

# Programação Orientada por Objetos

Herança de Classes

Polimorfismo

Classes Abstratas

(Livro *Big Java, Late Objects* – Capítulo 9)

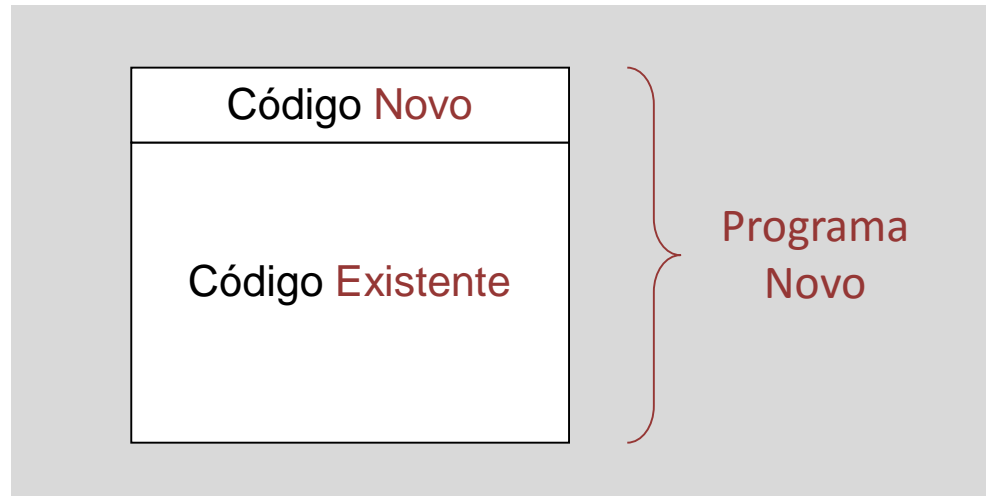
- [Mecanismos de Reutilização de Código](#)
- [Herança de Classes](#)
- [Polimorfismo](#)
- [Classes Abstratas](#)
- [Classe Object](#)

- [Mecanismos de Reutilização de Código](#)
- [Herança de Classes](#)
- [Polimorfismo](#)
- [Classes Abstratas](#)
- [Classe Object](#)



- [Mecanismos de Reutilização de Código](#)
  - [Interesse](#)
  - [Tipos de Mecanismos](#)

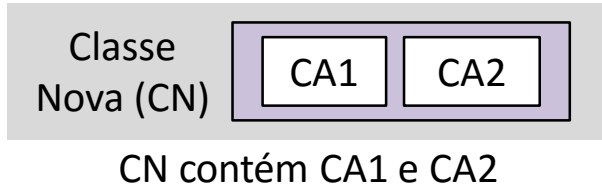
- Redução do esforço de programação  $\Rightarrow$  redução dos custos de produção de software
  - Uma das **vantagens** da POO
  - Como?
    - Programa **novo** obtido programando
      - Não todo o programa
      - Apenas uma pequena **parte nova** sobre código existente (reutilização)



- Concretamente
  - Baseada na construção de **classes novas** a partir de **classes existentes**

## Composição

- Permite criar uma classe **composta** por outras classes (CA)

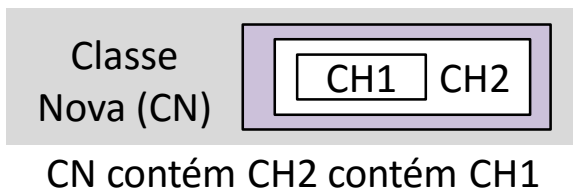


- Classes **agregadas** (CA) estão ao **mesmo nível**

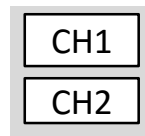


## Herança

- Permite criar uma classe que **herda** (**acrescenta**) outras classes



- Classes **herdadas** (CH) estão em **níveis diferentes**



- [Mecanismos de Reutilização de Código](#)
- [Herança de Classes](#)
- [Polimorfismo](#)
- [Classes Abstratas](#)
- [Classe Object](#)

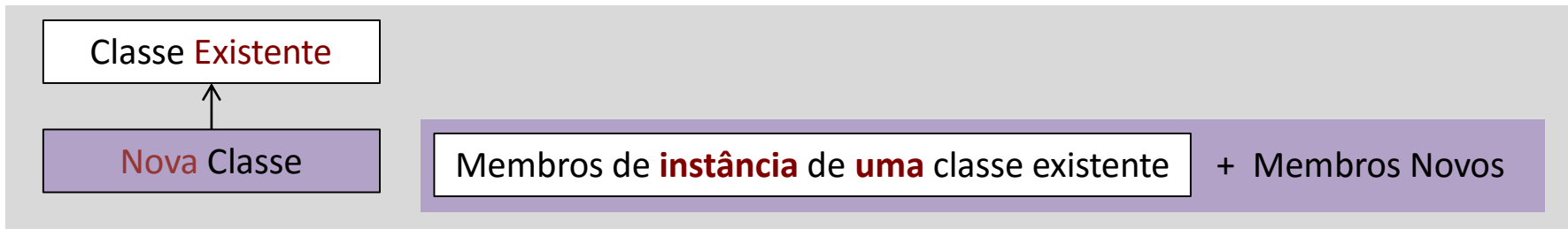


- Noções Básicas
  - [Noção de Herança de Classes](#)
  - [Superclasse e Subclasse de uma Classe](#)
  - [Relacionamento É-UM](#)
  - [Hierarquia de Classes](#)
    - [Hierarquia de Especialização](#)
  - [Herança Transitiva](#)
  - [Tipos de Herança em POO](#)
    - Simples
    - Múltipla
- Java
  - [Hierarquia Simples](#)
  - [Classe Object](#)
  - [Declaração de uma Subclasse](#)
  - [Tipos de Membros de uma Subclasse](#)
    - [Membros Herdados](#)
      - Acessíveis
      - Inacessíveis
    - [Membros Locais](#)
      - Definidos
      - Redefinidos (ou Reescritos)
- Java (continuação)
  - [Referência super](#)
    - Acesso a Membros Reescritos
    - Distinguir Métodos Locais de Métodos Herdados
  - [Invocação super\(\)](#)
    - Acesso a Construtores da Superclasse

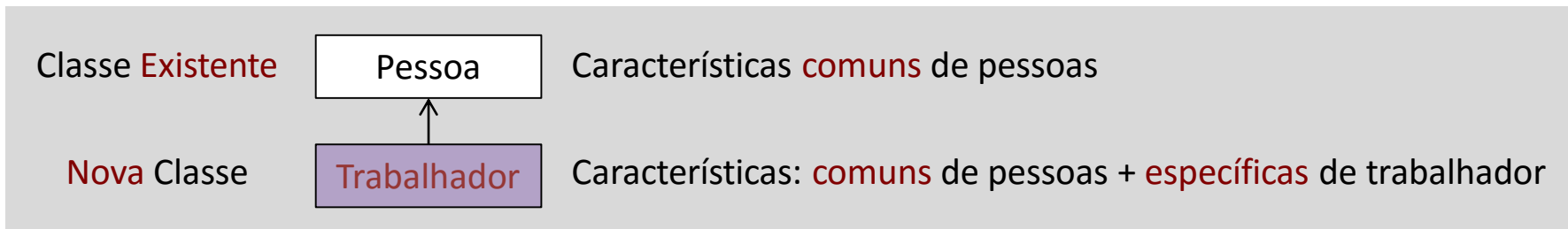


## ▪ Mecanismo

- Permite **criar** uma nova classe **por herança** (ou **extensão**) de uma classe existente // 1 em **Java**
  - Nova classe
    - **Herda** membros de instância de uma classe existente
      - **Variáveis** de Instância
      - **Métodos** de Instância
    - **Acrescenta** novos membros particulares

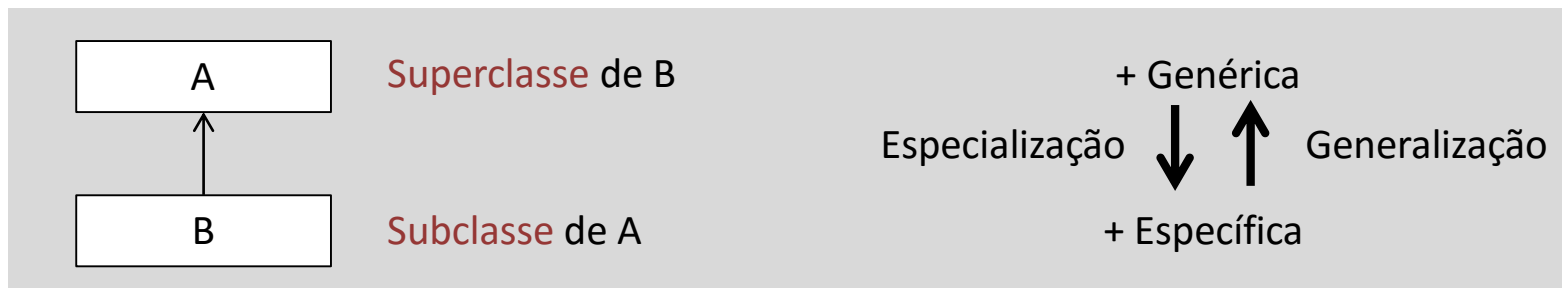


## ▪ Exemplo



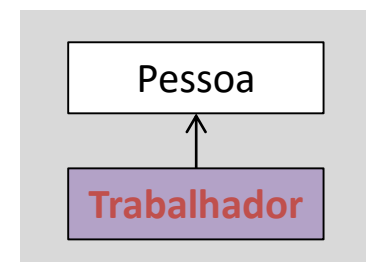
- Exemplos de características
  - **Comuns** de pessoas:      nome e data de nascimento
  - **Específicas** de trabalhador:      salário e empresa

## ▪ Exemplo



## ▪ Subclasse da classe A

- Classe que **herda** a classe A // classe **derivada** de A
- Mais **específica** do que A
  - i.e., é uma especialização de A // contém **mais** membros: métodos e/ou variáveis
- Exemplo
  - Classe Trabalhador **herda** classe Pessoa



## ▪ Superclasse da classe B

- Classe **herdada** pela classe B // contém **menos** membros: métodos e/ou variáveis
- Mais **genérica** do que B
- Exemplo
  - Classe Pessoa é **herdada** pela classe Trabalhador

- Tipo de Relacionamento entre

- Superclasse
- Suas subclasses

- Exemplo

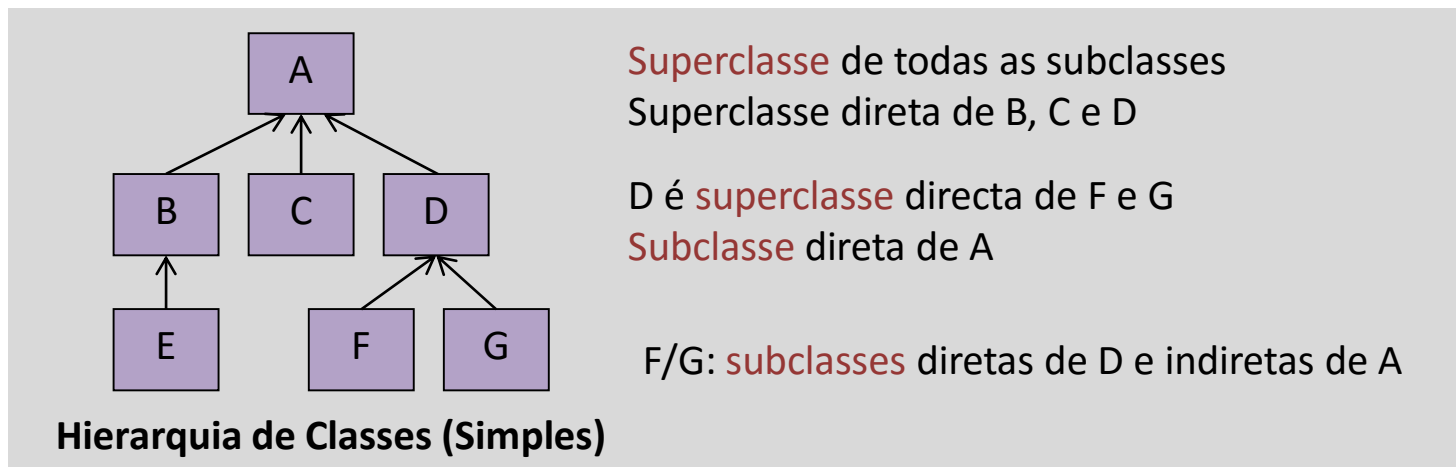
- Trabalhador é uma Pessoa



- Interesse

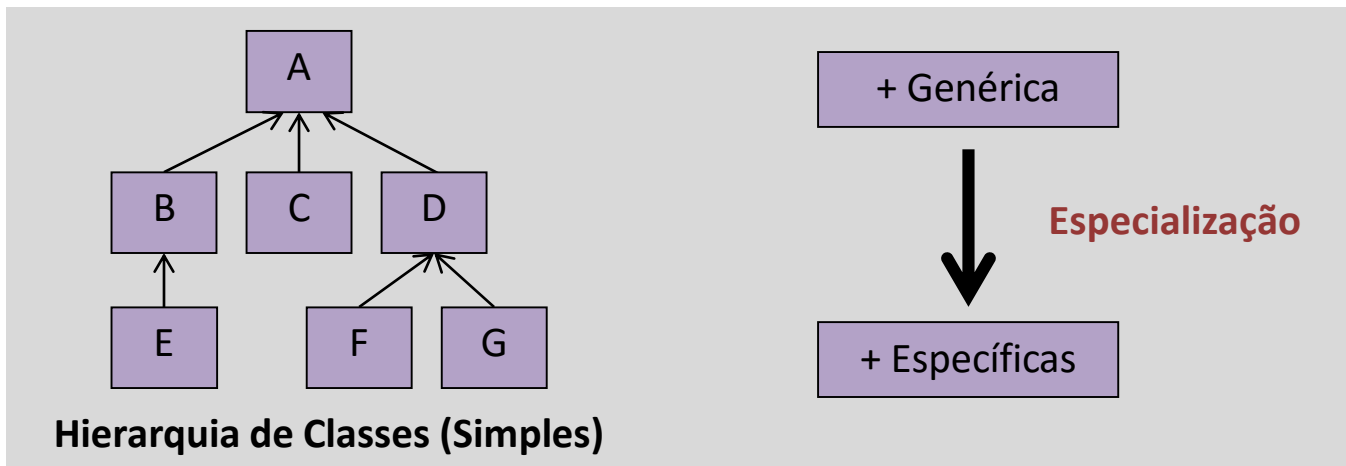
- Determinar o uso do mecanismo de herança

- **Noção**
  - Estrutura de classes organizadas em diferentes níveis
  - Construção permitida pelo mecanismo da herança
- **Classes**
  - Níveis inferiores: subclasses
  - Níveis superiores: superclasses
- **Superclasses e Subclasses podem ser**
  - Diretas // nível imediato
  - Indiretas



- Hierarquia de Especialização de Classes

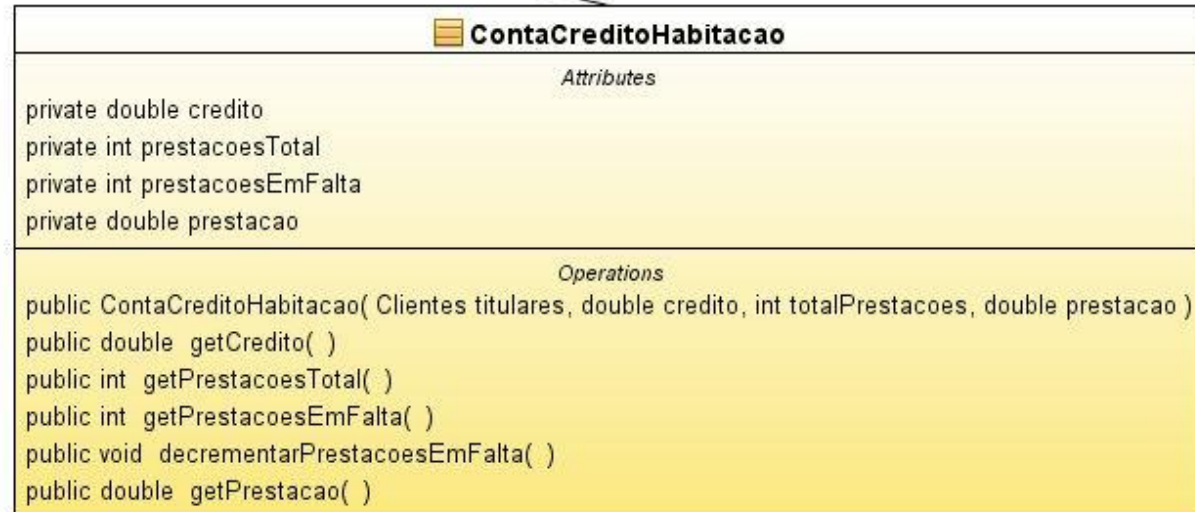
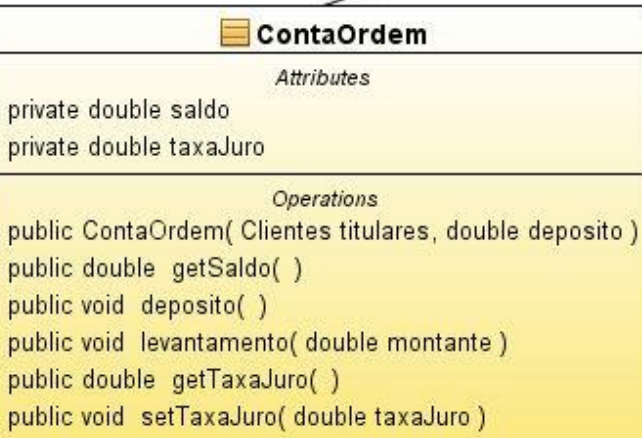
- Qualquer subclasse **é uma especialização** das suas superclasses
  - Topo da hierarquia: classe **mais genérica** // membros **comuns** a todas as subclasses
  - Base da hierarquia: classes **mais específicas** // membros **particulares**
- Sentido Topo → Base
  - Há especialização de classes



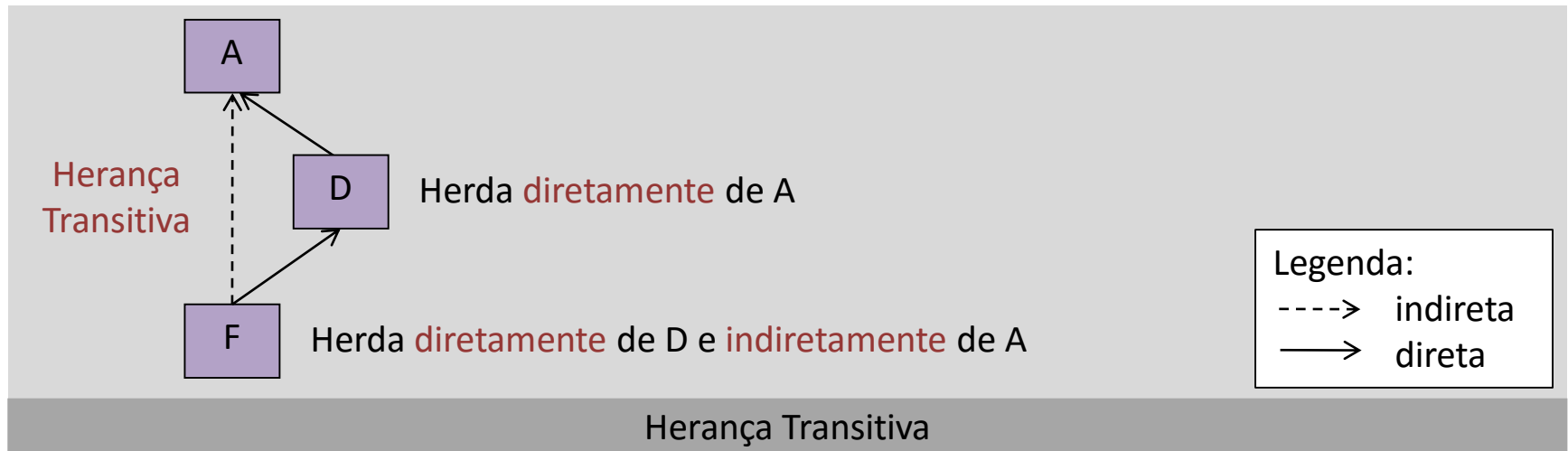
- **Exemplo** de uma **hierarquia de classes** de uma aplicação sobre **contas bancárias**
  - As subclasses são especializações da superclasse Conta (classe genérica)
    - Superclasse Conta: possui variáveis e métodos **comuns** a todos os tipos de contas
    - Subclasses: **acrescentam** variáveis e métodos específicos



Declaração de atributo **sublinhado** significa atributo de classe (**static**)



- **Noção**
  - Herança **acima** da superclasse direta
- **Uma classe herda membros de instância**
  - Da sua superclasse
    - Esta, por sua vez, herda da sua superclasse
      - E, assim sucessivamente, até ao topo da hierarquia
- **Exemplo**
  - F herda de A, porque herda de D e esta, por sua vez, herda de A

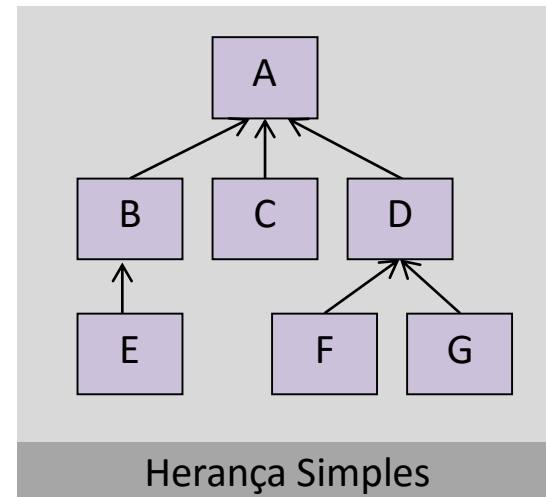


- Tipos de Herança em POO (Programação Orientada por Objetos )

- Simples
- Múltipla

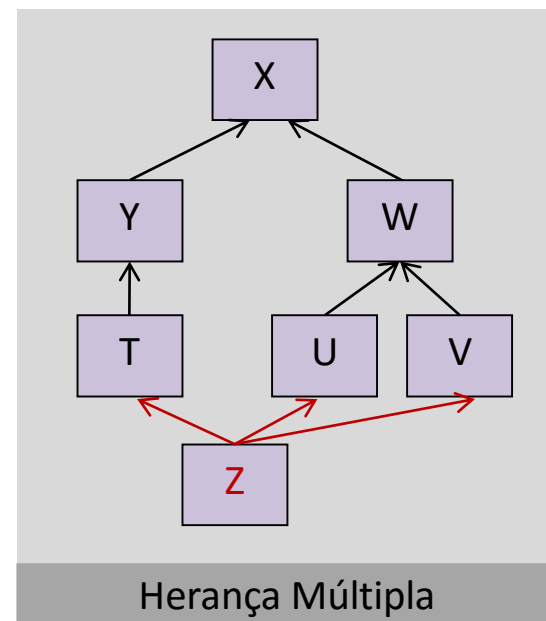
- Herança Simples

- Uma classe **apenas** pode herdar **diretamente** de outra classe
- Java
  - Única permitida em hierarquias de classes



- Herança Múltipla

- Uma classe pode herdar **diretamente** de múltiplas classes
- Exemplo
  - Classe Z
- Permitida pelo C++
- Java
  - Não permite em hierarquias **de classes**
  - Permite em hierarquias **de interfaces**





### Classe Java mais Genérica

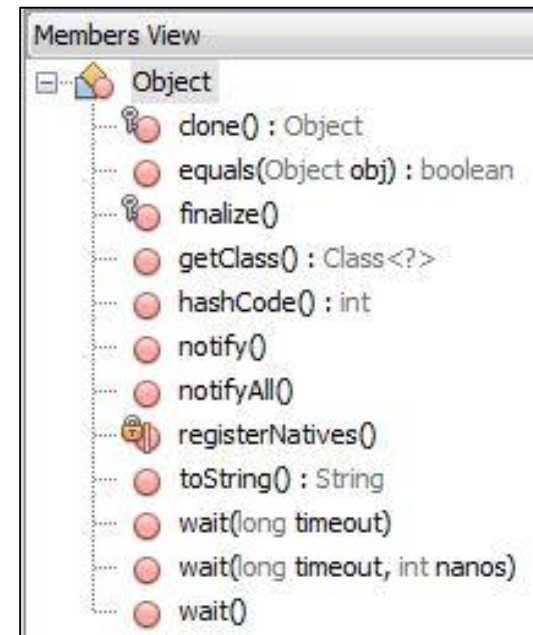
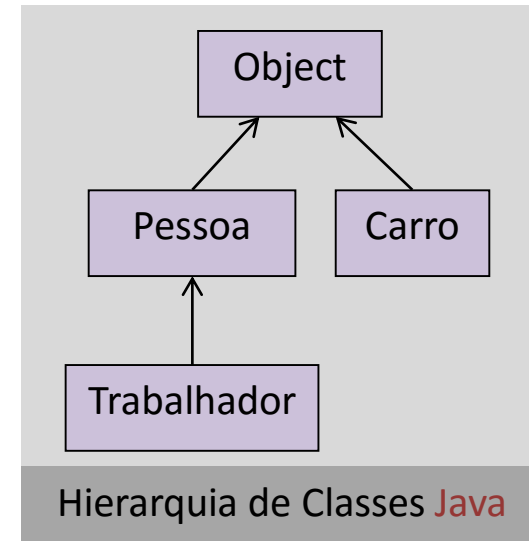
- Está no **topo** de todas as hierarquias de classes **Java**
- Superclasse de **todas** as classes
  - Qualquer classe em Java é subclasse, **pelo menos**, de Object

### Membros

- Comuns a todas as classes Java
  - Interessam a todas as classes
- Exemplos de Métodos
  - toString
  - equals

### Nome Object

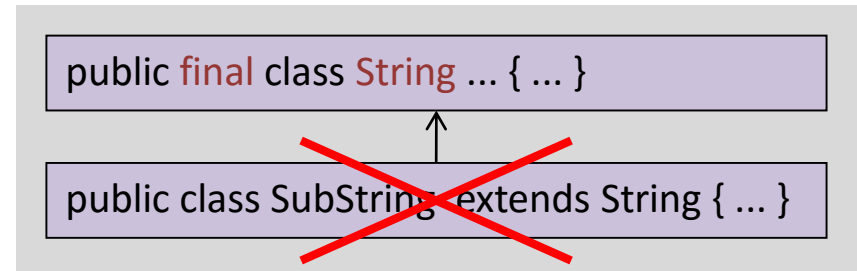
- Objeto ... é característica comum a todas as classes



## ▪ Sintaxe

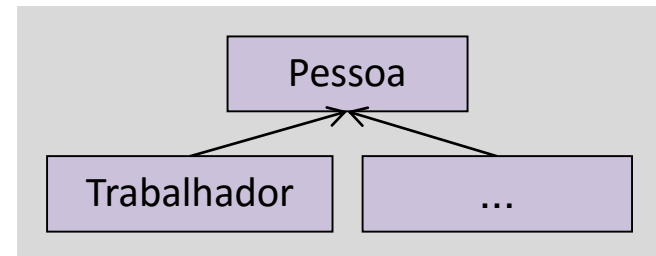
```
[modificador de acesso] [final] class subclasse extends superclasse { ... }
```

- [...] = opcional
- modificador de acesso
  - public
  - private
  - protected
  - sem modificador = package
- final
  - Classe não pode ser herdada
  - É considerada classe completa
    - Não há especializações
  - Exemplo
    - Classe `String`



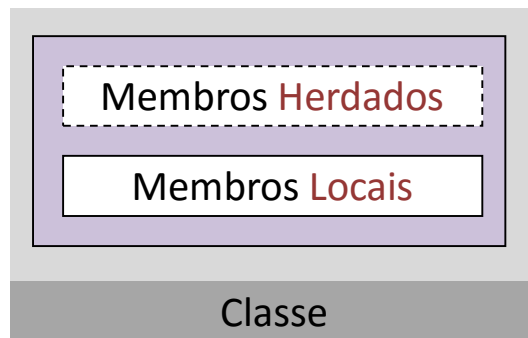
## ▪ Exemplo

```
public class Trabalhador extends Pessoa { ... }
```



- **Em Java**

- Qualquer classe herda código
  - **Pelo menos** da classe Object
- Então, relativamente à Herança, **qualquer** classe tem dois **tipos de membros**
  - Herdados
  - Locais



- **Membros Herdados**

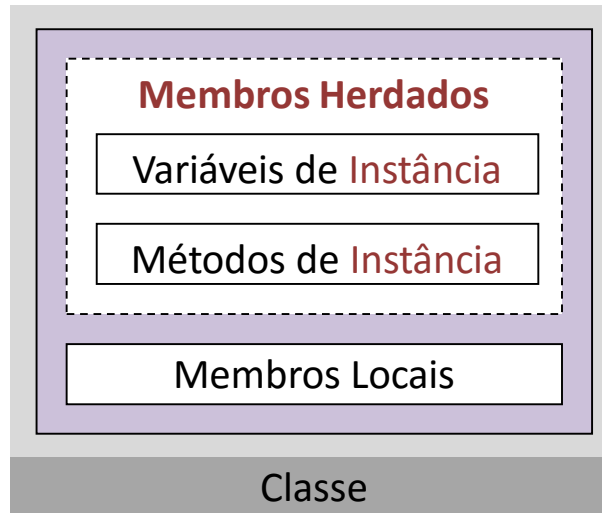
- Definidos nas suas superclasses

- **Membros Locais**

- Definidos na própria classe

- **Definição**

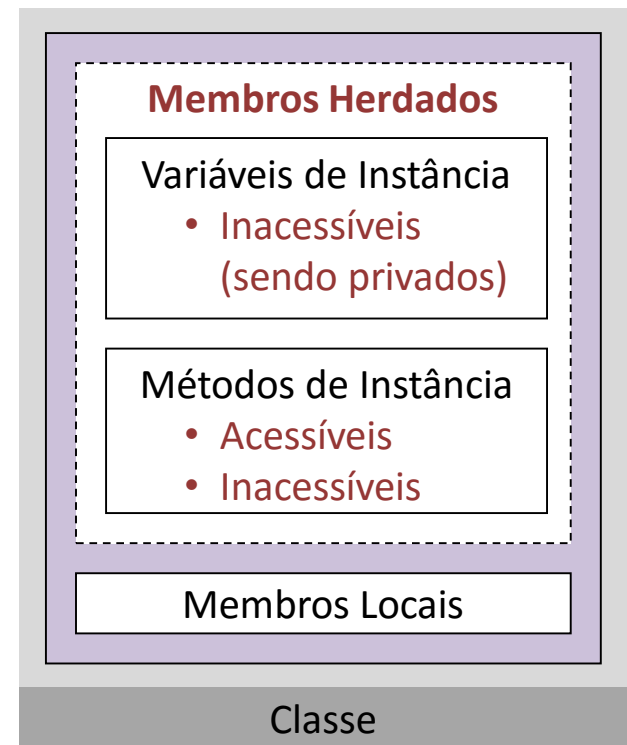
- Membros de **instância** definidos nas superclasses de uma classe (até à classe Object)
  - **Apenas**, variáveis e métodos de **instância**



- **Membros não Herdados**

- Membros de **classe** // variáveis e métodos de classe
  - Mas podem estar acessíveis // p. ex., os métodos de classe públicos
- **Construtores**
  - Porque **não** são **membros** de uma classe
  - Mas podem estar acessíveis
    - Através da invocação **super()** // abordada mais adiante

- **Relativamente ao acesso, podem ser**
  - Acessíveis
  - Inacessíveis
- **Acesso depende (em Java)**
  - Package da sua classe
  - Nível de acesso (visibilidade)
    - `public` // acessível
    - `protected` // acessível
    - `package` (sem modificador) // acessível se classe pertencer à package da superclasse (direta)
    - `private` // inacessível



- **Conceitos diferentes: herdar e aceder**
  - Um membro pode ser herdado ..... mas pode não estar acessível à subclasse
    - Exemplo: método de **instância** privado
  - Um membro pode estar acessível ... e não ser herdado
    - Exemplo: membros de **classe** públicos
- **Para simplificar a programação**
  - Podemos considerar **herdado** o que **está acessível**



## Definição

- Membros especificados na **própria** classe
  - Variáveis
    - Instância
    - Classe
  - Métodos
    - Instância
    - Classe

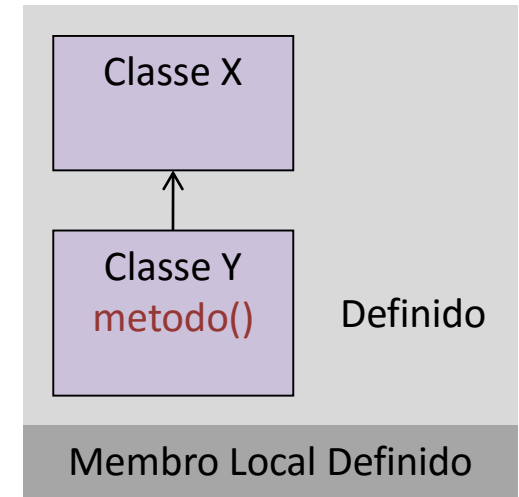
## Relativamente à definição, podem ser

- Definidos
- Redefinidos (*Override* ou Reescritos)



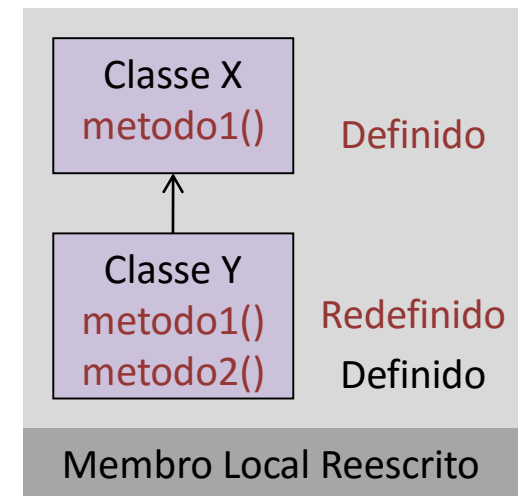
### ■ Membros Locais Definidos

- Cujas definições **residem apenas** na própria classe
  - Definições **inexistentes** nas suas superclasses



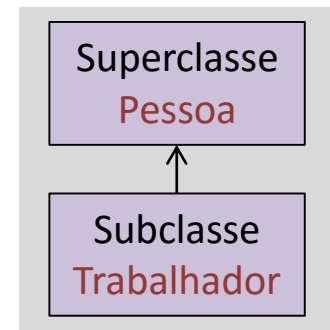
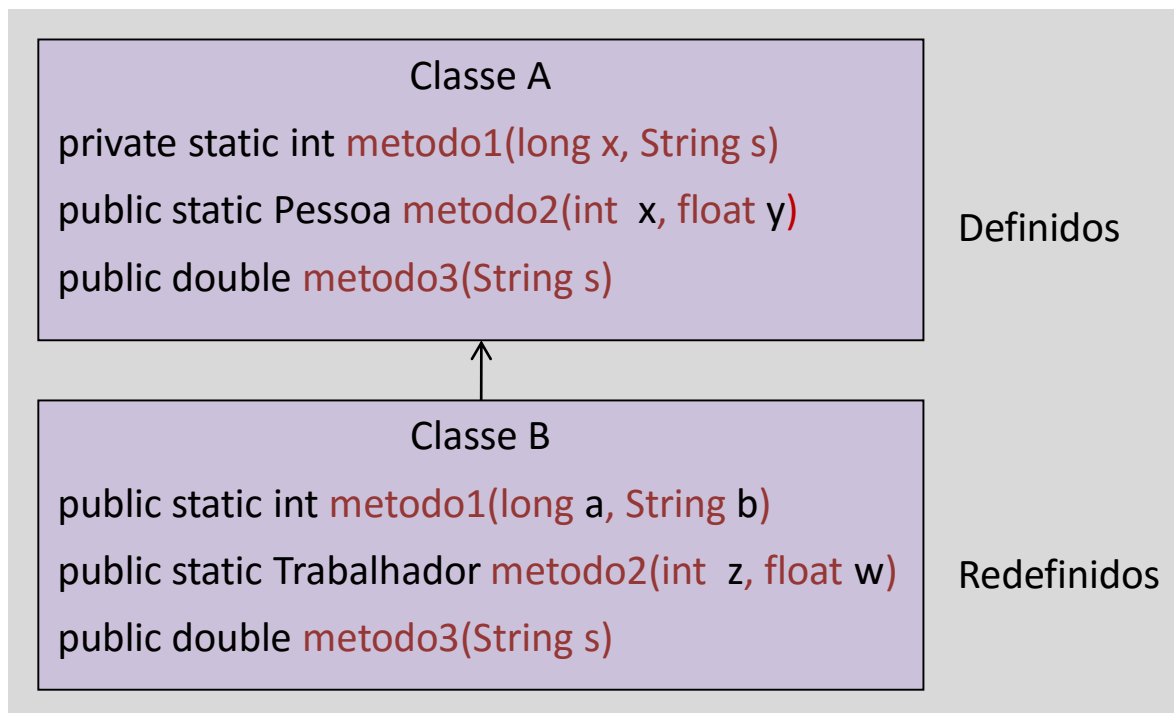
### ■ Membros Locais Redefinidos

- São **redefinições** de membros das superclasses
- Interesse
  - Quando os membros de superclasses **não** são **apropriados** para uma subclasse
  - Exemplo
    - Método toString() herdado, pelo menos, da classe Object
- Suportado pelo Mecanismo da Sobreposição (**Overriding**)
- Mais pormenores nos próximos slides

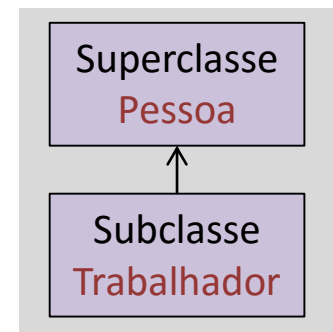
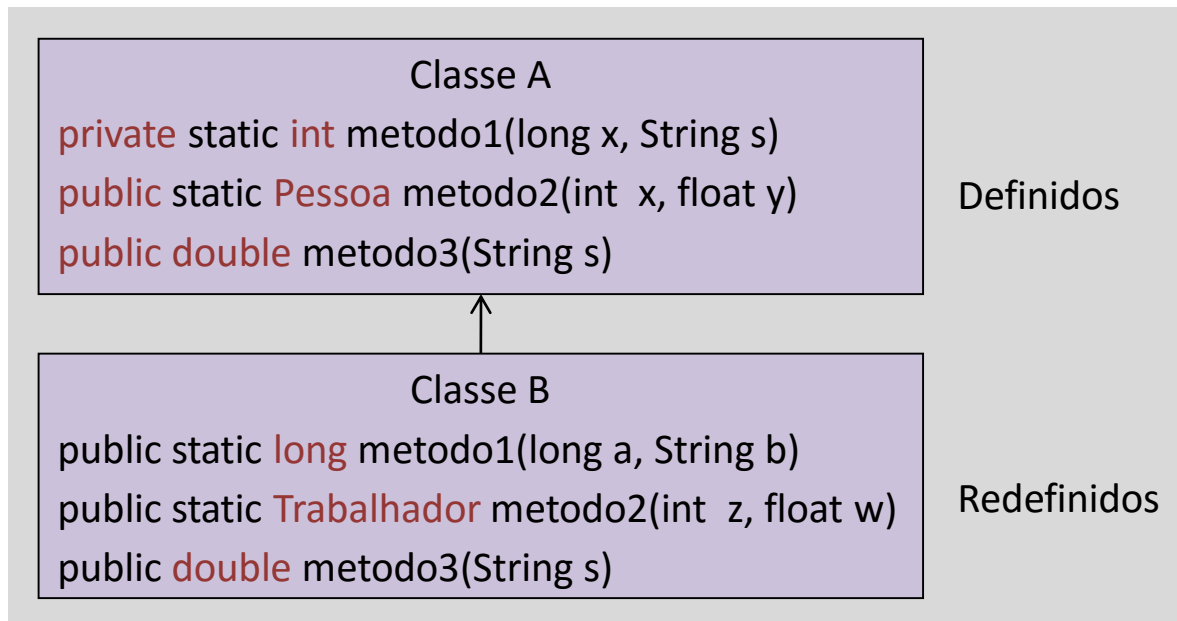




- Requisitos de Método Redefinido (1/5)
  - Assinatura
    - Igual ao do método **definido**
      - i.e, mesmo nome e mesma lista de **tipos** de parâmetros
  - Exemplo



- Requisitos de Método Redefinido (2/5)
  - Tipo de retorno
    - Depende da visibilidade (nível de acesso) do método **definido**
      - Inacessível
        - Pode ser qualquer tipo
      - Acessível
        - Desde **Java 5**, **também** pode ser qualquer **subtipo** (subclasse) do tipo de retorno da definição do método
        - Tipos de retorno **primitivos**, definido e redefinido, têm de ser iguais
  - Exemplos

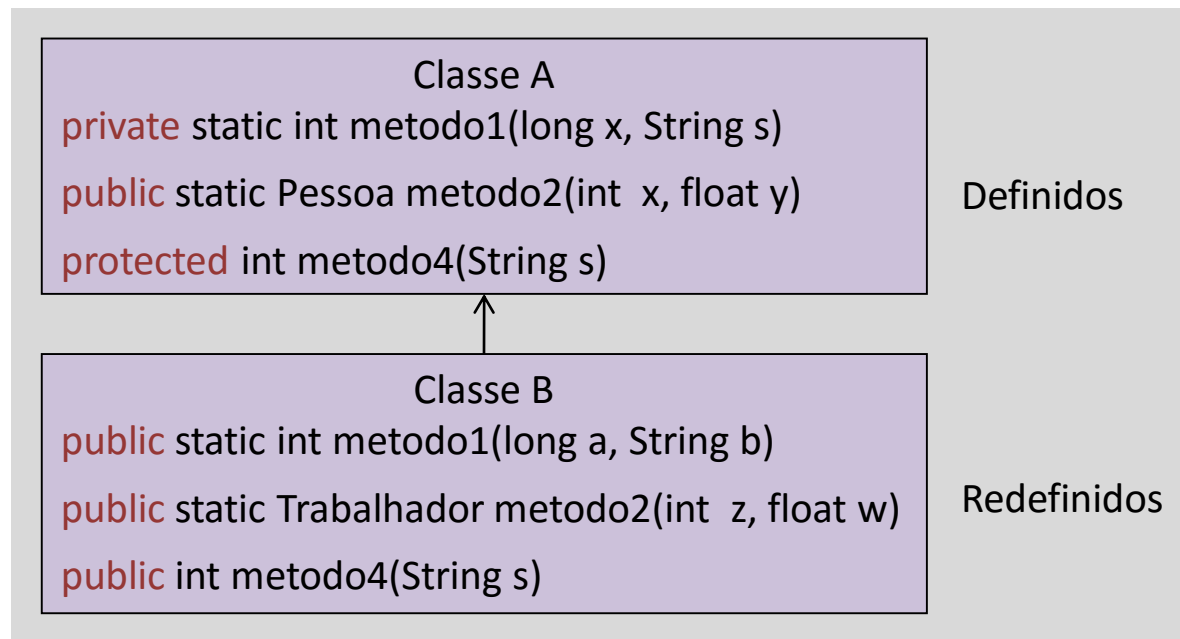


- Requisitos de Método Redefinido (3/5)
  - Nível de acesso (visibilidade)
    - Depende da visibilidade do método **definido**
      - Inacessível
        - Método redefinido pode especificar **qualquer** nível de acesso
      - Acessível
        - Regra geral
          - Método **redefinido** não pode **diminuir** o nível de acesso do método **definido**

Método Definido	Método Redefinido
public	public
protected	protected ou public
package	qualquer, excepto private

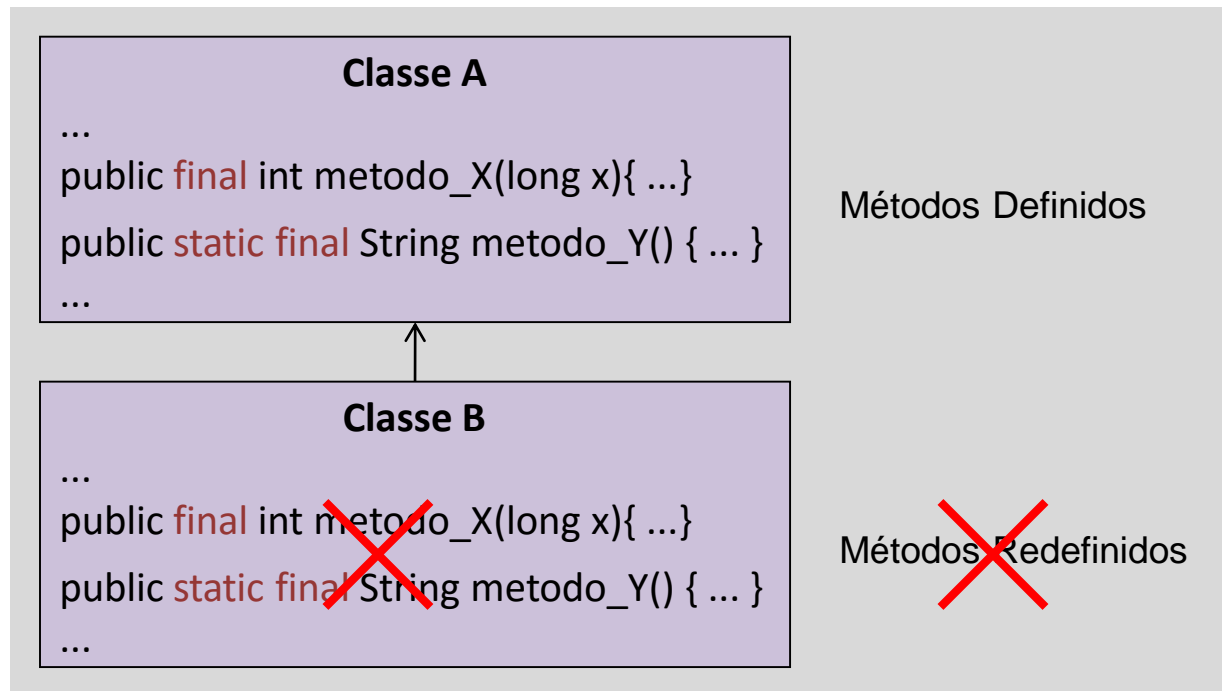
- Exemplo
  - Slide seguinte

- Requisitos de Método Redefinido (4/5)
  - Nível de acesso (visibilidade)
    - Exemplo



- Requisitos de Método Redefinido (5/5)

- Métodos declarados final
  - Não podem ser redefinidos
  - Exemplo



- **Variável Redefinida (classe/instância)**
  - Definição
    - Variável com o mesmo nome de variável **definida** nas suas superclasses
  - Definição local
    - **Esconde** (*hidden*) a definição da variável acessível
  - Irrelevante para variáveis privadas

### Definição

- Referência da **superclasse** (direta) da classe da **instância-recetora** de uma mensagem

### Interesse

- Invocar um membro de **instância herdado** e **acessível** da **superclasse direta**
  - Particularmente útil para acesso a métodos que foram sobrepostos

super **não** se **deve** **aplicar** a variáveis e métodos de classe

### Exemplo

- `super.toString()`      // invoca método toString() da superclasse

```
public class Trabalhador extends Pessoa {  
    private double salario;  
    private String empresa;  
    ...  
    public String toString () {      // baseado no toString() da superclasse  
        return String.format("%s Salário:%s, Empresa:%s", super.toString(), salario, empresa);  
    }  
    ...  
}
```

- Não pode ser usada de forma encadeada
  - Exemplo
    - `super.super.metodo();` // expressão ilegal
- Outro interesse de **super**
  - Distinguir dois tipos de métodos de instância
    - Herdados da superclasse direta // com prefixo **super**
    - Locais da classe // sem prefixo **super**



### Sintaxe

- `super(lista_parâmetros);`                      `// lista pode ser vazia`

### Interesse

- Acesso aos **construtores** da superclasse (**direta**) para **inicialização** das suas variáveis de instância

### Uso

- Só em construtores
- Obrigatoriamente a **1ª instrução** do construtor

### Exemplo

```
public class Trabalhador extends Pessoa {  
    // variável de instância local (própria)  
    private double salario;  
  
    // construtor  
    public Trabalhador( String nome, double salario) {  
        super( nome );                      // garante inicialização das variáveis de instância da superclasse  
        this.salario = salario;  
    }  
}
```

- **Quando não é declarado**
  - Compilador **adiciona automaticamente** `super()` aos construtores `// super sem parâmetros`
- **Criação de instância de subclasse**
  - Obriga **chamada em cadeia dos construtores** de todas as suas superclasses

- [Mecanismos de Reutilização de Código](#)
- [Herança de Classes](#)
- [Polimorfismo](#)
- [Classes Abstratas](#)
- [Classe Object](#)



- [Classe é Tipo de Dados](#)
  - Tipo Referência
- [Tipo e Subtipo Referência](#)
- [Tipo Referência](#)
  - Estático
  - Dinâmico
- [Conversão de Tipos Referência](#)
  - [Tipos Referência Compatíveis](#)
  - Tipos de Conversão
    - [Upcasting](#)
    - [Downcasting](#)
- [Tipos Referência da mesma Variável](#)
  - Um Estático
  - Múltiplos Dinâmicos

- [Procura de Métodos](#)
  - [Tipos de Procura](#)
    - [Estática](#)
    - [Dinâmica](#)
- [Determinação do Tipo Dinâmico de uma Variável](#)
  - Operador instanceof
- [Noção de Polimorfismo](#)
  - [Programação Genérica](#)
    - Vantagens
- [Classe Object](#)
- [Mecanismo de Sobreposição de Métodos \(Overriding\)](#)

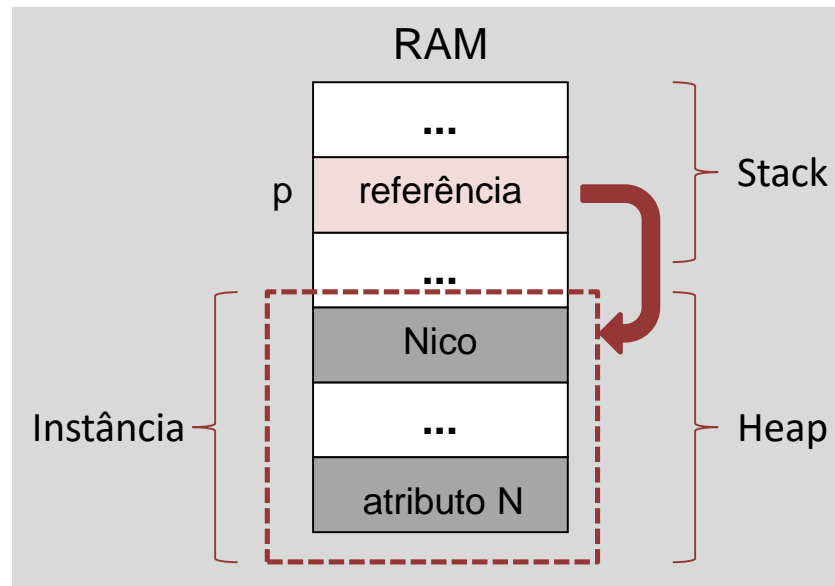
- Classe é também Tipo de Dados

- Tipo **referência**

- Variável deste tipo guarda referência de instância // referência = endereço de memória

- Exemplo

- Pessoa p = new Pessoa("Nico");
      - Classe Pessoa especifica **tipo da variável p**
      - Variável p guarda referência de instância Pessoa



- Porque **define**

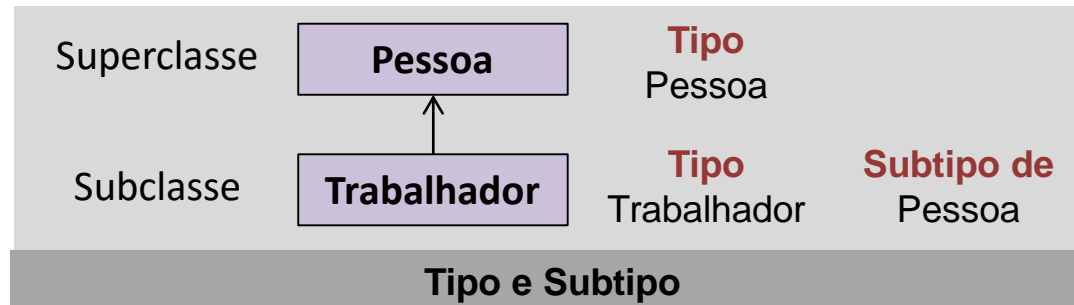
- Conjunto de **instâncias** que podem ser atribuídas a variável // instâncias: dados de programa
    - Conjunto de **operações** sobre essas instâncias // operações: métodos instância

- **Tipo Referência**

- Especificado por
  - Classe
  - Interface // próxima aula

- **Subtipo de Tipo Referência**

- Tipo especificado por classe
  - Tipo de uma **subclasse**
  - Exemplo
    - Trabalhador é **subtipo de Pessoa**



- Tipo Referência

- Estático
- Dinâmico

- Tipo Estático

- Considerado pelo compilador
  - Tempo de compilação (*compile-time*) // tradução do código fonte para código executável
  - Tempo de programação no Netbeans // verificação sintática do programa fonte
- Tipo declarado
- Exemplo

```
Pessoa p; // tipo estático da variável p = Pessoa // tipo declarado
```

- Tipo Dinâmico

- Considerado em tempo de execução (*run-time*)
- Exemplo

```
p = new Pessoa(); // tipo dinâmico da variável p = Pessoa  
p = new Trabalhador(); // tipo dinâmico da variável p = Trabalhador
```

## Requisito

- Tipos compatíveis

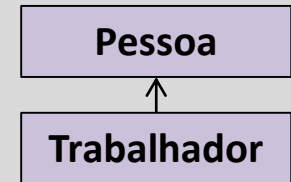
```
// (tipo_2) tipo_1 ⇒ tipo_1 e tipo_2 compatíveis
```

## Tipos Referência Compatíveis

- Tipos
- Respetivos Subtipos
  - Mesmo ramo da hierarquia

```
// Ex: tipo Pessoa
```

```
// Ex: subtipo Trabalhador
```



## Variável

- Tipo
  - Pode guardar objetos de subtipos
- Subtipo
  - Pode guardar objetos do tipo

```
// Pessoa p1, p2 = new Pessoa(), p3 = new Pessoa();
```

```
// p1 = new Trabalhador();
```

```
// Trabalhador t;
```

```
// t = (Trabalhador) p2;
```

## Justificação

- Possuem métodos **semelhantes**
  - Aplicam-se
    - Objetos do Tipo
    - Objetos do Subtipo

```
// Ex: Pessoa e Trabalhador possuem ambos getNome()
```

```
// ... p2.getNome()
```

```
// invoca getNome() de Pessoa
```

```
// p2 = (Pessoa) t;
```

```
// conversão
```

```
// ... p2.getNome()
```

```
// invoca getNome() de Trabalhador
```

```
// t = new Trabalhador();
```

```
// ... t.getNome()
```

```
// invoca getNome() de Trabalhador
```

```
// t = (Trabalhador) p3;
```

```
// conversão
```

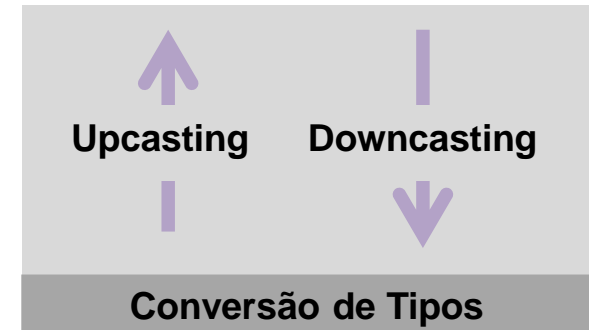
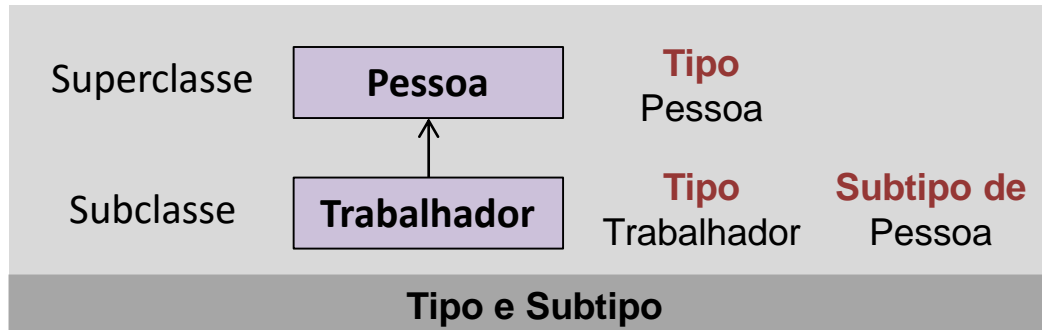
```
// ... t.getNome();
```

```
// invoca getNome() de Pessoa
```



## Conversão *Upcasting*

- Subtipo → Tipo



## Exemplo

```
Pessoa p = new Pessoa();           // p guarda referência de objeto do mesmo tipo
Trabalhador t = new Trabalhador();
p = t;                             // p guarda referência de objeto de um subtipo
String nome = p.getNome();         // executado getNome() de Trabalhador
```

## Conversão *Implícita*

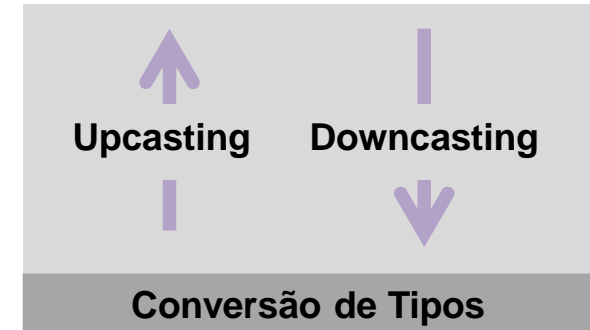
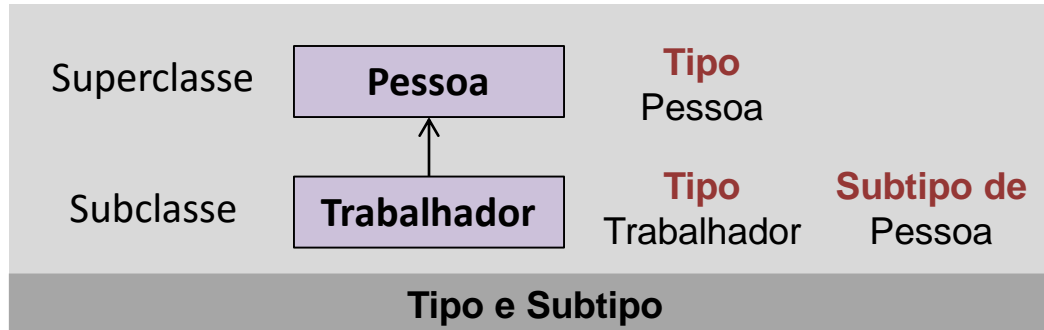
- Não precisa de ser declarada // Ex: `p = t` em vez de `p = (Pessoa) t`

## Conversão de tipos *segura*

- Não provoca erros de execução
- Porque todos os métodos do **tipo** são possuídos pelo **subtipo** // Ex: **Trabalhador** e **Pessoa**

- Conversão *Downcasting*

- Tipo → Subtipo



- Exemplo

```
Pessoa p = new Pessoa();           // p guarda referência de objeto do mesmo tipo
Trabalhador t = new Trabalhador();
t = (Trabalhador) p;               // t guarda referência de objeto de um tipo
String nome = t.getNome();         // executado getNome() de Pessoa
String empresa = t.getEmpresa();   // gera erro de execução ; p não possui getEmpresa()
```

- Conversão *explícita* (declarada)
  - Compilador passa responsabilidade para programador
- Conversão de tipos *insegura*
  - Pode provocar erros de execução
  - Porque *tipo*, geralmente, não possui todos métodos do *subtipo* // Ex: Pessoa e Trabalhador

- **Variável Tipo Referência**

- Tem **um** tipo estático
- Pode ter **múltiplos** tipos dinâmicos

- **Tipo Estático**

- Tipo declarado
  - Único tipo considerado pelo **compilador** // não se altera
- Exemplo

```
Pessoa p;           // tipo estático de p = Pessoa
p = new Trabalhador(); // tipo estático de p = Pessoa
```

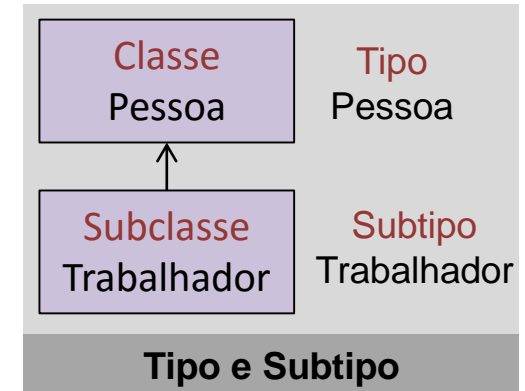
- **Tipos Dinâmicos**

- Considerados em tempo de execução
- Exemplo

```
p = new Trabalhador(); // tipo p: estático = Pessoa e dinâmico = Trabalhador
```

- Exemplo de variável com diferentes tipos dinâmicos

```
Pessoa p;           // p é do tipo dinâmico Pessoa
p = new Pessoa();    // p mantém-se do tipo dinâmico Pessoa
p = new Trabalhador(); // p passa a ser do tipo dinâmico Trabalhador
```



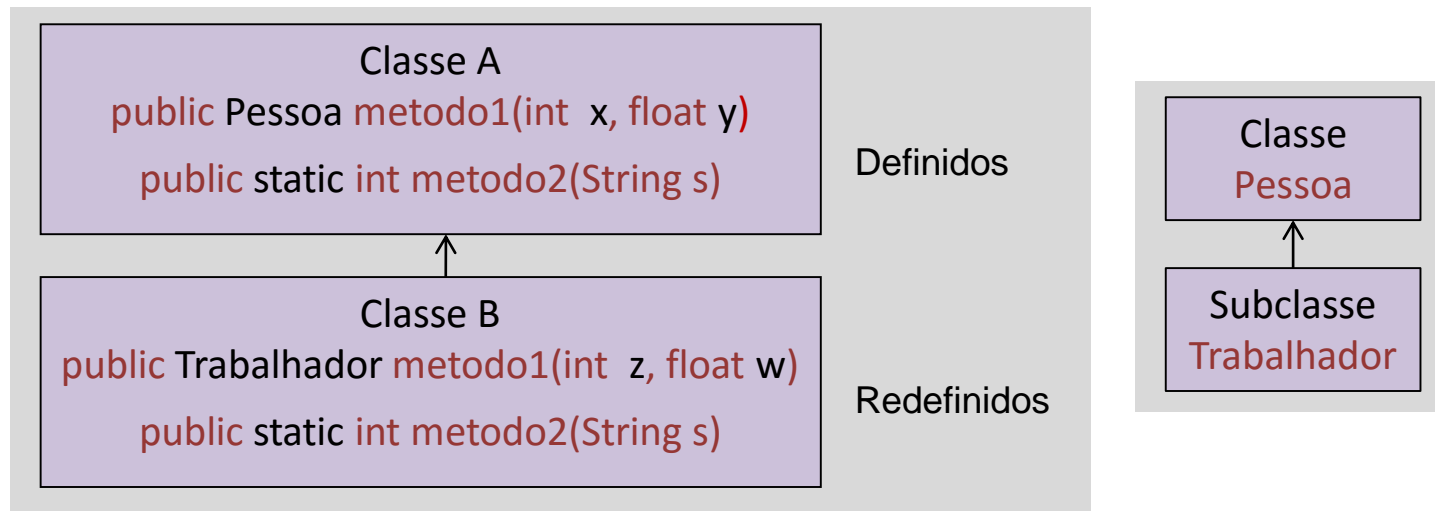
- **Redefinição de métodos permite a uma classe**

- Possuir **múltiplos** métodos de instância com a **mesma** assinatura
  - Um **redefinido** + outros **herdados**
  - Ou apenas herdados
- Aceder a **múltiplos** métodos de classe com a **mesma** assinatura, das suas superclasses

- **Questão**

- Como é procurado o método a executar entre vários alternativos ?
- Exemplo

```
instânciaB.metodo1(12, 23.4f) // método1 da classe B ou de A?  
B.metodo2("Mensagem")       // método2 da classe B ou de A?
```



- Tipos de Procura

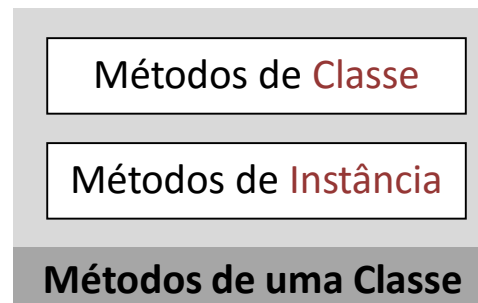
- Estática
  - // em tempo de compilação (*compile-time*) pelo compilador
  - // em tempo de programação no Netbeans pelo compilador
- Dinâmica
  - // em tempo de execução (*run-time*)

- Procura Estática

- Métodos de Classe
- Métodos de Instância

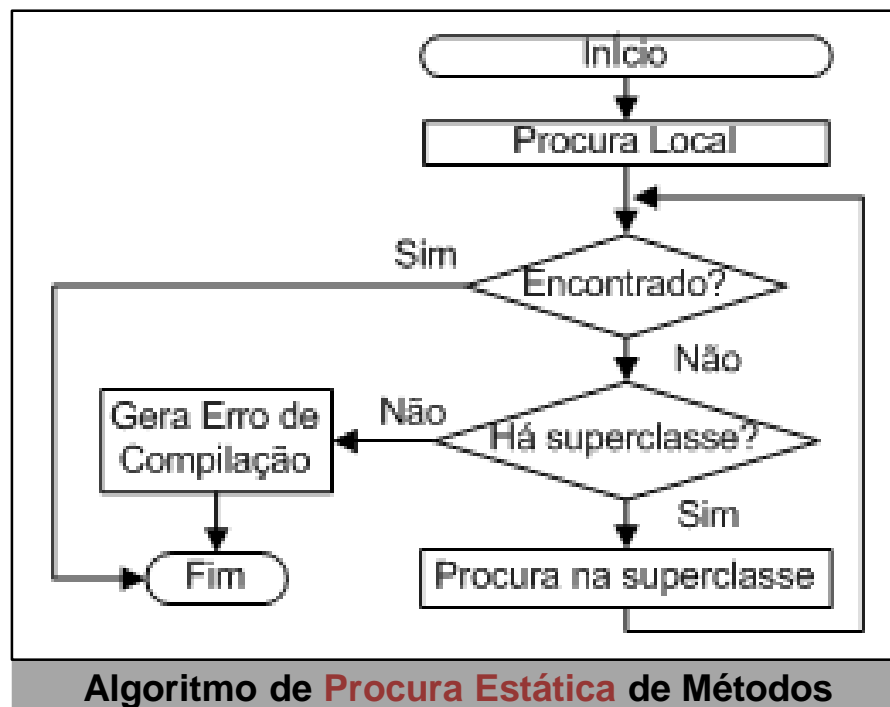
- Procura Dinâmica

- Métodos de Instância



- Algoritmo de Procura Estática de Métodos

- Procura do método solicitado começa sempre na **classe do objeto-recetor** de mensagem
  - Chamada procura local
- Se o método pretendido **não** for **encontrado** nessa classe, a procura continua nas suas **superclasses**, segundo a ordem hierárquica e até à classe Object
- Se a procura chegar à classe Object e o método pretendido não for encontrado, é gerado um **erro de compilação**



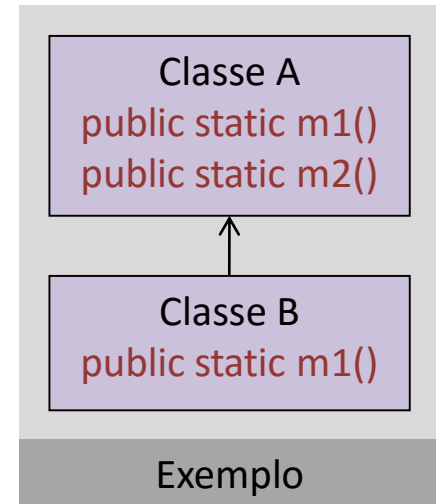
- Métodos de Classe

- Procurados apenas em tempo de compilação
- Exemplos de Métodos **Acessíveis**
  - Aplicados a **Classes**

```
A.m1();    // tipo estático de A é A ⇒ determinado m1 de A
B.m1();    // tipo estático de B é B ⇒ determinado m1 de B
B.m2();    // encontrado m2 de A
```

- Aplicados a **Instâncias**

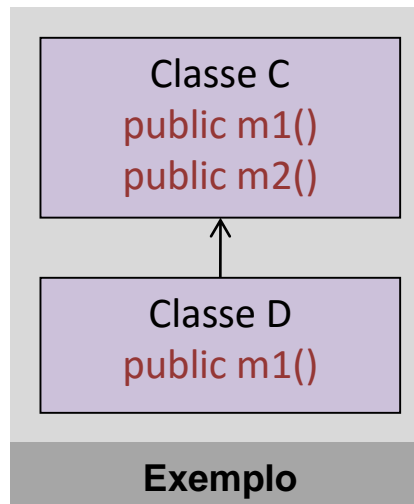
```
B b1 = new B();
b1.m1();    // tipo estático de b1 é B ⇒ determinado m1 de B
A a1 = b1;
a1.m1();    // tipo estático de a1 é A ⇒ determinado m1 de A
```



- Métodos de Instância

- Acessíveis só podem ser aplicados a instâncias
- Exemplo

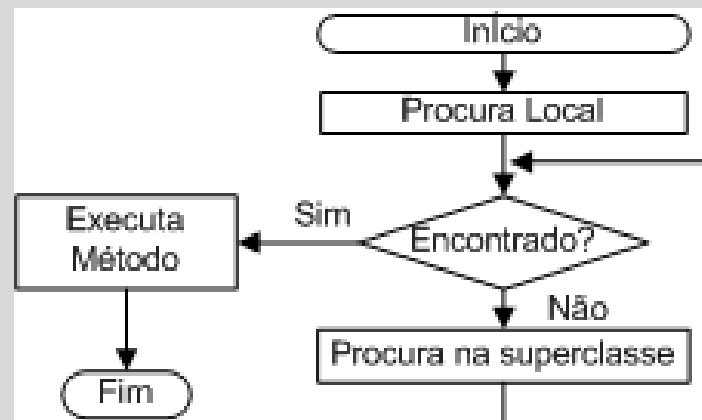
```
D d1 = new D();  
C c1 = new C();  
c1.m1();           // tipo estático de c1 é C ⇒ determinado m1 de C  
c1 = d1;  
c1.m1();           // tipo estático de c1 é C ⇒ determinado m1 de C  
d1.m2();           // tipo estático de d1 é D ⇒ determinado m2 de C
```





- Algoritmo de Procura Dinâmica

- Permite à **instância-recetora** de uma mensagem
  - Determinar o método **apropriado** para responder, entre múltiplos métodos **sobrepostos**
    - Métodos com a mesma assinatura
- Método será **sempre** encontrado
  - Código garantido pelo compilador
    - Tudo exceto *downcastings*
      - Responsabilidade do programador



Algoritmo de **Procura Dinâmica** de Métodos em Código Garantido pelo Compilador (tudo exceto *downcastings*)

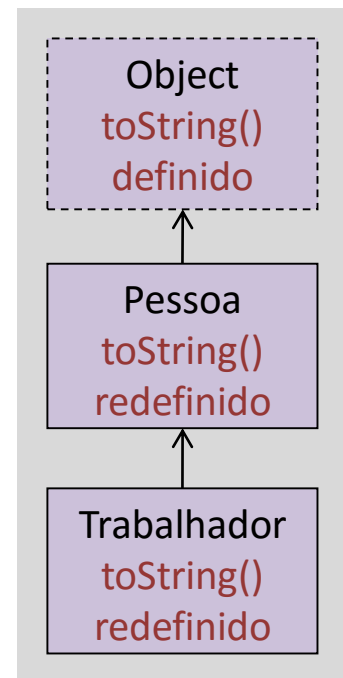
## ■ Métodos de Instância

- Métodos **acessíveis** só podem ser **aplicados** a instâncias
- Exemplo

```
Pessoa p;                // p é do tipo Pessoa
p = new Pessoa();         // p mantém-se do tipo Pessoa
System.out.println( p.toString() ); // executado toString() da classe Pessoa
p = new Trabalhador();    // p passa a ser do tipo Trabalhador
System.out.println( p.toString() ); // executado toString() de Trabalhador
```

## ■ Observações

- Mesma variável **p**
  - Com diferentes tipos dinâmicos
- Mesma expressão sintática **p.toString()**
  - Com diferentes semânticas



## ▪ Operador instanceof

- Sintaxe: nomeInstância instanceof nomeClasse
- Retorna: true ou false
  - true se Instância for da Classe ou de qualquer uma das suas subclasses
    - Instância de subclasse é considerada instância de qualquer superclasse sua
      - Porque responde às mesmas mensagens das suas instâncias

## ▪ Exemplo

```
// Declaração de contentor de instâncias de diferentes tipos Pessoa
Pessoa[] pessoas = new Pessoa[100];

// Preenchimento do contentor
pessoas[0] = new Trabalhador() ;
pessoas[1] = new Pessoa() ;
...
// Mostrar nomes de todas as pessoas do contentor
for (int i=0; i<pessoas.length; i++)
    System.out.println(" Nome: " + pessoas[i].getNome() );           // instâncias Pessoa, Trabalhador e Estudante

// Mostrar as empresas dos trabalhadores
for (int i=0; i<pessoas.length; i++)
    if (pessoas[i] instanceof Trabalhador)
        System.out.println( "Empresa: "+ ( (Trabalhador) pessoas[i] ).getEmpresa() ); // conversão tipo estático
```

- **Noção**

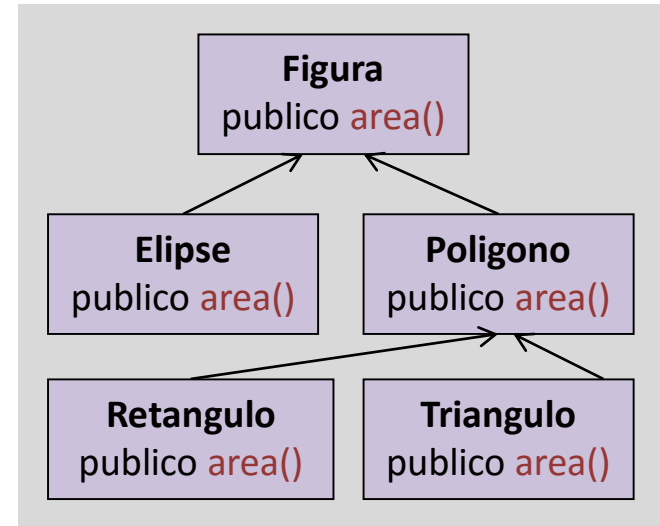
- **Mecanismo** da POO que permite a variável de um tipo **estático** assumir múltiplos tipos **dinâmicos**
  - i.e., permite a uma **variável** assumir **múltiplas formas** (polimorfismo ... das variáveis)

- **Interesse**

- Permitir uma **Programação Genérica** para criar programas
  - Mais **simples**
  - Mais facilmente **estendíveis** a novas classes
    - **Programação Incremental**

- **Programação Genérica**

- Princípios
  - Todas as classes da hierarquia devem possuir a **mesma API**:
    - Mesmos métodos públicos para **responderem** às **mesmas mensagens**
  - Programação deve ser **feita para o tipo mais genérico** da hierarquia (topo)
    - Programador não tem de se preocupar com subtipos (existentes e futuros)
- É **particularmente simples**
  - Usando contentores de instâncias



- **Exemplo:** programa para mostrar áreas de diferentes figuras geométricas
  - Todas as classes da hierarquia têm a mesma API (método `area()`)
    - Definido na classe mais genérica (topo da hierarquia)
    - Redefinido nas subclasses
  - Programação **mais simples** feita para o tipo mais genérico (**Figura**) requer um contentor desse tipo
    - Tipo genérico é o tipo compatível com todos os subtipos da hierarquia

// **Estrutura de Dados** para guardar todas as instâncias da hierarquia de classes: **contentor**

```
Figura[] figuras = new Figura[100];
```

// **Preenchimento** do contentor

```
figuras[0] = new Elipse();
```

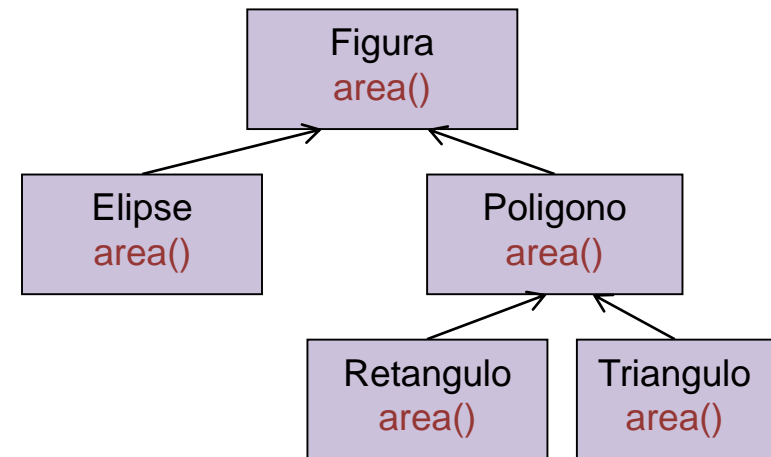
```
figuras[1] = new Retangulo();
```

```
figuras[2] = new Triangulo();
```

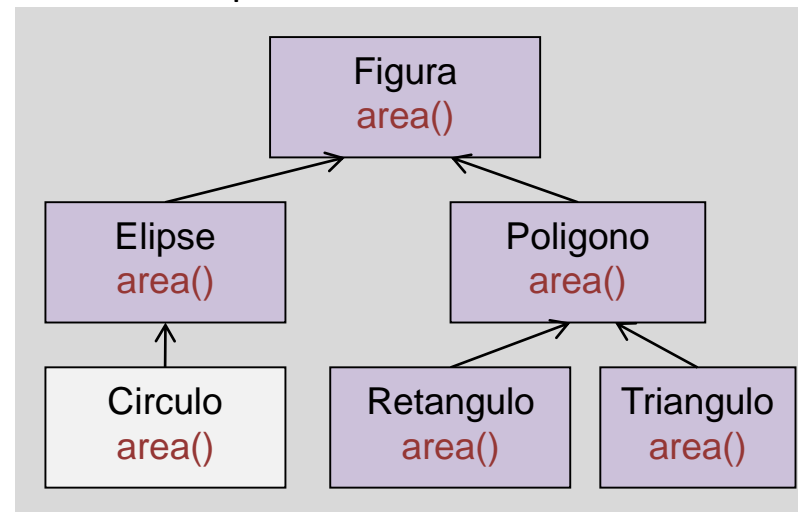
// **Algoritmo** para calcular e mostrar áreas

```
for( int i=0; i<figuras.length; i++)
```

```
    System.out.println( figuras[i].area() );
```



- Permite a criação de **programas**
  - Mais **Simples**
  - Mais Compactos
  - Mais fáceis de escrever, ler, perceber e manter
- **Facilita a extensão dos programas** a novos subtipos da hierarquia
  - Porque o código genérico também funciona para novos subtipos da hierarquia
  - Minimiza as alterações necessárias
    - Apenas ao nível do preenchimento de Estruturas de Dados
    - Algoritmo mantém-se intacto
  - Exemplo
    - Extensão do programa anterior para suportar um novo subtipo **Circulo**
    - Basta acrescentar **apenas** uma linha de código  
`figuras[3] = new Circulo();`
    - Propriedade dos programas muito importante
      - Garante a **continuidade do código**
- Exige um **menor esforço de programação**
  - ⇒ Menores custos de produção de software



## ▪ Tipo Object

- **Compatível** com **todos** os tipos referência
- Variável deste tipo
  - Pode representar, **dinamicamente**
    - Instâncias de qualquer classe
  - Exemplo

```
Object obj = new Pessoa();  
obj = new Trabalhador();
```

## ▪ API (Métodos Públicos)

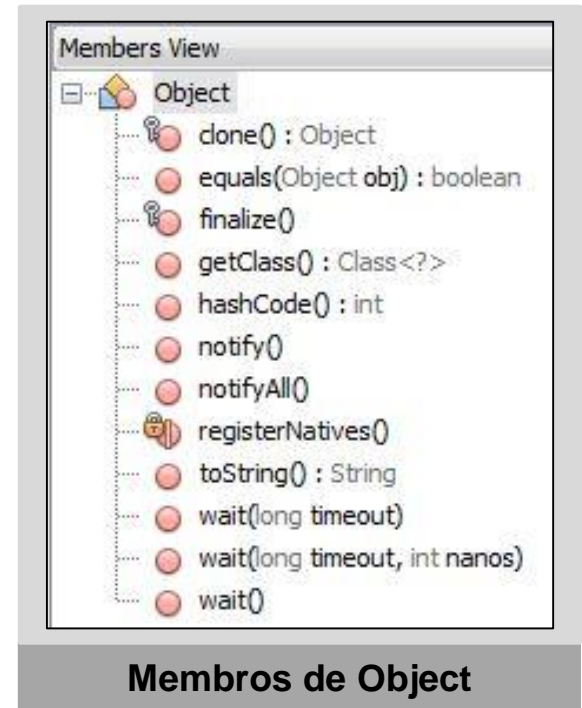
- Muito limitada
  - Poucas mensagens para enviar a instâncias Object
  - Ex: `obj.toString()`      `// mensagem toString() enviada a obj`

## ▪ Programação para tipo Object

- Aumento de **mensagens** para enviar a instâncias Object
  - Requer **downcastings** para subtipos de Object
  - Exemplo

```
((Trabalhador) pessoas[i]).getEmpresa()      // mensagem getEmpresa() enviada a pessoas[i]
```

- Há preocupações com os tipos das instâncias

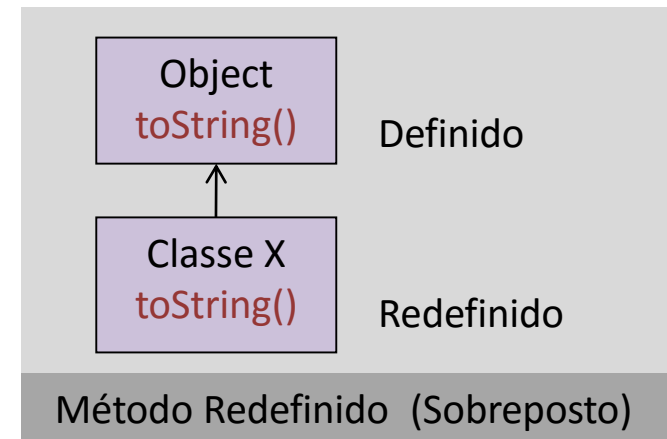


- **Suporta**

- Redefinição de métodos **acessíveis** (classe/instância)

- **Sobreposição**

- Porque a redefinição **local** de método acessível **sobrepõe-se** à sua **definição**
    - Método **redefinido** (**sobreposto**), segundo o algoritmo de procura de métodos
      - Primeiro a ser encontrado
      - Tem maior **prioridade**
      - Objeto-recetor de mensagem executa sempre o **método sobreposto**



- **Não confundir com *Overloading***

- Mecanismo de sobrecarga de métodos/construtores numa classe
  - Permite a definição de **múltiplos** métodos/construtores com o mesmo nome
  - Exemplo
    - Múltiplos construtores na mesma classe



- [Mecanismos de Reutilização de Código](#)
- [Herança de Classes](#)
- [Polimorfismo](#)
- [Classes Abstratas](#)
- [Classe Object](#)



- Noção de Classe
  - [Abstrata](#)
  - [Concreta](#)
- Classe Abstrata
  - [Declaração](#)
    - [Método Abstrato](#)
  - [Herança](#)
  - [Interesse](#)
  - [Construtores](#)
  - [Tipo Referência](#)

## ■ Classe Abstrata

- Pode definir métodos abstratos
  - Métodos de instância
  - Não privados
  - Declarados apenas sintaticamente // só cabeçalho; corpo vazio; não implementados
- Classe não instanciável
  - Sem capacidade para criar instâncias
  - Caso contrário, instâncias não seriam capazes de responder a mensagens correspondentes a métodos abstratos ⇒ erros de execução
- Classe com definição incompleta
  - Se possuir métodos abstratos // métodos não implementados

## ■ Exemplo

```
public abstract class Exemplo {           // modificador abstract declara classe abstrata
    ...
    public abstract double m1();           // método abstrato (declarado abstract)
    public abstract int m2( String s );
    public int m3( int x ) { return x*20; } // método não abstrato (implementado)
}
```

## ▪ Classe Concreta

- Classe não abstrata
- Tem definição completa // todos os métodos não-abstratos
- Pode ser instanciável

## ▪ Exemplo

```
public class Exemplo2 {                                // sem modificador abstract
    ...
    // métodos de instância
    public double m1() {
        return Math.random() * 100;                    // implementado
    }
    public String m2( String s ) {
        return String.format("%-7s",s);                // implementado
    }
    public double m3( int x ) {
        return Math.random() * x;                      // implementado
    }
}
```

▪ **Sintaxe**

```
[modificador de acesso] abstract class NomeClasse [extends SuperClasse] [implements Interfaces] {  
    //membros da classe  
}
```

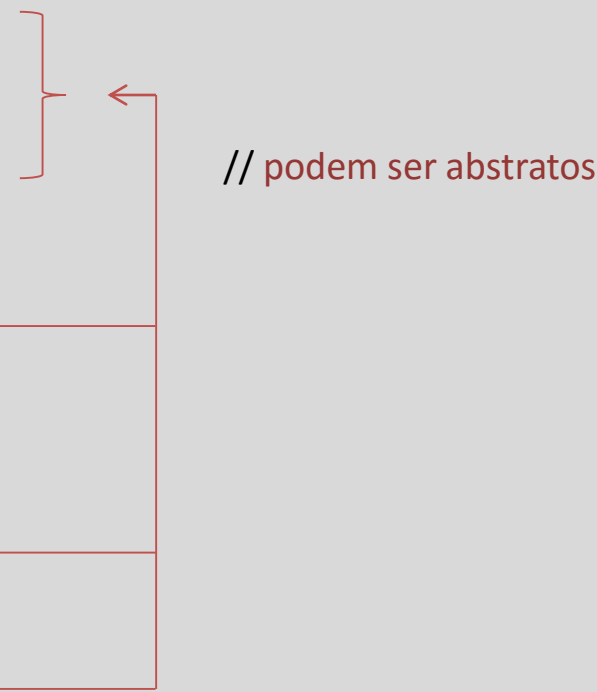
- [...] opcional
- **modificador de acesso** public ou sem modificador = package
- **NomeClasse** letra inicial maiúscula
- **extends** aplica-se a classe que estende outra classe (herança)
- **implements** aplica-se a classe que implementa um ou mais interfaces

▪ **Exemplo**

```
public abstract class Exemplo {           // Nome da classe iniciado com letra maiúscula  
    ...  
}
```

## ▪ Organização dos Membros da Classe

```
[modificador de acesso] abstract class NomeClasse [extends SuperClasse] [implements Interfaces] {  
    // variáveis de instância  
    // constantes de classe  
    // variáveis de classe  
  
    // membros públicos  
        // construtores  
        // métodos de instância  
            // métodos de consulta (gets)  
            // métodos de modificação (sets)  
            // métodos complementares e auxiliares  
        // métodos de classe  
        // organização  
  
    // outros membros privados  
        // métodos de instância  
        // organização  
        // métodos de classe  
        // organização  
}
```



// podem ser abstratos

## ▪ Método Abstrato

## ▪ Sintaxe

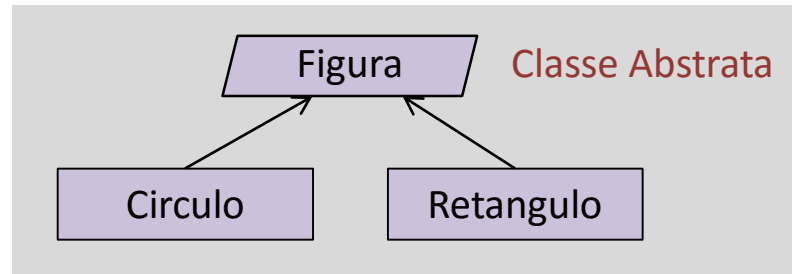
```
[modificador de acesso] abstract tipo_retorno nome_método( lista_parâmetros );
```

- [...] opcional
- **modificador de acesso** public, protected ou sem modificador = package
- **nome\_método** letra inicial minúscula
- **lista\_parâmetros** tipo1 nome1, tipo2 nome2, ... ; pode ser lista vazia

## ▪ Exemplo

```
public abstract class Exemplo2 {  
    ...  
    public abstract double m1();  
    protected abstract String m2( int x );  
    abstract void m3( Pessoa p, int n );  
    ...  
}
```

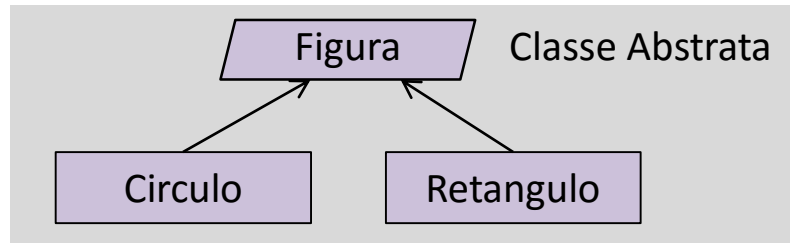
- **Classe Abstrata**
  - Pode ser Herdada
    - Exemplo



- **Subclasse de Classe Abstrata**
  - Herda métodos abstratos
    - Se implementar
      - Todos  $\Rightarrow$  Classe **Concreta**
      - Alguns  $\Rightarrow$  Classe **Abstrata**



- **Garantir uma Programação Genérica** (i.e., programação simples)
  - Quando é difícil criar uma hierarquia de classes todas concretas, devido à **impossibilidade de implementação** de todos os **métodos** em classes genéricas
    - Exemplo
      - Classes para representação e manipulação de figuras geométricas
      - Como implementar os métodos comuns, `área()` e `perímetro()`, na classe genérica `Figura`?



- Uma classe abstrata colocada no **topo** de uma hierarquia de classes pode garantir que todas as classes concretas (instanciáveis) da hierarquia implementem a **mesma API**
  - Tenham implementações dos mesmos métodos de instância públicos
  - Sejam classes cujas instâncias respondam às mesmas mensagens

## ▪ Classe Abstrata

- Pode declarar construtores

## ▪ Interesse

- Para subclasses
  - Inicializarem variáveis de instância herdadas
- Não servem para criar instâncias

// classe abstrata não é instanciável

## ▪ Exemplo

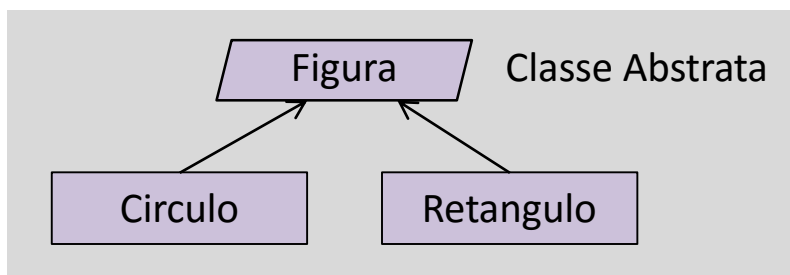
```
public abstract class Exemplo3 {  
    ...  
    private int x;  
    public Exemplo3( int x ) { this.x = x; };  
    ...  
}
```

```
public class Exemplo4 extends Exemplo3 {  
    ...  
    public Exemplo4( int x ) { super(x); };  
    ...  
}
```

- Tipo referência // como qualquer classe concreta

- Tipo Figura

- Definido por classe abstrata Figura



- Usado na declaração de variável

Figura f;

- Tipo compatível com seus subtipos

- Exemplo

- Figura compatível com subtipos Circulo e Retangulo
- f pode representar instâncias Circulo e Retangulo

```
Figura f = new Circulo();  
f = new Retangulo();
```

- [Mecanismos de Reutilização de Código](#)
- [Herança de Classes](#)
- [Polimorfismo](#)
- [Classes Abstratas](#)
- [Classe Object](#)



### Classe Java mais Genérica

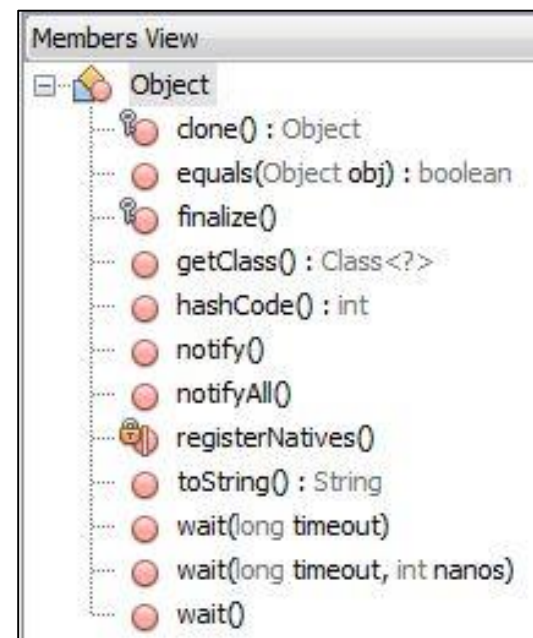
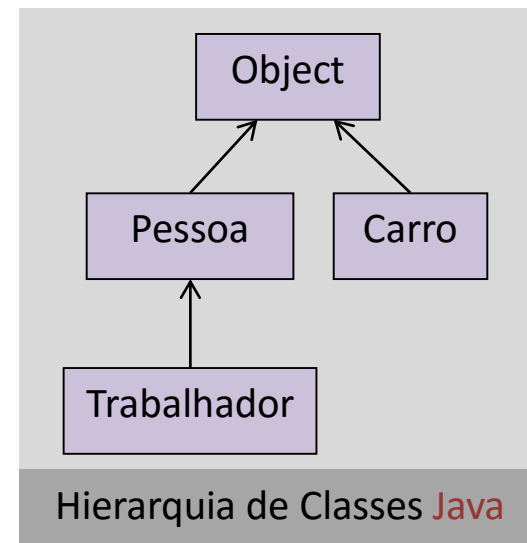
- Está no **topo** de todas as hierarquias de classes Java
- Superclasse de **todas** as classes
  - Qualquer classe em Java é subclasse, **pelo menos**, de Object

### Membros

- Comuns a todas as classes Java
  - Interessam a todas as classes
- Exemplos de Métodos
  - toString
  - equals

### Nome Object

- Objeto ... é característica comum a todas as classes



- Métodos (instância) que classes instanciáveis devem reescrever (1/3)
  - `public String toString() { ... }`
    - Deve retornar representação textual legível do objeto
    - **Em Object**
      - Retorna nome da classe e identificador interno hexadecimal
  - `protected Object clone() { ... }`
    - Java não garante implementação adequada
    - Construtor de cópia (ou clone)

```
public class Pessoa {  
    private String nome;  
    private int idade;  
    ...  
    public Pessoa(Pessoa p){                // construtor de cópia  
        this.nome = p.nome;  
        this.idade = p.idade;  
    }  
    ...  
    public Pessoa clone(){ return new Pessoa(this); }  
}
```

- Métodos (instância) que classes instanciáveis devem reescrever (2/3)
  - `public boolean equals(Object obj) { ... }`
    - Deve testar a igualdade de características de 2 objetos // equivalência
      - Mesma classe
      - Estados iguais
    - Em Object
      - Testa apenas igualdade de **endereços** // Object não tem estado
        - `obj == this` (objeto recetor da mensagem)
  - Exemplo de método reescrito ...

```
public class Pessoa {  
    private String nome;  
    private int idade;  
  
    ...  
  
    public boolean equals(Object obj){  
        if(this==obj) return true;  
        if(obj==null || this.getClass()!=obj.getClass()) return false;  
        Pessoa p = (Pessoa) obj;  
        return nome.equals(p.getNome()) && idade==p.getIdade();  
    }  
}
```

Método `getClass()`

- Retorna objeto da classe Class que representa classe do objeto recetor.

- Métodos (instância) que classes instanciáveis devem reescrever (3/3)
  - public boolean **equals**(Object obj) { ... }
    - Exemplo de método reescrito ... numa **hierarquia de classes**

```
public class Pessoa {    // superclasse

    private String nome;
    private int idade;
    ...

    public boolean equals(Object obj){
        if(this==obj) return true;
        if(obj==null || this.getClass()!=obj.getClass()) return false;
        Pessoa p = (Pessoa) obj;
        return nome.equals( p.getNome() ) && idade==p.getIdade();
    }
}

public class Trabalhador extends Pessoa {    // subclasse

    private String empresa;
    ...

    public boolean equals(Object obj){
        if( !super.equals(obj) ) return false;
        Trabalhador t = (Trabalhador) obj;
        return empresa.equals( t.getEmpresa() );
    }
}
```

Método **getClass()**

- Retorna objeto da classe Class que representa classe do objeto recetor.