Programação Orientada a Objetos em java

Polimorfismo

Polimorfismo

- Uma característica muito importante em sistemas orientados a objetos
- Termo proveniente do grego, e significa *muitas formas*
  - Em POO, significa a possibilidade de um tipo abstrato (classe abstrata ou interface) ser utilizado sem que necessariamente programador conheça a sua implementação concreta
- Programação voltada a tipos abstratos
  - Independência de implementação

Polimorfismo

Uma Cesta de Compras

- Um elemento muito frequente nos sites de *e-commerce* capaz de guardar produtos diversos e calcular o preço deles na conclusão da compra
  - Na prática existe uma infinidade de produtos diferentes
  - Como tratá-los de forma a não ter que construir uma cesta de compras para cada produto ?
    - Existiriam centenas de tipos de cestas diferentes: livros, cd's, celulares, tv's etc.

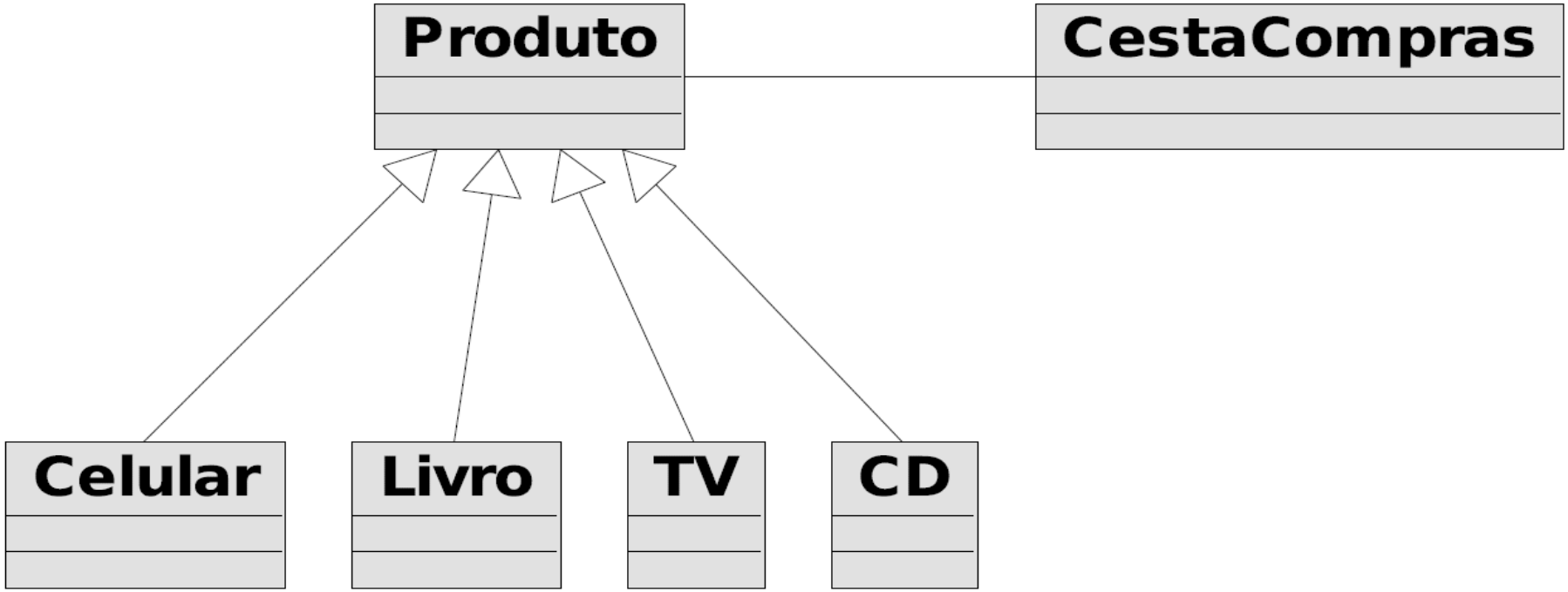
Polimorfismo

Uma Cesta de Compras

- Necessário desenvolver uma cesta única capaz de manipular todos os tipos de produtos
- **Solução:** projetar o sistema de tal forma que a cesta enxergue apenas uma interface denominada **produto**, de onde todas as implementações de produtos distintos serão feitas

Polimorfismo

Uma Cesta de Compras



Polimorfismo
Conceitos

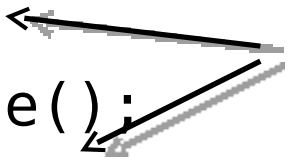
- Com o polimorfismo em mente, o programador pode projetar as classes para interagirem entre si o máximo possível apenas através de tipos abstratos
- A ação específica dependerá de cada situação concreta

Polimorfismo
Trabalhando com superclasses e interfaces

- Uma das primeiras coisas que o programador deve ter em mente para evitar a referência a classes concretas é que os atributos e parâmetros de métodos envolvidos na classe criada devem **"esquecer" os tipos concretos o máximo possível**
  - Apenas quando for inevitável, trabalha-se com tipos concretos – ao contrário, utiliza-se tipos abstratos

Referenciando Classes Concretas

```
public class Dispositivo { ... }
public class Mouse extends Dispositivo { ... }
public class Teclado extends Dispositivo { ... }

Mouse mouse = new Mouse();

Teclado teclado = new Teclado();
```

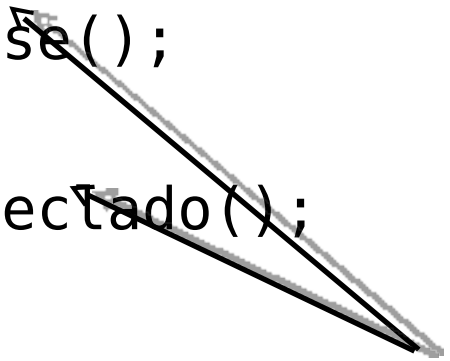


Criação de instâncias de *Teclado* e *Mouse* **referenciando as classes concretas** (lado esquerdo da atribuição de variável)

Referenciando Tipos Abstratos

- Mudança no estilo de programação para explorar o polimorfismo:

```
Dispositivo mouse = new Mouse();

Dispositivo teclado = new Teclado();
```



Em Java, a máquina virtual guarda informações internas que permitem identificar qual classe concreta é referenciada pela variável de tipo genérico!

As variáveis são do tipo *Dispositivo*, mas instanciados como *Teclado e Mouse*

Polimorfismo

Exemplo – Gerenciador de Dispositivos

- Considere um gerenciador de drivers de dispositivos para um sistema operacional:

```
// Classe gerenciadora dos drivers de um sistema operacional
public class GerenciadorDrivers {
    // Apenas referências ao tipo genérico
    private Dispositivo[] dispositivos;
    public GerenciadorDrivers() {
        // Código para ler a configuração atual do S.O.
    }
    public void inicializarDispositivos() {
        for (int i = 0; i < dispositivos.length; i++) {
                // Inicialização de cada dispositivo
        }
    }
}
```

O gerenciador considera apenas o tipo genérico *Dispositivo*

A inicialização é feita de forma genérica, sem que o gerenciador precise saber cada tipo de dispositivo

Polimorfismo

Análise do exemplo – Gerenciador de Dispositivos

- Se não fosse possível referenciar cada driver de dispositivo através de sua superclasse Dispositivo, o gerenciador desenvolvido teria de ser capaz de **lidar com várias classes de dispositivos**
- Da forma apresentada, é possível lidar com cada tipo de dispositivo, visto que ele **enxerga apenas um array de objetos da classe Dispositivo**. Neste caso, um método *inicializar()* em cada classe concreta de dispositivo poderia conter as particularidades envolvidas na inicialização de um mouse, teclado, câmera ou disco rígido externo, por exemplo.

Polimorfismo

Exemplo – Sistema de Compras

- Considere um sistema de compras com cálculo de alíquotas de impostos incidentes nos produtos:

```
public interface Imposto {
	public float getAliquota();
}
public class CPMF implements Imposto {
	public float getAliquota() {
		return 0.38;
	}
}
public class IRPF implements Imposto {
	public float getAliquota() {
		return 27.5;
	}
}
```

```
Imposto cpmf = new CPMF();
Imposto irpf = new IRPF();
```

Instanciação de objetos de impostos apenas utilizando o tipo abstrato definido pela interface Imposto

Polimorfismo

Continuação - Sistema de Compras

```
// Cesta de compras em um sistema online
public class CestaDeCompras {
      private ArrayList produtos;
      private Imposto[] impostos;
      // Adicionar e remover produtos do ArrayList
      public void adicionarProduto(Produto p) {
            produtos.add(p);
      }
      public void removerProduto(Produto p) {
            produtos.remove(p);
      }
       public float getValorTotal() {
```

```
}
```

Polimorfismo

Continuação - Sistema de Compras

```
// Obtém o valor total da compra realizada
public float getValorTotal() {
	float valorTotal = 0;
	// Varre a lista de produtos
	for (int i = 0; i < produtos.size(); i++) {
		// Obtém o produto atual
		Produto p = (Produto) produtos.get(i);
		// Valor sem os impostos
		float valorTemp = p.getValor();
		// Varre o array de impostos e soma a alíquota
		for (int j = 0; j < impostos.length; j++) {
			valorTemp += valorTemp *
					impostos[j].getAliquota();
		}
		valorTotal += valorTemp;
	}
	return valorTotal;
}
```

Polimorfismo

Análise do exemplo – Sistema de Compras

- Necessidade de se calcular o valor total da compra após a incidência das alíquotas de impostos associados
- Ao invés de ter que “saber” lidar com vários tipos de impostos, a utilização da hierarquia apresentada permite simplificar a manutenção e facilitar a adição de novos impostos ou remoção de impostos existentes, além de permitir a incidência simultânea de diferentes impostos nos produtos, de forma simplificada.

Resolução Dinâmica de Métodos

- A referência aos tipos abstratos, em vez de utilizar classes concretas, é possível devido a uma característica denominada **resolução dinâmica de métodos**
- Definição de “resolução de método”:
  - Conexão de uma chamada de método a um corpo de método.

Tipos de Resolução de Métodos

- Antecipada ou estática
  - Feita antes do programa ser executado, ou seja, estaticamente, e que normalmente é realizada pelo compilador. (Ex: linguagem C)
  - O programa compilado já tem as chamadas resolvidas previamente, sem que haja a possibilidade de uma chamada diferente
  - Exige que os tipos concretos sejam conhecidos previamente

Tipos de Resolução de Métodos

- Tardia ou dinâmica
  - Deixada a cargo do ambiente de execução do programa, que dinamicamente permite a resolução adequada a uma referência que não é necessariamente conhecida em tempo de compilação
  - O compilador, neste caso, não sabe qual o método correto a ser chamado, já que ele enxerga apenas uma chamada de método a um tipo de alto nível, que pode ser uma interface ou uma determinada superclasse.
  - Adotado na linguagem Java por padrão (exceto os métodos declarados com *final*), cuja implementação é feita de forma automática pela máquina virtual

Resolução Dinâmica de Métodos

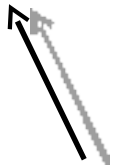
Exemplo – Jogo de Xadrez

```
class Peca { // Classe raiz
   public int x,y; // Valores atuais de posição
   public void mover(int x, int y) {
        System.out.println("Peça desconhecida");
   }
}
class Peao extends Peca {
   public void mover(int x, int y) {
        System.out.println("Movendo peão para (" + x + "," + y + ")");
   }
}
class Bispo extends Peca {
   public void mover(int x, int y) {
        System.out.println("Movendo bispo para (" + x + "," + y + ")");
   }
}
```

Resolução Dinâmica de Métodos

Continuação – Jogo de Xadrez

```
class Rainha extends Peca {
   public void mover(int x, int y) {
        System.out.println("Movendo rainha para (" + x + "," + y + ")");
   }
}

class JogoDeXadrez {
   public void moverPeao(Peao p, int x, int y)     { p.mover(x, y); }
   public void moverBispo(Bispo b, int x, int y)   { b.mover(x, y); }
   public void moverRainha(Rainha r, int x, int y) { r.mover(x, y); }
}
```

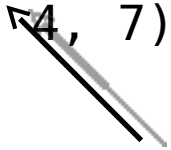


Definição de métodos mover para cada tipo de peça diferente!

Resolução Dinâmica de Métodos

Jogo de Xadrez - Executável

```
public static void main(String[] args) {
   JogoDeXadrez jx = new JogoDeXadrez();
   Peao p = new Peao();
   Bispo b = new Bispo();
   Rainha r = new Rainha();

   jx.moverPeao(p, 6, 3);
   jx.moverBispo(b, 2, 8);
   jx.moverRainha(r, 4, 7);
}
```



Necessidade da chamada de métodos diferentes para o mesmo tipo de ação!

Resolução Dinâmica de Métodos

Jogo de Xadrez – Inclusão da Peça Torre

- No caso de incluir, por exemplo, a peça torre, seria necessário também desenvolver um novo método para movê-la:

```
public void moverTorre(Torre t, int x, int y) { t.mover(x, y); }
```

Resolução Dinâmica de Métodos

Como melhorar o Jogo de Xadrez ?

- Considerando a solução anteriormente proposta, é possível observar que cada peça incluída demanda um novo método
  - E se ao invés do xadrez, que possui 6 peças previamente conhecidas, fosse um jogo de estratégia com um número desconhecido de personagens?
- Solução: eliminar a referência às classes concretas, referenciando apenas o tipo abstrato **Peça**

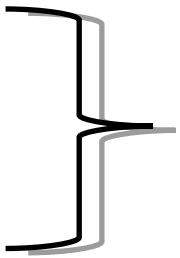
```
public void moverPeca(Peca p, int x, int y) { p.mover(x, y); }
```

Resolução Dinâmica de Métodos

Novo Formato de Chamada

- Refazendo as chamadas ao jogo de xadrez:

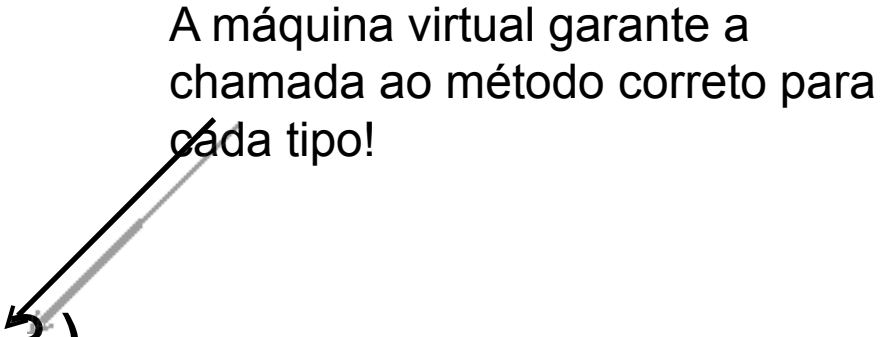
```
public static void main(String[] args) {
   JogoDeXadrez jx = new JogoDeXadrez();
   Peao p = new Peao();
   Bispo b = new Bispo();
   Rainha r = new Rainha();

   jx.moverPeca(p, 6, 3);
   jx.moverPeca(b, 2, 8);
   jx.moverPeca(r, 4, 7);
}
```



Agora é possível chamar o mesmo método para qualquer uma das peças!

Resolução Dinâmica de Métodos

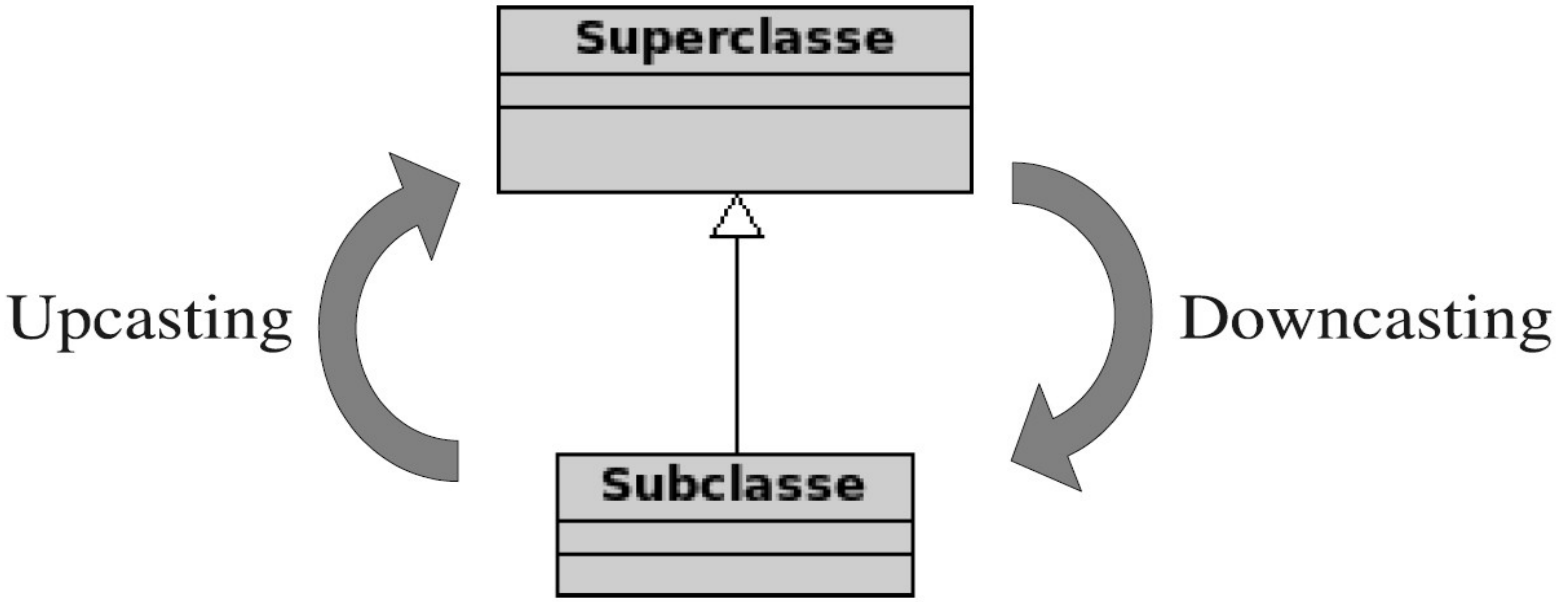
Resultado da execução – nova chamada




```
# java JogoDeXadrez
Movendo peão para (6,3)
Movendo bispo para (2,8)
Movendo rainha para (4,7)
```

Casting

- Relembre: uma classe, ao herdar de outra, **assume o tipo desta** onde quer que seja necessário
- Existem dois tipos de coerção entre tipos, que também é chamada de **casting**

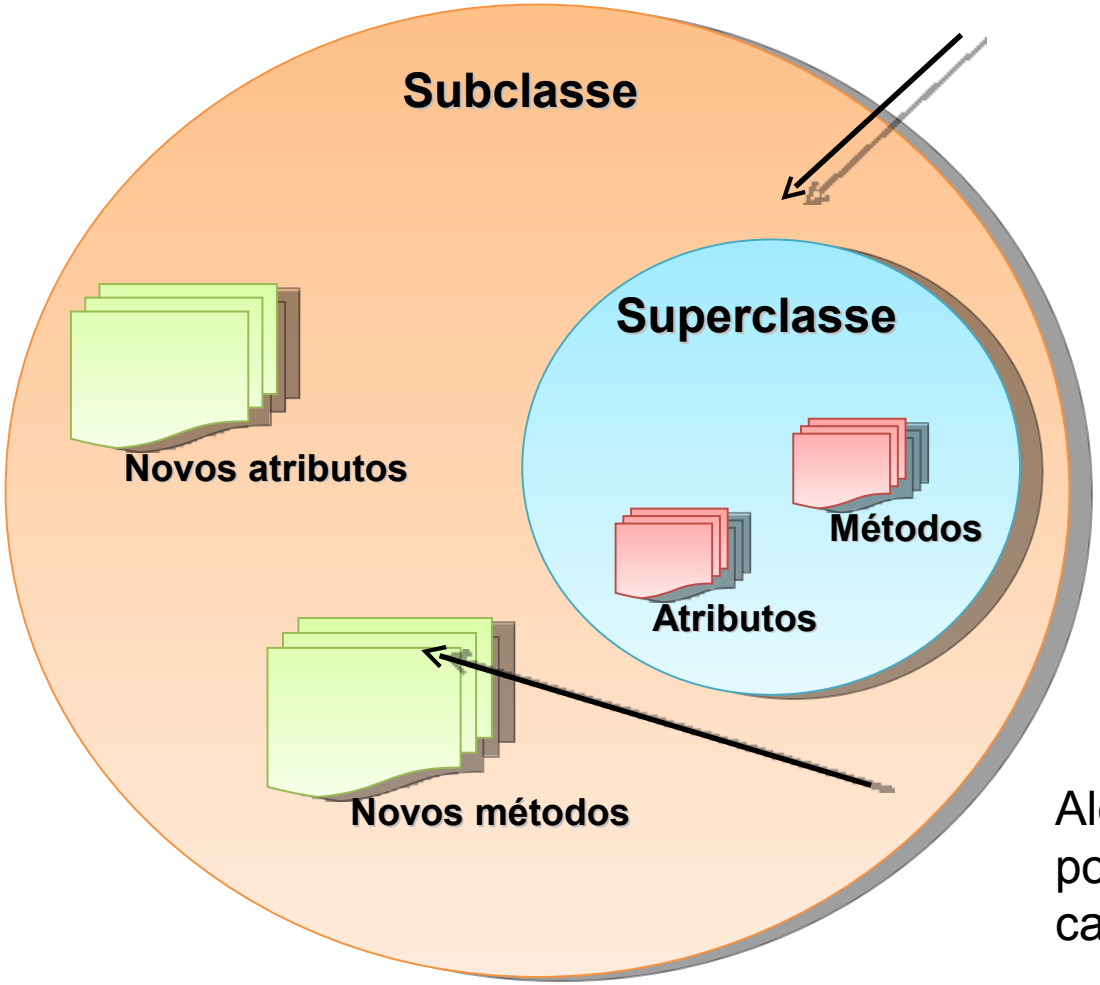


Upcasting

- Este tipo de coerção acontece de baixo para cima da hierarquia, no sentido das subclasses para as superclasses
  - Não há necessidade de nenhuma indicação explícita para realizar *upcasting*
  - Observe que a classe derivada sempre vai manter as características públicas da sua superclasse

Upcasting

Visão através de conjuntos

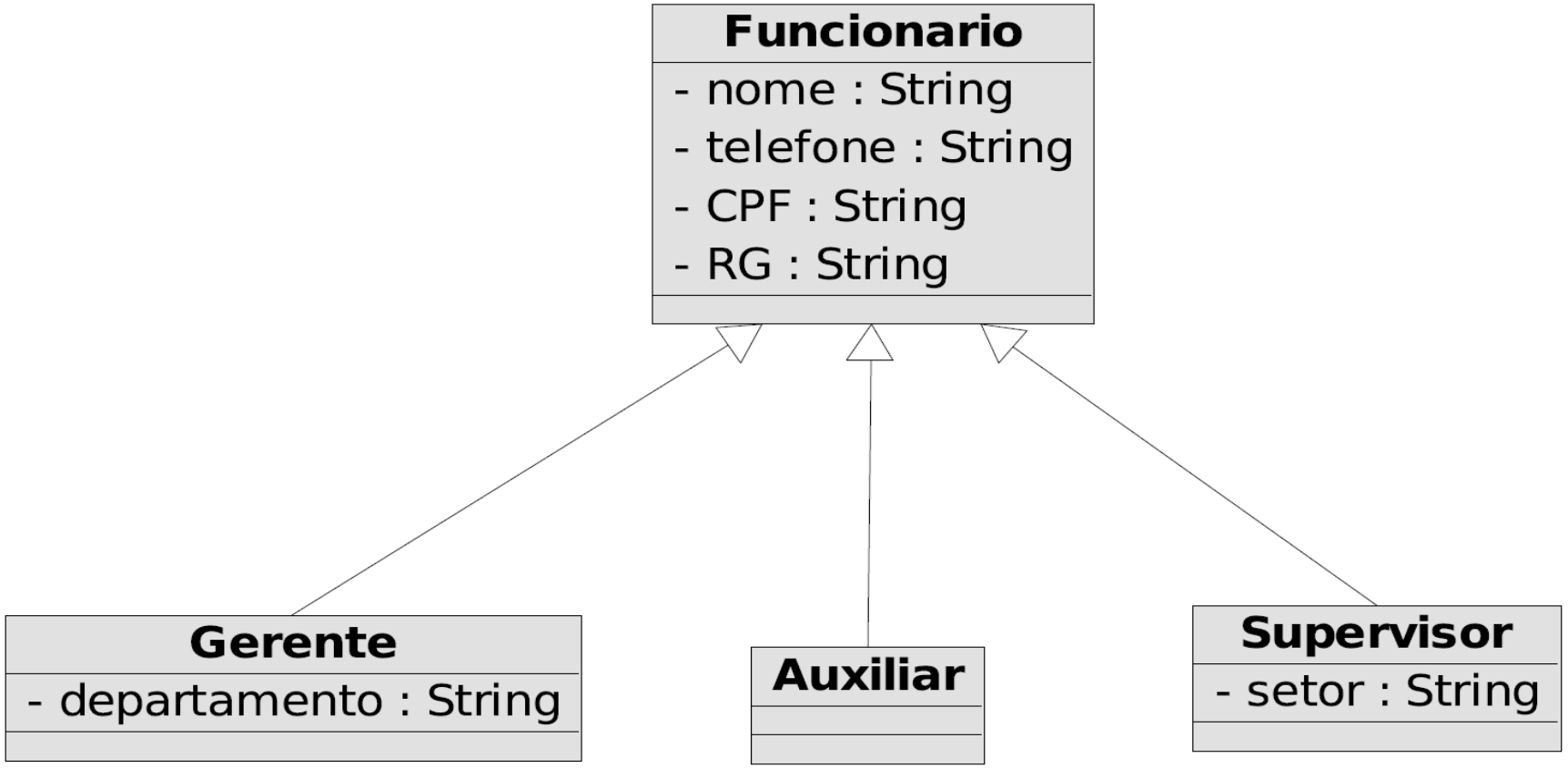


É possível observar que a subclasse conserva a estrutura da superclasse, e por isso o **upcasting** é feito de forma automática

Além da estrutura da superclasse, podem ser definidas novas características

Upcasting

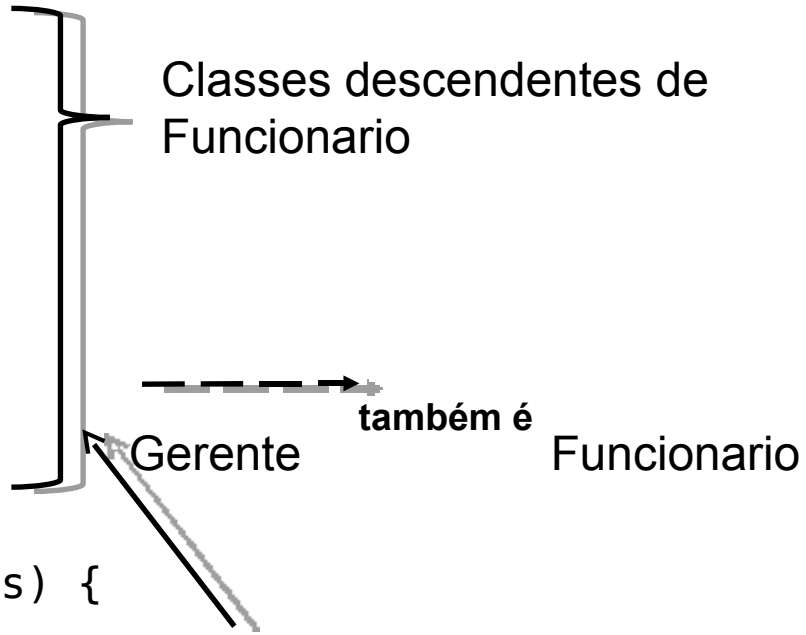
Exemplo – Sistema de RH



Upcasting

Continuação – Sistema de RH

```
class Funcionario {
   protected String CPF, RG, telefone, nome;
}
class Gerente extends Funcionario {
   private String departamento;
}
class Supervisor extends Funcionario {
   private String setor;
}
class Auxiliar extends Funcionario {
}
public class TesteUpcasting {
   public static void main(String[] args) {
        Gerente ger = new Gerente();
        Supervisor sup = new Supervisor();
        Funcionario func = ger;
   }
}
```



func é do tipo Funcionario, mas recebe uma instância do tipo Gerente - **UPCASTING**

Downcasting

- Este tipo de coerção acontece de cima para baixo na hierarquia, ou seja, no sentido das superclasses para as subclasses
  - Não é feito de forma automática
  - É necessário explicitar o downcasting através de parênteses antes do nome da variável que indiquem o tipo desejado a ser convertido

Downcasting
Exemplo

```
// Um exemplo de Downcasting
public class TesteDowncasting {
   public static void main(String[] args) {
       Gerente ger = new Gerente();
       Supervisor sup = new Supervisor();
       Funcionario func = ger;

       Gerente ger_2 = (Gerente) func;
   }
}
```

Aqui existe um Upcasting - Feito automaticamente

Um novo objeto do tipo Gerente recebe a referência a um objeto do tipo Funcionario. **O downcasting necessita de conversão explícita**

Downcasting
ClassCastException

- A coerção via *downcasting* não é automática porquê nem sempre uma superclasse pode assumir o tipo da subclasse
  - Ex: no exemplo anterior do sistema de RH **- todo auxiliar é funcionário, mas nem todo funcionário é auxiliar (pode ser gerente ou supervisor)**
- Caso a conversão com *downcasting* não seja possível, uma exceção do tipo *java.lang.ClassCastException* será lançada

ClassCastException

Exemplo

```
// Um exemplo de Downcasting
public class TesteDowncasting {
   public static void main(String[] args) {
        Gerente ger = new Gerente();
        Supervisor sup = new Supervisor();
        Funcionario func = ger;

        Supervisor sup_2 = (Supervisor) func;
   }
}
```

Este *downcasting* vai gerar uma exceção em tempo de execução

Saída do programa:

```
#java br.teste.TesteDowncasting
Exception in thread "main" java.lang.ClassCastException:
br.teste.Gerente cannot be cast to br.teste.Supervisor
at br.teste.TesteDowncasting.main(TesteDowncasting.java:11)
```

Identificação de Tipos

- O polimorfismo estimula a referência a tipos genéricos, que deve ser usada sempre que possível
  - Há situações, entretanto, que existe a necessidade de se saber **qual o tipo concreto** por trás da referência genérica
- Esta identificação é chamada de Identificação de Tipos em Tempo de Execução (do inglês RTTI – *Runtime Type Identification*)

Identificação de Tipos
Formas de Identificação

- Java permite o acesso às informações em tempo de execução através de duas maneiras:
  - Através da palavra-chave **instanceof**, disponibilizada pela linguagem Java especificamente para a identificação de tipos em tempo de execução.
  - Através de **reflexão**, que é uma forma de objetos terem acesso à sua própria estrutura interna (métodos, atributos, nome de classe, superclasse, etc.) e que também pode ser utilizada para identificar tipos em tempo de execução.

Identificação de Tipos
Exemplo – Jogo de Xadrez

- Relembrando alguns movimentos do Xadrez:

Peça	**Movimento**
Peão	- Uma casa na direção vertical (apenas quando não há peça adversária no caminho) - Uma casa na direção diagonal (apenas quando há peça adversária, que é consumida)
Torre	- Número ilimitado de casas - Apenas nas direções horizontal e vertical (não é permitida a diagonal)
Bispo	- Número ilimitado de casas - Apenas na diagonal (não são permitidas as direções horizontal e vertical)
Rainha	- Número ilimitado de casas - Movimentos horizontal, vertical e diagonal
Rei	- Apenas uma casa - Movimentos horizontal, vertical e diagonal

Identificação de Tipos
Continuação – Jogo de Xadrez

- Para caracterizar um mecanismo mais realista no jogo de xadrez, é necessário construir uma classe capaz de validar os movimentos das peças:
  - Ex:
    - Uma torre não pode andar na diagonal
    - Um bispo não pode andar na vertical e horizontal
    - Um peão não pode andar múltiplas casas
- Esta classe será denominada **Tabuleiro**

Identificação de Tipos
Continuação – Jogo de Xadrez

Método mover genérico

```
class JogoDeXadrez {
   public void mover(Peca p, int x, int y) {
       if (Tabuleiro.analisarJogada(p, x, y) {
          // Jogada válida!
       }
       else
           System.out.println("Jogada inválida! Tente" +
                              "novamente.");
   }
}
class Tabuleiro {
    public static boolean analisarJogada(Peca b, int x, int y) {

    }
}
```

Chama a validação do tabuleiro

Para analisar se a jogada foi válida é necessário saber qual é o tipo da peça (torre, bispo, rainha, peão, rei, etc.)

Identificação de Tipos
Continuação – Jogo de Xadrez

- A classe **Tabuleiro**
  - Para que possa validar o movimento das peças, uma solução é que a classe *Tabuleiro* **tenha conhecimento** de qual a classe concreta da peça em questão no ato de validação do movimento
- Através de um dos métodos anteriormente citados (*instanceof* ou reflexão), a classe será capaz de, baseada na classe da peça, verificar se a jogada é válida

Identificação de Tipos
Instanceof

- A palavra-chave *instanceof* permite identificar qual o tipo concreto de um objeto (mesmo que ele seja referenciado por uma superclasse ou interface)
  - Sintaxe:

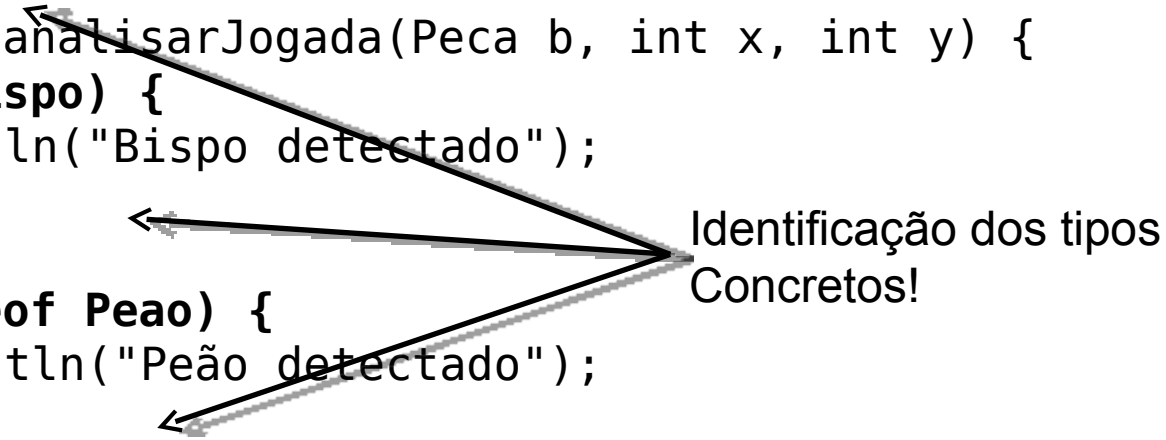
```
[Nome do Objeto] instanceof [Nome da Classe]
```

- O retorno da expressão com *instanceof* é um booleano
  - True caso o objeto seja do tipo em questão
  - False do contrário

Identificação de Tipos

Tabuleiro com instanceof

```
class Tabuleiro {
   public static boolean analisarJogada(Peca b, int x, int y) {
      if (b instanceof Bispo) {
         System.out.println("Bispo detectado");
         return true;
      }
      else if (b instanceof Peao) {
          System.out.println("Peão detectado");
          return true;
      }
      else if (b instanceof Rainha) {
          System.out.println("Rainha detectada");
          return true;
      }
      else {
           System.out.println("Erro! Peça desconhecida!");
           return false;
      }
   }
}
```



Identificação de Tipos

Outro exemplo de uso de instanceof

```
public class TesteInstanceOf {
    public static void main(String[] args) {
      Funcionario func_1 = new Auxiliar();
      Funcionario func_2 = new Supervisor();
      Funcionario func_3 = new Gerente();
      if (func_1 instanceof Auxiliar)
          System.out.println("Func_1 -> Auxiliar");
      if (func_2 instanceof Gerente)
          System.out.println("Func_2 -> Gerente");
      if (func_3 instanceof Gerente)
          System.out.println("Func_3 -> Gerente");
      func_1 = new Supervisor();
      if (func_1 instanceof Supervisor)
          System.out.println("Func_1 -> Supervisor");
      }
}
```

Identificação dos tipos Concretos!

Falso!

Func_1 agora é associado a um objeto do tipo Supervisor

Identificação de Tipos

instanceof - Saída do programa

```
# java br.teste.TesteInstanceOf
Func_1 -> Auxiliar
Func_3 -> Gerente
Func_1 -> Supervisor
```

Identificação de Tipos
Reflexão

- Termo utilizado para descrever um conjunto de facilidades que algumas linguagens de POO oferecem para que as classes possam obter informações também sobre outras classes
  - Ex: quais os atributos de determinada classe, ou o nome dela, ou os métodos públicos, superclasses, etc.

- A reflexão pode ser utilizada para várias finalidades
  - Execução de métodos, modificação da estrutura de uma classe, criação de objetos dinamicamente, etc.
  - No contexto da programação OO será abordado o aspecto de **identificação de tipos**

Identificação de Tipos
Reflexão – o método getClass()

- Existe em Java uma classe denominada *Class* que representa uma classe Java, com informações tais como nome, atributos e métodos
- Qualquer objeto em Java possui um método denominado *getClass()* que retorna um objeto representando sua classe, do tipo *Class*.
  - A partir deste objeto, é possível obter informações sobre a classe em questão, como a própria identificação do tipo concreto

Identificação de Tipos

Reflexão – exemplo

```
public class TesteGetClass {
   public static void main(String[] args) {
      Funcionario func_1 = new Auxiliar();
      Funcionario func_2 = new Supervisor();

      Class classeFunc_1 = func_1.getClass();
      Class classeFunc_2 = func_2.getClass();

      System.out.println("Classe de Func_1: " +
                                 classeFunc_1.getName());
      System.out.println("Classe de Func_2: " +
                               classeFunc_2.getName());
   }
}
```

Obtém o objeto "Class"

[Objeto do tipo Class].getName()
obtém o nome da classe

Identificação de Tipos

Reflexão - Saída do programa

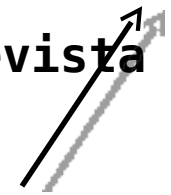
```
# java br.teste.TesteGetClass
Classe de Func_1: br.teste.Auxiliar
Classe de Func_2: br.teste.Supervisor
```

Reflexão
O atributo estático class

- Além do método getClass(), utilizado para extrair a estrutura de uma classe de um objeto, existe um atributo estático em cada classe ou interface, denominado *class*, que pode ser usado para comparações entre tipos. A sintaxe para acessá-lo é a mesma de um atributo estático qualquer:

```
Class [objetoClass] = [nome da classe].class
```

Ex: `Class classeRevista = Revista.class;`



Observe que não foi necessária a criação de um objeto da classe Revista – a própria classe já fornece seu tipo através do atributo estático *class*

Identificação de Tipos

Tabuleiro com reflexão

```
class Tabuleiro {
   public static boolean analisarJogada(Peca b, int x, int y) {
      if (b.getClass().equals(Bispo.class) {
         System.out.println("Bispo detectado");
         return true;
      }
      else if (b.getClass().equals(Peao.class) {
          System.out.println("Peão detectado");
          return true;
      }
      else if (b.getClass().equals(Rainha.class) {
          System.out.println("Rainha detectada");
          return true;
      }
      else {
           System.out.println("Erro! Peça desconhecida!");
           return false;
      }
   }
}
```