Documentar as classes

- Documentar as nossas classes tal como as classes do Java
- Qualquer pessoa deve poder usar as nossas classes, sem ter que ler a implementação
  - Nome da classe
  - Comentário descrevendo o objectivo geral da classe
  - Versão (nº)
  - Nome do(s) autor(es)
  - Documentação para cada construtor e cada método
    - » Nome do método/construtor
    - » Tipo de retorno
    - » Nomes e tipos dos parâmetros
    - » Descrição do objectivo do método
    - » Descrição de cada parâmetro
    - » Descrição do valor devolvido

javadoc

```
/**
*  Exemplo de um comentario em javadoc
*/
```

Comentário imediatamente antes da declaração da classe é lido como um comentário de classe

Comentário antes da assinatura de um método é lido como um comentário de método

@version
@author
@param
@return

Em javadoc os símbolos especiais para formatar a documentação começam por @

Javadoc
Comentário de classe

```
/**
 * The DVD class represents a DVD object. Information about the
 * DVD is stored and can be retrieved. We assume that we only deal
 * with movie DVDs at this stage.
 *
 * @author Michael Kolling and David J. Barnes
 * @version 2008.03.30
 */
public class DVD
{ ...}
```


Documentação gerada com Javadoc para a classe DVD:

```
public class DVDextends java.lang.Object
```

The DVD class represents a DVD object. Information about the DVD is stored and can be retrieved. We assume that we only deal with movie DVDs at this stage.

Version:
2008.03.30
Author:
Michael Kolling and David J. Barnes

Constructor Summary

DVD(java.lang.String theTitle, java.lang.String theDirector, int time) Constructor for objects of class DVD

Method Summary

java.lang.String	getComment()
boolean	getOwn()

Javadoc
Comentário de método

```
/**
     * Constructor for objects of class DVD
     * @param theTitle The title of this DVD.
     * @param theDirector The director of this DVD.
     * @param time The running time of the main feature.
     */
    public DVD(String theTitle, String theDirector, int
   time)
    {...}

/**
     * @return The comment for this DVD.
     */
    public String getComment()
    {
        return comment;
    }
```


Documentação gerada com Javadoc para métodos:

Constructor Detail

DVD

```
public DVD(java.lang.String theTitle,
           java.lang.String theDirector,
           int time)
```

Constructor for objects of class DVD

Parameters:

`theTitle` - The title of this DVD.
`theDirector` - The director of this DVD.
`time` - The running time of the main feature.

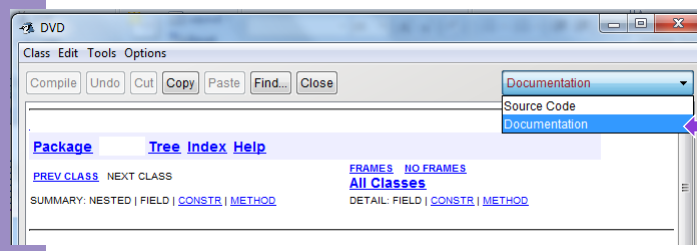
Method Detail

getComment

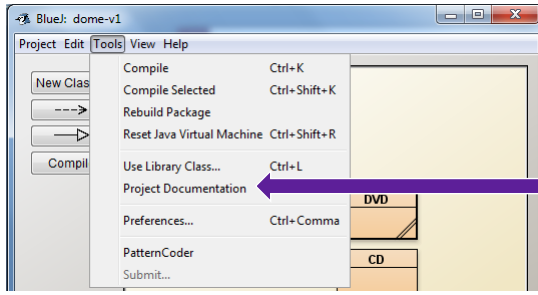
```
public java.lang.String getComment()
```

Returns:

The comment for this DVD.



Para gerar a documentação de uma classe (na janela do editor)



Para gerar a documentação de todas as classes (no menu principal)

Melhorar estrutura da aplicação

- Conceitos principais:
  - » **Herança**
  - » **Polimorfismo**

Caso prático: Projecto DoME (Database of Multimedia Entertainment)

Fonte: Cap 8 de Objects first with Java: a practical introduction using BlueJ, David Barnes, Michael Kolling, Prentice Hall/Pearson

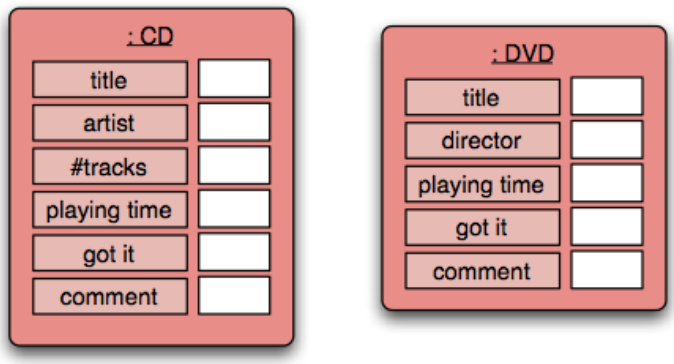
Caso prático DoME

- Funcionalidades pretendidas
  - Registar informação de CDs e DVDs
  - Pesquisar informação
  - Listar informação de CDs e DVDs
  - Eliminar informação

Principais conceitos a abordar

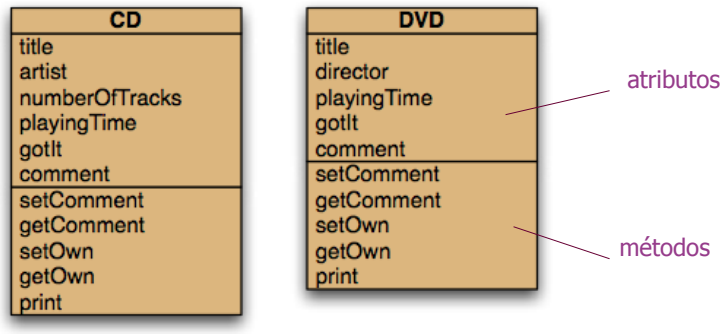
- Herança (Inheritance)
- SubTipos
- Substituição
- Variáveis polimórficas

Principais objectos do projecto DoME



Fonte: Objects first with Java: a practical introduction using BlueJ, David Barnes, Michael Kolling, Prentice Hall/Pearson

Classes do projecto DoME



Modelo de objectos do DoME

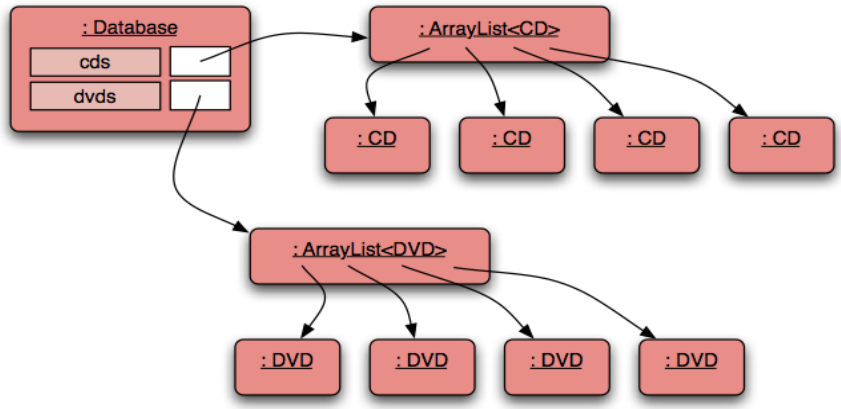
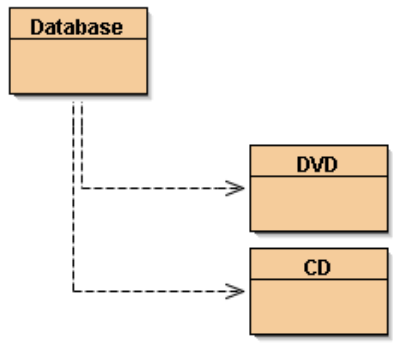


Diagrama de classes DoME



- **Crie um objecto Database**
- **Crie alguns objectos DVD e CD**
- **Insira os objectos criados na base de dados**
- **Liste a informação da BD**
- **Insira um comentário num CD**
- **Quando listar de novo a informação o comentário aparece?**

E se quisermos agora adicionar objectos do tipo livro?

Classes DoME

- Analise o código das classes
  - Repetições?
  - Como evitar?

HERANÇA

Herança (Inheritance)

- Reutilização

  Mecanismo que permita a reutilização da definição da classe já existente, ou seja a definição de uma classe a partir de classes já existentes

- Permite definir uma classe como uma extensão de outra classe
- Relação "is-a" e relação "has-a"
- "is-a" – herança : carro "is-a" veículo
- "has-a" – composição: carro "has-a" volante

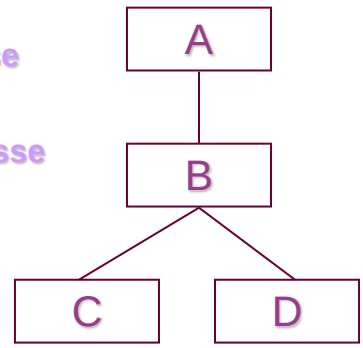
HERANÇA

- **Mecanismo para organizar as classes e os seus comportamentos**
- **Possibilidade de criar uma classe especificando em que difere de outra existente**
- **Dá um acesso directo à informação contida nessa outra classe**
- **Cada classe tem:**
  Uma SUPERCLASSE (a classe acima na hierarquia)
  e pode ter:
  uma ou mais SUBCLASSES (classes abaixo na hierarquia)

Hierarquia de classes

A é **superclasse** de B

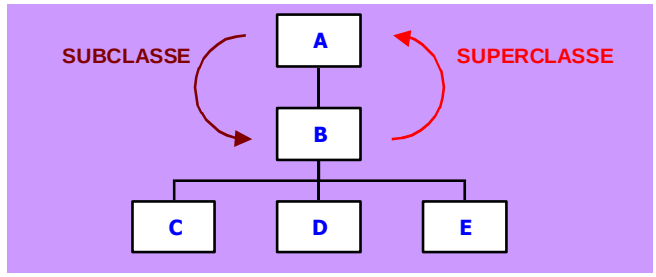
B é **subclasse** de A



Todas as classes se encontram hierarquicamente relacionadas entre si

HERANÇA

- AS CLASSES ABAIXO NA HIERARQUIA HERDAM DAS CLASSES ACIMA.



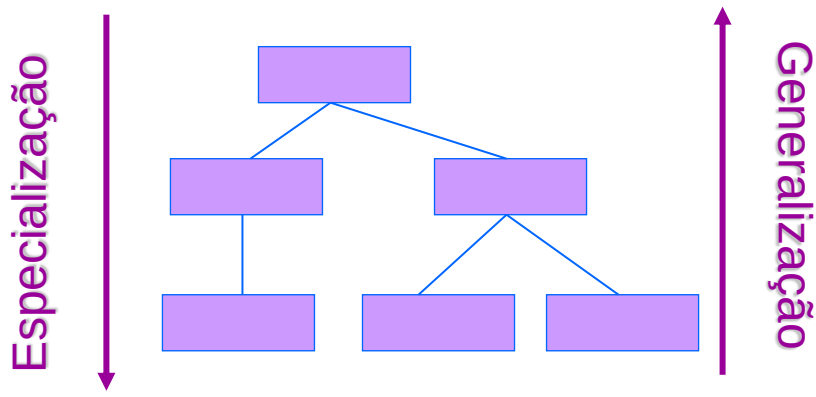
B herda de **A** todas as variáveis e métodos de instância que não sejam declarados como private

B pode definir novas variáveis e novos métodos próprios

B pode redefinir variáveis e métodos herdados

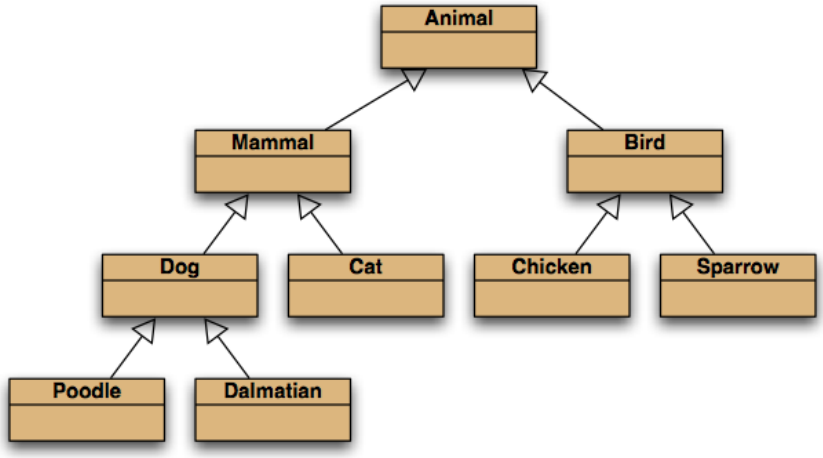
A herança não se aplica a variáveis e métodos de classe

Hierarquia de especialização



HERANÇA

Hierarquia de herança



Superclasse/Subclasse

Superclasse	Subclasse
Aluno	Aluno 2º ciclo, aluno 1º ciclo
Figura	Círculo, Triângulo, Rectângulo
Funcionário	Docentes, Administrativo, Auxiliar
Pessoa	Aluno, Funcionário

Exemplo

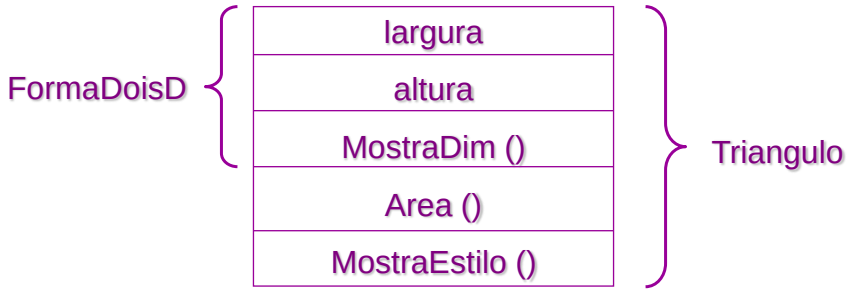
```
class FormaDoisD { // classe para objectos bidimensionais
    double largura;
    double altura;

    public void MostraDim() {
        System.out.println ("largura e altura "+ largura + "e" + altura);
    }
}  // fim class FormaDoisD

Class Triangulo extends FormaDoisD {
    String tipo;

    public double area() {
        return largura * altura /2; }

    public void MostraEstilo() {
        System.out.println("Triangulo e " + tipo);}
}  // fim class Triangulo
```

class subclasse_nome extends superclasse_nome {

... }

```
class FormaDoisD {              // classe para objectos bidimensionais
    private double largura;
    private double altura;

    public double Getlargura () { return largura;}
    public double Getaltura () { return altura;}
    public void Setlargura (double l) {largura=l;}
    public void Setaltura (double a) {altura=a;}
    public void MostraDim() {
        System.out.println ("largura e altura "+ largura + "e" + altura);
    }
}  // fim class FormaDoisD

Class Triangulo extends FormaDoisD {
    String tipo;

    public double area() {
        return Getlargura () * Getaltura () /2; }

    public void MostraEstilo() {
        System.out.println("Triangulo e " + tipo);}
}  // fim class Triangulo
```


- HERANÇA: métodos e variáveis

⇒ Não há necessidade de redefinir ou copiar código de outras classes

- COLECÇÃO GRANDE DE CLASSES

⇒ **ESTRUTURAR UMA HIERARQUIA**

- Concentrar nas superclasses o comum
- Acrescentar ou alterar uma classe no topo implica que todas as subclasses alterem o seu comportamento (sem alterar ou recompilar) porque recebem por herança
- Só se define um atributo ou comportamento uma vez na hierarquia

FUNCIONAMENTO DA HERANÇA

VARIÁVEIS

Uma nova instância de uma classe tem espaço para cada variável definida na classe corrente e para cada uma das variáveis das superclasses.

MÉTODOS

Os novos objectos têm acesso a todos os métodos da sua classe e das suas superclasses, mas com uma escolha dinâmica na hierarquia de classes, desde a classe corrente até ao topo.


```
public class Pessoa {
        private String Nome;
        public Pessoa ()
        { Nome = "ainda não tem";}
        public Pessoa (String oNome)
        { Nome = oNome;}
        public String GetNome ()
        {return Nome;     }
        public void mudaNome (String NovoNome)
        { Nome = NovoNome;}
        public void imprime ()
        { System.out.println ("Nome " + Nome); }
        public boolean mesmoNome (Pessoa outraPessoa)
        { return (this.Nome.equalsIgnoreCase(outraPessoa.Nome));}
}
```

```
public class Aluno extends Pessoa       {
        private int ano;
        public aluno ()
        {       super();
                ano = 1;  }
        public aluno (String oNome, int oAno)
        {       super(oNome);
                ano = oAno;       }
        public void muda (String novoNome, int novoAno)
        {       mudaNome (novoNome);
                ano = novoAno;    }
        public int GetAno ()
        {       return ano;       }
        public void SetAno (int novoAno)
        {       ano = novoAno;    }
        public void imprime ()
        {System.out.println ("nome "+ GetNome() + "ano " + ano);  }
}       // fim class Aluno
```

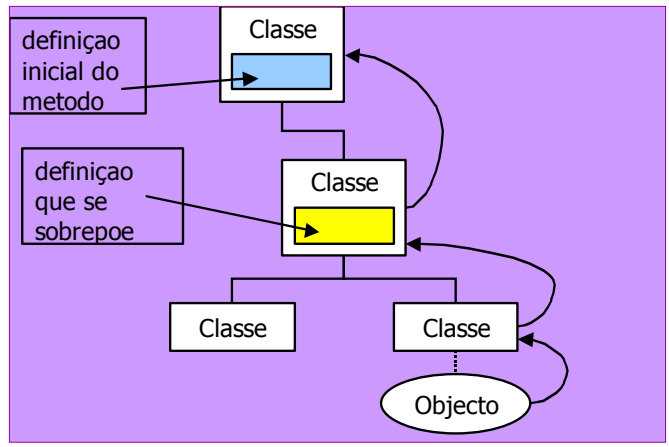
Invocação do construtor da superclasse – **deve ser sempre a 1ª linha do construtor da subclasse**

Sobreposição – método imprime () da classe Pessoa

Chamada ao construtor da Superclasse

- Construtores das subclasses devem ter sempre uma chamada "super"
- Se não for escrita nenhuma, o compilador insere uma (sem parâmetros)
  - Só funciona se a superclasse tiver um construtor sem parâmetros
- Tem que ser a primeira instrução no construtor na subclasse

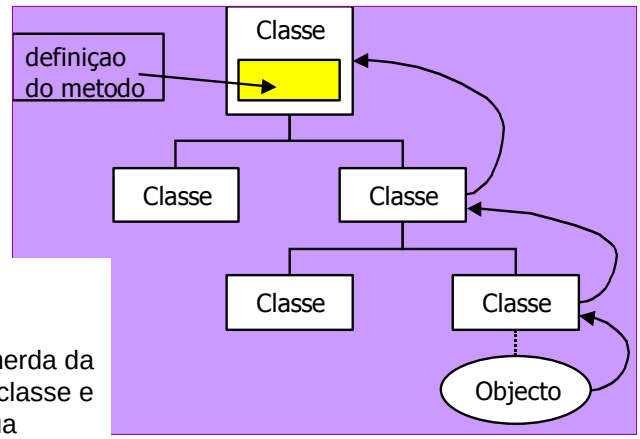
Se uma subclasse define um método com a mesma assinatura (nome, número e tipo de argumentos) – SOBREPOSIÇÃO, ***OVERRIDING***, é este que é executado



Sobreposição e sobrecarga

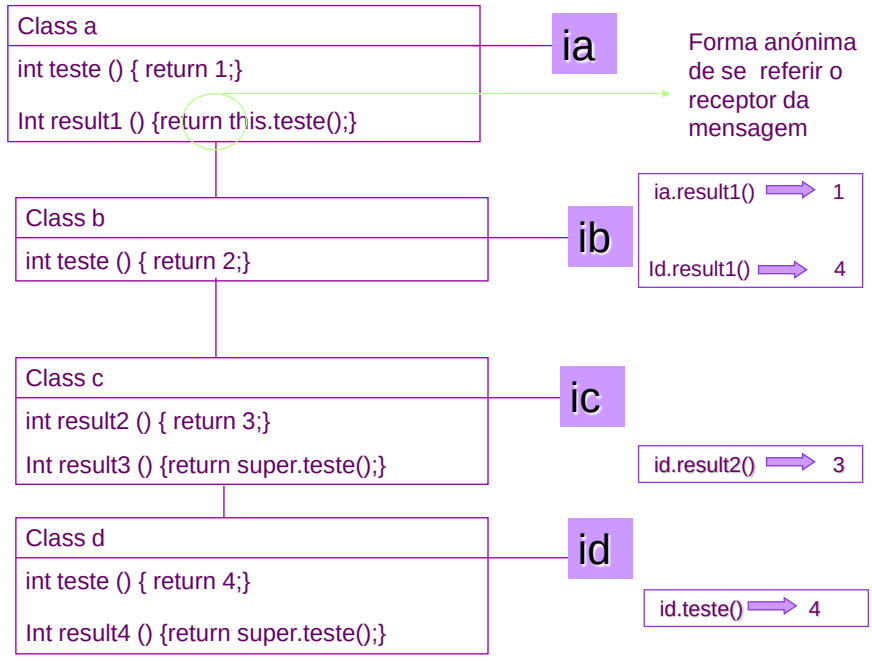
- **Sobreposição**, **OVERRIDING** – redefinição de um método herdado, mantendo o seu nome, tipo e ordem dos parâmetros e resultado, mas modificando o seu código
- **Sobrecarga**, **OVERLOADING** – criação de um método com o mesmo nome dos já existentes na classe ou herdados, mas que têm parâmetros com tipos e/ou ordens diferentes e, eventualmente diferentes tipos de resultado

LOCALIZAÇÃO DE UM MÉTODO



Herança transitiva:

Classe x herda da sua superclasse e esta da sua



Polimorfismo

- A sobreposição de métodos é a base do conceito **procura dinâmica de métodos** (*dynamic method dispatch)* – mecanismo pelo qual a chamada a um método sobreposto é resolvida durante a execução e não durante a compilação
- O tipo de objecto referenciado é que determina qual a versão de um método sobreposto vai ser executada


```
class Sub {
  void who() {
    System.out.println("who() em Sub");
  }
}

class Sub1 extends Sub {
  void who() {
    System.out.println("who() em Sub1");
  }
}

class Sub2 extends Sub {
  void who() {
    System.out.println("who() em Sub2");
  }
}
```

```
class DynDispDemo {
 public static void main(String args[]) {
  Sub superOb = new Sub();
  Sub1 subOb1 = new Sub1();
  Sub2 subOb2 = new Sub2();

  Sup supRef;

  supRef = superOb;
  supRef.who();

  supRef = subOb1;
  supRef.who();

  supRef = subOb2;
  supRef.who();
 }
}
```

Em cada caso, a versão de who() a chamar é determinada em run time pelo tipo de objecto referenciado

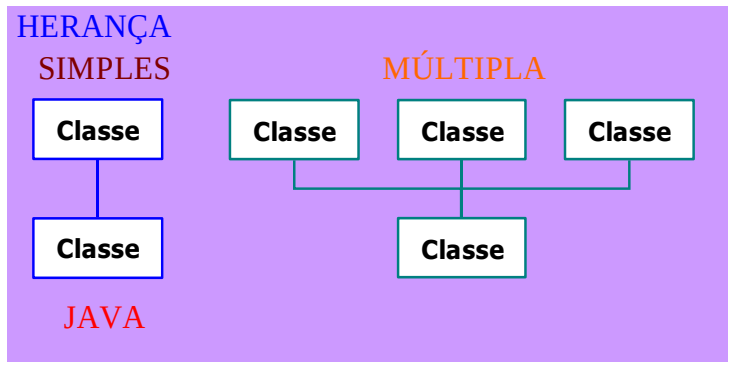
Resultado:

Who() em Sub

Who() em Sub1

Who() em Sub2

Herança Simples e Múltipla

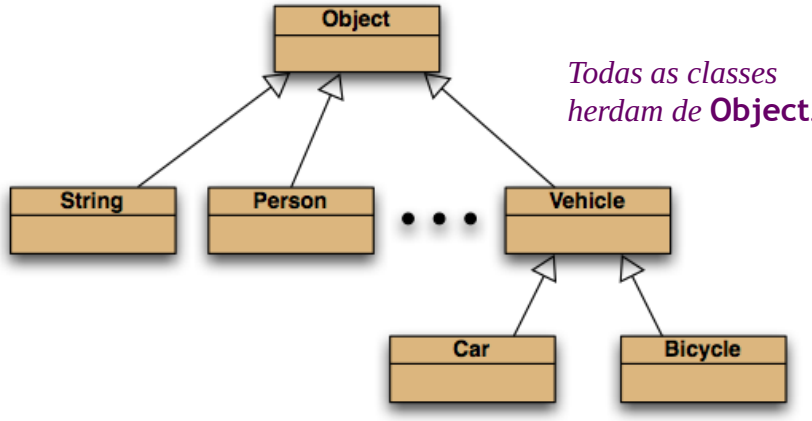


Em Java qualquer classe, excepto a classe Object, tem sempre, no máximo, uma superclasse
herança simples

Outras linguagens POO uma classe pode ser subclasse de mais do que uma classe

herança múltipla

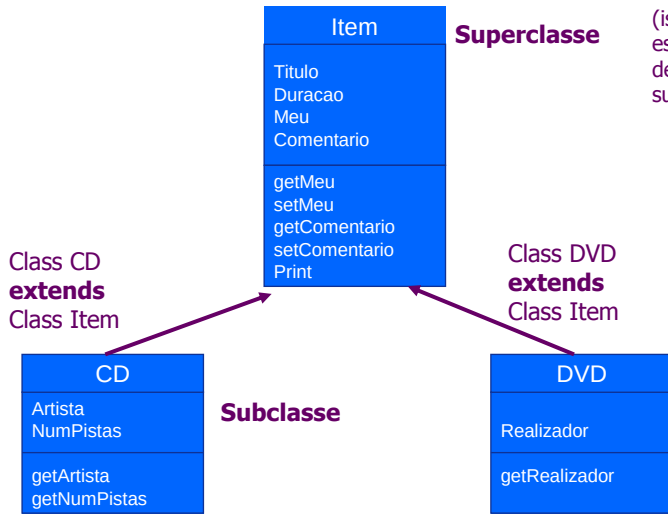
A classe Object em Java



Todas as classes herdam de **Object**.

Herança = relacionamento *is-a*

Subclasse é (is-a) especialização de uma superclasse



Herança em Java

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

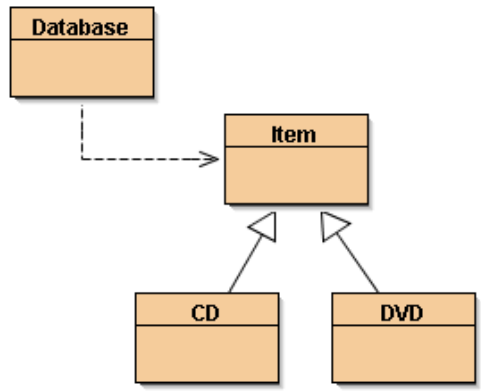
Nenhuma alteração

Altera aqui

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

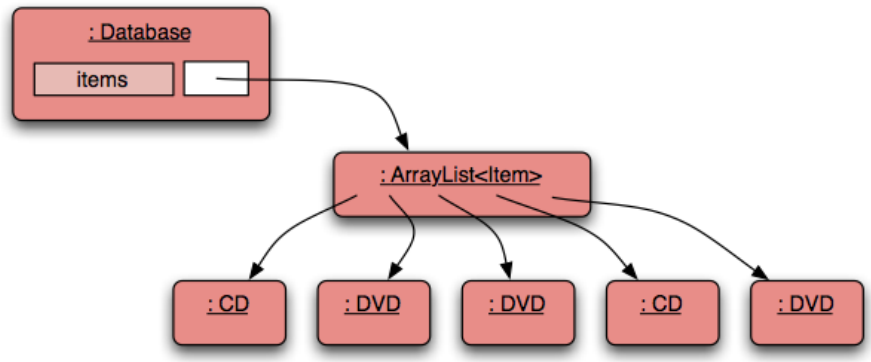
```
public class CD extends Item
{
    ...
}
```


Exemplo: BD de CDs e DVDs – versão Herança

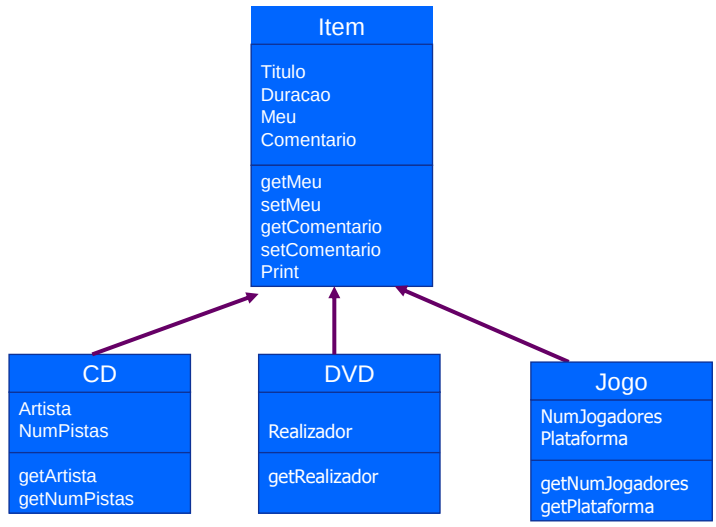


Possibilita REUTILIZAÇÃO de classes

Diagrama de objectos



BD de CDs, DVDs e Jogos



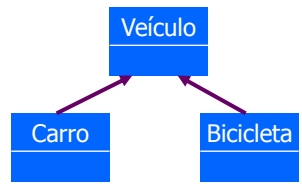
Subclasses e subtipos

- Classe define um tipo
  - Tipo de objecto criado pela classe DVD é DVD
- Classes podem ter subclasses
  - Subclasses definem subtipos
    - » O tipo DVD é um subtipo de Item
- Hierarquia de tipos: o tipo definido por uma subclasse é um subtipo do tipo da sua superclasse
- Objectos das subclasses podem ser usados onde são requeridos objectos dos supertipos

  → substituição

Variáveis, subtipos e atribuições

- Uma variável pode conter objectos do seu tipo declarado ou de qualquer subtipo do seu tipo declarado



Objectos de subclasses podem ser atribuidos a variáveis de superclasses

```
Veiculo V1 = new Veiculo();
Veiculo V2 = new Carro();
Veiculo V3 = new Bicicleta();
```

Todos legais para armazenar Veiculo

Princípio da SUBSTITUIÇÃO

A atribuição de um objecto subtipo a um objecto supertipo é permitida

Variáveis, subtipos e atribuições

- Mas
  - Carro c1 = new Veiculo();

  ou
  - Carro c3 = new Bicicleta();

ERRO

Exemplo

- Considere 4 classes: Pessoa, Professor, Aluno e alunoDout. Professor e Aluno são ambas subclasses de Pessoa. AlunoDout é uma subclasse de Aluno.
- Quais das seguintes atribuições são verdadeiras, porquê?
  - Pessoa p1 = new Aluno ();
  - Pessoa p2 = new AlunoDout ();
  - AlunoDout ad1 = new Aluno ();
  - Professor pf1 = new Pessoa();
  - Aluno a1 = new AlunoDout();

  - a1 = p1;
  - a1 = p2;
  - p1 = a1;
  - pf1 = a1;
  - a1 = ad1;

Subtipos e passagem de parâmetros

```
public class Database
{

    public void addItem(Item theItem)
    {
        ...
    }
}

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

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

objectos de subclasses podem ser passados como parâmetros das superclasses

Variáveis polimórficas

- Variáveis objecto em Java são **polimórficas**
  podem conter objectos de mais do que um tipo
  - » O tipo declarado ou qualquer subtipo do tipo declarado
- Podem conter objectos do tipo declarado, ou de subtipos do tipo declarado

```
public void list()
    {
        for(Item item : items) {
            item.print();
            System.out.println();
        }
    }
```

Apesar da lista de elementos conter CDs e DVDs pode-se percorrer a lista usando uma variável do tipo **Item**

Casting

- Veiculo v;
- Car c = new Car();
- v = c;
- c = v; **ERRO – não é permitido atribuir um supertipo a um subtipo**

Mas

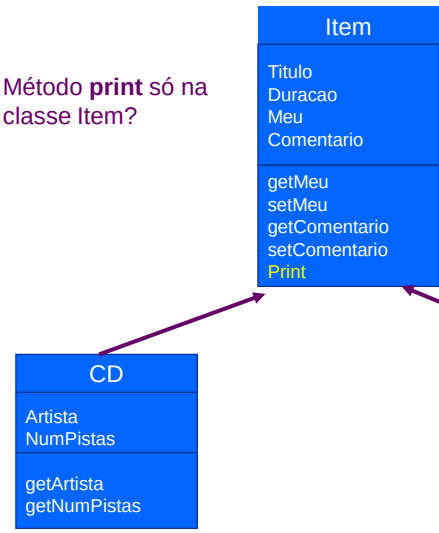
c = (Car) v; OK (se o veiculo for realmente um carro)

Mas, seguinte sequência de instruções

```
Car c;
Veiculo v;
Bicicleta b;
c = new Car();
v = c;
b = (Bicicleta) c;      // ERRO COMPILAÇÃO
b = (Bicicleta) v;      // ERRO EXECUÇÃO
```


Herança

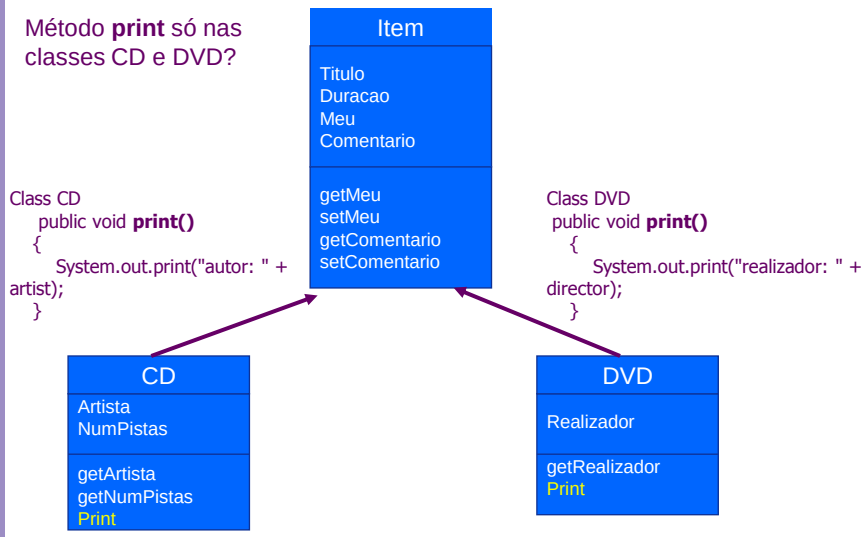
Método **print** só na classe Item?




```
Class Item
 public void print()
 {
       System.out.print("title: " + title + " ("
+ playingTime + " mins)");
       if(gotIt) {
          System.out.println("*");
       } else {
          System.out.println();
       }
       System.out.println("    " + comment);
   }
```

Herança

Método **print** só nas classes CD e DVD?




```
Class CD
    public void print()
    {
        System.out.print("autor: " +
artist);
    }
```

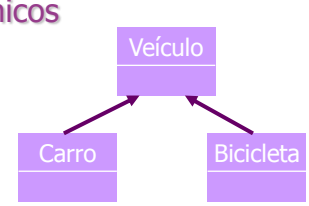
```
Class DVD
 public void print()
    {
        System.out.print("realizador: " +
director);
    }
```

E o que acontece a **item.print();** na classe Database?

O problema

- O método `print` em `Item` só imprime os atributos comuns
- Herança é "uma rua num sentido".
  - Uma subclasse herda os atributos da superclasse
  - A superclasse nada sabe dos atributos das suas subclasses
- São necessários novos conceitos
  - Tipo estático
  - Tipo dinâmico
  - ***method lookup, method binding, method dispatch***

Tipos estáticos e dinâmicos



- Carro c1 = new Carro()
- O tipo de c1 é Carro
  - O tipo da variável c1
    Ou
  - O tipo do objecto armazenado em c1

Sem herança é indiferente

- Veiculo v1 = new Carro();
- Qual o tipo de v1? Depende do que se entende por tipo de v1
  - O tipo da variável v1 é Veiculo
  - O tipo do objecto armazenado em v1 é Carro

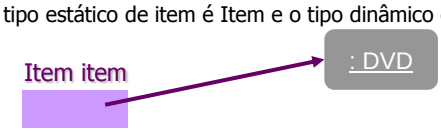
Tipos estáticos e dinâmicos

- Chama-se ao tipo declarado da variável o **Tipo estático (static type),** porque é declarado no código fonte – a representação estática do programa
- Chama-se ao tipo do objecto armazenado na variável o **Tipo dinâmico (dinamic type),** porque depende de atribuições durante a execução – o comportamento dinâmico do programa
- Compilador verifica violações do tipo estático
- Veiculo v1 = new Carro();
  - O tipo estático de v1 é Veiculo
  - O tipo dinâmico de v1 é Carro

Se não existisse um método print() em Item, mas apenas em CD e DVD?

```
for(Item item : items)
        item.print();
```

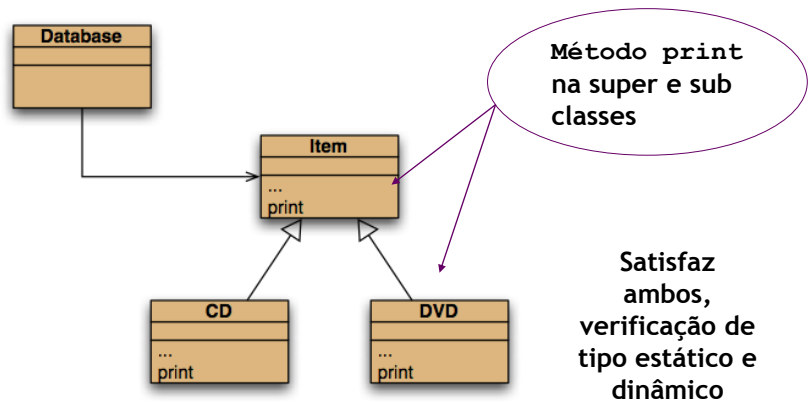
aquando a chamada o tipo estático de item é Item e o tipo dinâmico é CD ou DVD



Para verificação de tipos, durante a compilação, é usado o tipo estático

Cada classe vai ter um método Print()

A solução: Sobreposição



Satisfaz ambos, verificação de tipo estático e dinâmico

Herança

Class Item

```
public void print()
{
    System.out.print("title: " + title + " (" + playingTime + " mins)");
    if(gotIt) {
        System.out.println("*");
    } else {
        System.out.println();
    }
    System.out.println("    " + comment);
}
```

Class CD

```
public void print()
{
    System.out.print("autor: " + artist);
}
```

Class DVD

```
public void print()
{
    System.out.print("realizador: " + director);
}
```

Criar instância CD

Criar instância DVD

Introduzir CD e DVD na BD

Executar Print da BD

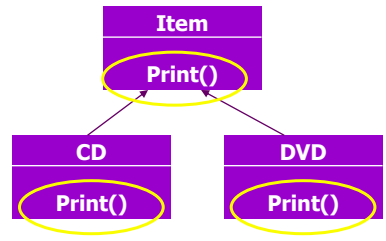
O que imprime?

```
Class Database
} public void list()
  {
     for(Item item : items) {
        item.print();
        System.out.println();
     }
  }
```

?

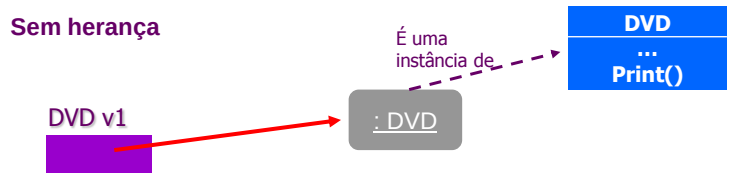
Sobreposição, OVERRIDING

- Sobreposição = redefinição
- Método com a mesma assinatura na superclasse e na subclasse mas com "corpo" diferente



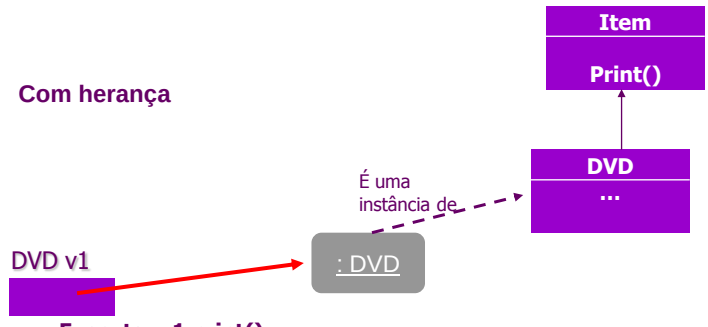
- A verificação de tipos usa o tipo estático, mas durante a execução, são executados os métodos do tipo dinâmico
  - É necessário um método print na classe Item()
  - Mas são executados os métodos das subclasses DVD e CD

- Como são chamados os métodos?
  - *method lookup, method binding, method dispatch*



  - **Executar v1.print()**
    - » A variável v1 é acedida
    - » É encontrado o objecto armazenado nesta variável (seguindo a referência)
    - » É encontrada a classe do objecto (seguindo a instância da referência)
    - » É encontrada a implementação do método Print() na classe e é executado

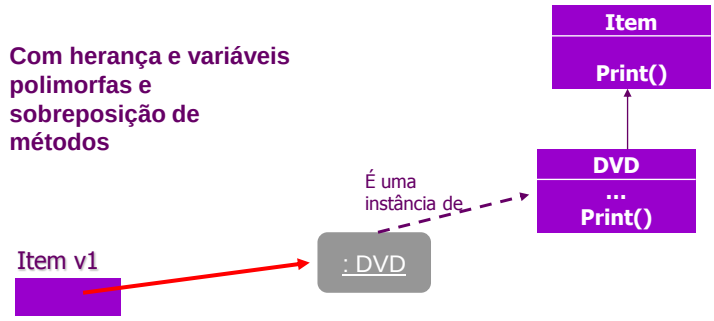
Como são chamados os métodos? (*method lookup*)



  - **Executar v1.print()**
    - » A variável v1 é acedida
    - » É encontrado o objecto armazenado nesta variável (seguindo a referência)
    - » É encontrada a classe do objecto (seguindo a instância da referência)
    - » Não é encontrado nenhum método Print na classe Video
    - » É pesquisada a superclasse para ver se tem um método Print
    - » É encontrada a implementação do método Print() na classe Item e é executado

Sobreposição, OVERRIDING

Com herança e variáveis polimorfas e sobreposição de métodos



Executar v1.print()

- A variável v1 é acedida
- É encontrado o objecto armazenado nesta variável (seguindo a referência)
- É encontrada a classe do objecto (seguindo a instância da referência)
- É encontrada a implementação do método Print() na classe e é executado

- **Solução para executar também Print da classe Item?**

Class CD

```
public void print()
{
   super.print();
   System.out.print("autor: " + artist);
}
```

Class DVD

```
public void print()
{
   super.print();
   System.out.print("realizador: " + director);
}
```

Polimorfismo de Métodos – a mesma chamada do método pode, em alturas diferentes, invocar métodos diferentes, dependendo do tipo dinâmico da variável usada na chamada