (*platform* e *numberOfPlayers*), métodos de acesso e ajuste o construtor. Compile a classe.
- Crie uma instância de *VideoGame*, a inspecione e verifique se os métodos funcionam como esperado.

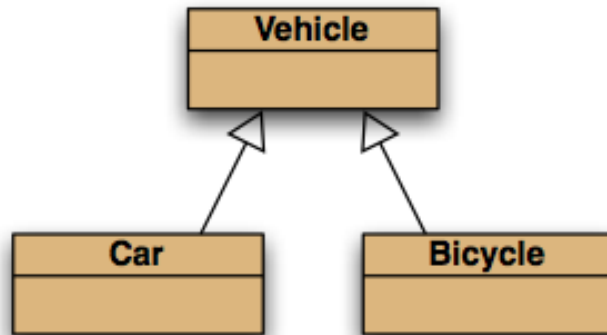
# Exercício

- **Database multimídia**

- Compare o estágio atual do projeto em uso com *dome-v2vg*.
- Remova a classe *VideoGame*.

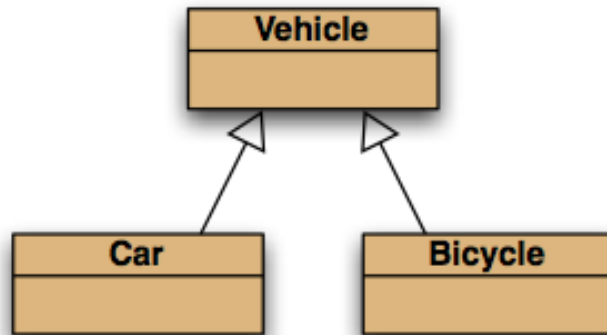


# Conversão de tipos



```
Vehicle v;  
Car c = new Car();  
v = c;           // correto  
c = v;           // erro  
c = (Car) v;     // ok (se v apontar um Car)
```

# Conversão de tipos



Conversão  
pode levar  
a erros.

```
Vehicle v;  
Bicycle b;  
Car c = new Car();  
v = c;                // ok  
b = (Bicycle) c;      // erro compilação  
b = (Bicycle) v;      // erro execução
```

# Conversão de tipos

- Pode atribuir um subtipo ao supertipo.
- Não pode atribuir um supertipo ao subtipo!

```
Vehicle v;  
Car c = new Car();  
v = c; // correct;  
c = v; erro em tempo de compilação!
```

- A conversão de tipos (casting) corrige isso:

```
c = (Car) v;
```

(ok só se v é realmente um Car !)

# Conversão de tipos

- A conversão de tipos deve ser evitada sempre que possível porque ela pode levar a erros em tempo de execução.
- Na prática, a conversão de tipos é raramente necessária em um programa orientado a objetos bem estruturado, pois quase sempre é possível substituir a conversão por métodos polimórficos (que veremos mais adiante).

# Operador *instanceOf*

- Há ocasiões em que precisamos saber o tipo dinâmico de um objeto.
- Em Java, o operador *instanceOf* testa se um objeto é, direta ou indireta, uma instância de uma dada classe.

`obj instanceof MyClass`



Retorna `true` se o tipo dinâmico de `obj`  
for `MyClass` ou uma subclasse de `MyClass`

# Operador *instanceOf*

```
ArrayList<CD> cds = new ArrayList<CD>();  
for(Item item : items) {  
    if(item instanceof CD) {  
        cds.add( (CD) item);  
    }  
}
```

← Testa se `item` é  
instância de `CD`

↖ Caso afirmativo,  
converte o tipo  
de `item` para `CD`  
e o usa como tal

# Classes empacotadoras (wrappers)

- Todos os objetos podem ser inseridos em coleções...
- ... uma vez que as coleções aceitam elementos do tipo parametrizado.
- E valores de tipos primitivos ?  
Como colocá-los em coleções ?

# Classes empacotadoras (wrappers)

- Tipos primitivos (`int`, `char`, etc.) não são objetos. Eles precisam ser empacotados em um objeto!
- Há classes empacotadoras para todos os tipos primitivos:

<i>tipo simples</i>	<i>classe empacotadora</i>
<code>int</code>	<code>Integer</code>
<code>float</code>	<code>Float</code>
<code>char</code>	<code>Character</code>
<code>...</code>	<code>...</code>



# Classes empacotadoras (wrappers)

```
int i = 18;  
Integer iwrap = new Integer(i); ← empacota i  
...  
int value = iwrap.intValue(); ← desempacota i
```

# Encaixotamento automático (Autoboxing)

- Sempre que um valor de um tipo primitivo é utilizado em um contexto que exija um tipo objeto, o compilador empacota automaticamente o valor do tipo primitivo em um objeto empacotador apropriado. Esta operação é chamada *autoboxing*.
- A operação inversa, também executada automaticamente pelo compilador, é chamada *unboxing*.

# Encaixotamento automático (Autoboxing)

```
private ArrayList<Integer> markList;  
...  
public void storeMarkInList(int mark) {  
    markList.add(mark);  ← autoboxing  
}  
...  
public int removeFirstMarkInList() {  
    int firstMark = markList.remove(0); ← unboxing  
    return firstMark;  
}
```

# Crítica ao DoME

- A versão atual do DoME ainda apresenta alguns problemas:
  - o método **print** não imprime todos os detalhes (trataremos disso no próximo cap.);
  - nada impede que criemos um objeto **Item** (o que não faz sentido no domínio de um banco de dados de entretenimento multimídia).

# Análise conceitual (análise de domínio)

- As superclasses e subclasses conceituais estão relacionadas em termos de pertinência a conjuntos:
  - se todo membro de uma classe C deve ser também membro de uma subclasse de C, a classe C é uma *classe conceitual abstrata*.

# Palavra-chave *abstract*

- Em Java, a presença da palavra-chave *abstract* em uma definição de classe a torna uma classe de software abstrata.
- Uma classe abstrata não pode ser instanciada.
- As classes que não são abstratas são chamadas de classes concretas.

```
public abstract class Item
{
    // Campos, construtores e métodos omitidos.
}
```

# Exercício

- **Database multimídia**

- Crie uma instância de *Item*.  
Inspeccione os campos do objeto.
- Edite a classe *Item* e a defina como uma classe abstrata. Compile.
- Note no diagrama de classes a presença do estereótipo `<<abstract>>` na classe *Item* e a ausência da opção *new* no menu pop-up desta classe.
- Compare o estágio atual do projeto em uso com *dome-v3*.

# Revisão (1)

- A herança permite a definição de classes como extensões de outras classes.
- Herança:
  - evita a duplicação do código;
  - permite a reutilização do código;
  - simplifica o código; e
  - simplifica a manutenção e a extensão.



## Revisão (2)

- Variáveis do supertipo podem referenciar objetos do subtipo.
- Subtipos podem ser utilizados sempre que esperamos objetos do supertipo (substituição).
- Classes abstratas não podem ser instanciadas, mas são extendidas.



# Contatos

Câmara dos Deputados  
CENIN - Centro de Informática

Carlos Renato S. Ramos

[carlosrenato.ramos@camara.gov.br](mailto:carlosrenato.ramos@camara.gov.br)