

# Herança

Programação Orientada a Objetos — QXD0007



UNIVERSIDADE  
FEDERAL DO CEARÁ  
CAMPUS QUIXADÁ

Prof. Atílio Gomes Luiz  
gomes.atilio@ufc.br

Universidade Federal do Ceará

1º semestre/2022



# Leituras para este tópico

- **Capítulo 9** (Herança) do livro Java Como Programar, Décima Edição.
- **Capítulo 9** (Herança, reescrita e polimorfismo) da apostila da Caelum – Curso FJ-11,

# Introdução



# Reutilização de Classes

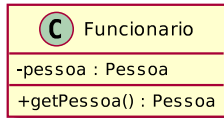
- Vimos 3 relacionamentos entre objetos que permitem o reuso de classes:
  - **Associação, Agregação e Composição**
  - Nestes relacionamentos, um objeto de uma classe contém uma referência para um objeto de outra classe.
    - A **tem um** B
    - A **contém** B
    - A **é formada por** B
  - Esse mecanismo é chamado de **delegação**.

# Reutilização de Classes

- Vimos 3 relacionamentos entre objetos que permitem o reuso de classes:
  - **Associação, Agregação e Composição**
  - Nestes relacionamentos, um objeto de uma classe contém uma referência para um objeto de outra classe.
    - A **tem um** B
    - A **contém** B
    - A **é formada por** B
  - Esse mecanismo é chamado de **delegação**.
- Porém, nem sempre o mecanismo de delegação é o mais natural para a reutilização de classes já existentes.
  - Vamos ver um exemplo a seguir.

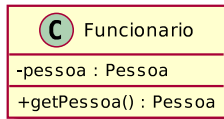
## Reutilização de Classes – Exemplo

Temos duas classes **Pessoa** e **Funcionario** tal que uma instância de **Pessoa** é declarada dentro da classe **Funcionario** para representar os dados da pessoa/funcionário.



## Reutilização de Classes – Exemplo

Temos duas classes **Pessoa** e **Funcionario** tal que uma instância de **Pessoa** é declarada dentro da classe **Funcionario** para representar os dados da pessoa/funcionário.



Queremos criar uma classe **ChefeDeDepartamento**. Um chefe de departamento é um funcionário que é responsável por um departamento.

## Reutilização de Classes – Exemplo

Temos duas classes **Pessoa** e **Funcionario** tal que uma instância de **Pessoa** é declarada dentro da classe **Funcionario** para representar os dados da pessoa/funcionário.

<b>C</b> Funcionario
-pessoa : Pessoa
+getPessoa() : Pessoa

Queremos criar uma classe **ChefeDeDepartamento**. Um chefe de departamento é um funcionário que é responsável por um departamento.

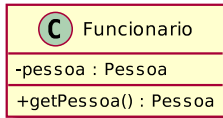
Usando o mecanismo de *delegação*, podemos declarar uma instância de **Funcionario** dentro da classe **ChefeDeDepartamento** e acrescentar alguns campos que diferenciam **ChefeDeDepartamento** de **Funcionario**.

<b>C</b> ChefeDeDepartamento
-funcionario : Funcionario -departamento : Departamento
+getFuncionario() : Funcionario +getDepartamento() : Departamento



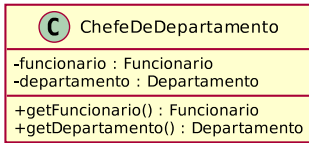
## Reutilização de Classes – Exemplo

Temos duas classes **Pessoa** e **Funcionario** tal que uma instância de **Pessoa** é declarada dentro da classe **Funcionario** para representar os dados da pessoa/funcionário.



Queremos criar uma classe **ChefeDeDepartamento**. Um chefe de departamento é um funcionário que é responsável por um departamento.

Usando o mecanismo de *delegação*, podemos declarar uma instância de **Funcionario** dentro da classe **ChefeDeDepartamento** e acrescentar alguns campos que diferenciam **ChefeDeDepartamento** de **Funcionario**.



**Problema:** Declarar que **ChefeDeDepartamento** contém um funcionário soa artificial — um chefe de departamento **é um tipo de** funcionário, que tem campos adicionais para representar dados específicos de um chefe de departamento, e métodos para manipular esses campos.

# Herança

- Herança é um tipo de relacionamento entre classes que permite que uma classe herde **todos** os dados e comportamentos de outra classe.
- O mecanismo de herança é o mais apropriado para criar relações *é-um-tipo-de* entre classes.

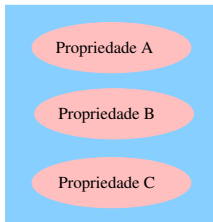
# Herança

- Herança é um tipo de relacionamento entre classes que permite que uma classe herde **todos** os dados e comportamentos de outra classe.
- O mecanismo de herança é o mais apropriado para criar relações *é-um-tipo-de* entre classes.

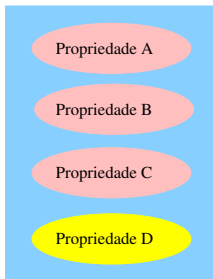
- **Superclasse:** é a classe cujas propriedades são herdadas por outra classe. É também chamada de **classe base** ou **classe pai**.
- **Subclasse:** é a classe que herda propriedades da classe base. É também chamada de **classe derivada** ou **classe filha**.

# Conceito de Herança

Superclasse



Subclasse



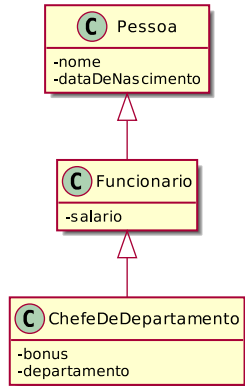
Definido na superclasse,  
mas acessível  
a partir da subclasse



Definido na subclasse

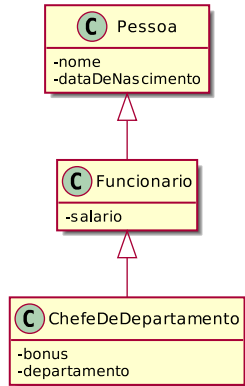
# Herança — Superclasses e subclasses

- Uma subclasse é uma forma especializada da superclasse.
- Uma subclasse também pode vir a ser uma superclasse.



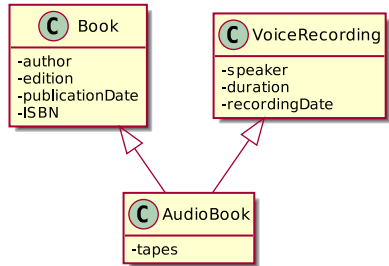
# Herança — Superclasses e subclasses

- Uma subclasse é uma forma especializada da superclasse.
- Uma subclasse também pode vir a ser uma superclasse.
- A **superclasse direta** é a superclasse da qual a subclasse herda explicitamente.
  - As outras são consideradas **superclasses indiretas**.



# Herança Única × Herança Múltipla

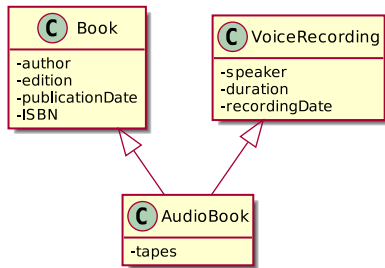
- **Herança múltipla** é quando uma subclasse pode herdar de mais de uma superclasse direta.
- Na **Herança única**, uma subclasse herda somente de uma superclasse direta.



Exemplo de herança múltipla

# Herança Única × Herança Múltipla

- **Herança múltipla** é quando uma subclasse pode herdar de mais de uma superclasse direta.
- Na **Herança única**, uma subclasse herda somente de uma superclasse direta.



Exemplo de herança múltipla

- C++ permite herança múltipla, porém **Java não permite herança múltipla**.
- No entanto, é possível utilizar **interfaces** para desfrutar de alguns dos benefícios da herança múltipla.



# Herança em Java

Em Java, para se estabelecer que uma classe é herdeira de outra, após o nome da subclasse que está sendo declarada coloca-se a cláusula **extends** seguido do nome da superclasse. Por exemplo:

```
class Funcionario extends Pessoa {...}
```

# Herança em Java

Em Java, para se estabelecer que uma classe é herdeira de outra, após o nome da subclasse que está sendo declarada coloca-se a cláusula **extends** seguido do nome da superclasse. Por exemplo:

```
class Funcionario extends Pessoa {...}
```

- Com o mecanismo de herança, podemos declarar a classe Funcionario como sendo um tipo de Pessoa, e a classe Funcionario **herdará** todos os campos e métodos da classe Pessoa, não sendo necessária a sua redeclaração.

# Herança em Java


Em Java, para se estabelecer que uma classe é herdeira de outra, após o nome da subclasse que está sendo declarada coloca-se a cláusula **extends** seguido do nome da superclasse. Por exemplo:


```
class Funcionario extends Pessoa {...}
```

- Com o mecanismo de herança, podemos declarar a classe **Funcionario** como sendo um tipo de **Pessoa**, e a classe **Funcionario** **herdará** todos os campos e métodos da classe **Pessoa**, não sendo necessária a sua redeclaração.
- **Atenção:** Os atributos privados são herdados, mas, como só podem ser acessados e modificados pelas classes que os declararam diretamente, não podem ser acessados diretamente pela subclasse.

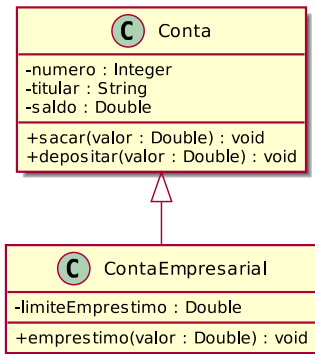
# Exemplo

Suponha um negócio de banco que possui uma conta comum e uma conta para empresas, sendo que a conta para empresa possui todos membros da conta comum, mais um limite de empréstimo e uma operação de realizar empréstimo.

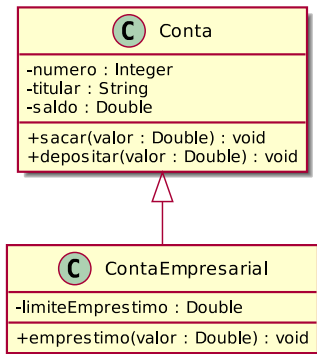
 Conta
-numero : Integer -titular : String -saldo : Double
+sacar(valor : Double) : void +depositar(valor : Double) : void

 ContaEmpresarial
-numero : Integer -titular : String -saldo : Double -limiteEmprestimo : Double
+sacar(valor : Double) : void +depositar(valor : Double) : void +emprestimo(valor : Double) : void

# Herança permite o reuso de atributos e métodos



# Herança permite o reuso de atributos e métodos



Vamos implementar as classes **Conta** e **ContaEmpresarial** e fazer alguns testes.

# Herança e Construtores

- A subclasse `ContaEmpresarial` invoca o construtor da superclasse explicitamente através da instrução

```
super(numero, titular, saldo);
```

# Herança e Construtores

- A subclasse `ContaEmpresarial` invoca o construtor da superclasse explicitamente através da instrução

```
super(numero, titular, saldo);
```

- Construtores não são herdados.
  - A primeira tarefa de qualquer construtor é invocar o construtor da superclasse direta de forma implícita ou explícita.
- Se não houver uma chamada explícita ao construtor da superclasse direta, o compilador invoca o construtor default.
  - *Construtor default*: construtor sem argumentos.
  - **Atenção:** Se a superclasse direta não tiver um construtor default, o compilador lançará uma exceção.



# Modificadores de acesso



# Modificadores de acesso

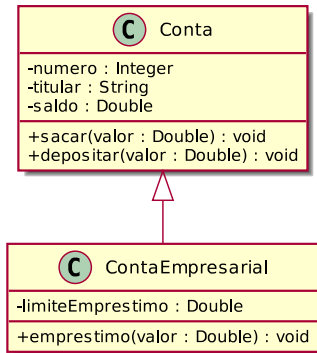
- **public:** Os membros `public` de uma classe são acessíveis em qualquer parte de um programa em que haja uma referência a um objeto da classe ou das subclasses.
- **private:** Membros `private` são acessíveis apenas dentro da própria classe.
- **protected:** Membros `protected` podem ser acessados por membros da própria classe, por membros de subclasses e de classes do mesmo pacote.

# Modificadores de acesso

	Atributos e métodos com visibilidade:			
<b>Classes que têm acesso</b>	<b>private</b>	<b>protected</b>	<b>default</b>	<b>public</b>
A mesma classe	sim	sim	sim	sim
<b>Classes herdeiras</b>	não	sim	sim*	sim
Demais classes no mesmo pacote	não	sim	sim	sim
Demais classes em outro pacote	não	não	não	sim

\* se estiverem no mesmo pacote que a superclasse

# Exemplo de uso do protected



Suponha que para realizar um empréstimo, é descontada uma taxa no valor de 10.0

Isso resulta em erro:

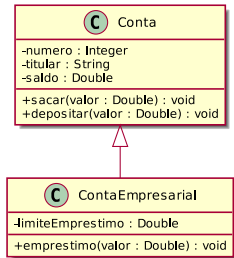
```
public void emprestimo(Double valor) {
    if(valor <= limiteEmprestimo) {
        saldo += valor - 10.0;
    }
}
```

# Upcasting e Downcasting

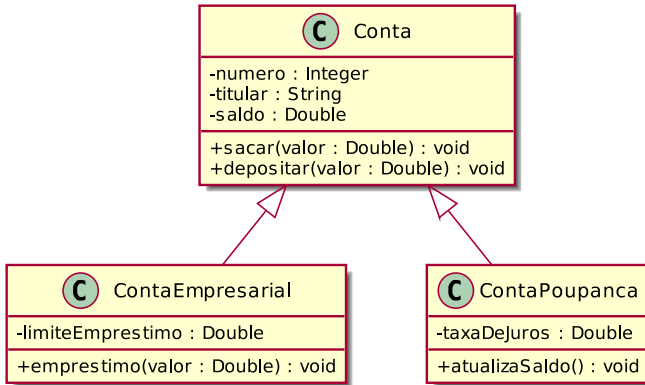


# Upcasting e Downcasting

- Upcasting
  - Casting da subclasse para a superclasse
  - Uso comum: polimorfismo
- Downcasting
  - Casting da superclasse para a subclasse
  - Palavra **instanceof**
  - Uso comum: métodos que recebem parâmetros genéricos (ex.: **equals**)



# Exemplo



# Métodos herdados





# Problemas comuns com métodos herdados

- **Problema 1:** uma subclasse pode herdar métodos que não precisa ou que não deveria ter.
  - **Solução:** Podemos declarar um método como **final** a fim de forçar que este método não seja herdado pelas subclasses de uma superclasse.

# Problemas comuns com métodos herdados

- **Problema 1:** uma subclasse pode herdar métodos que não precisa ou que não deveria ter.
  - **Solução:** Podemos declarar um método como **final** a fim de forçar que este método não seja herdado pelas subclasses de uma superclasse.
- **Problema 2:** o método herdado pode ser necessário na subclasse, mas inadequado.
  - **Solução:** A classe pode **sobrescrever/sobrepor** (*override*) um método herdado para adequá-lo.
  - **Definição:** **Sobreposição de métodos** é a declaração de métodos com a mesma assinatura que métodos de classes ancestrais.
  - **Exemplo:** o método **toString()**

# A anotação @Override

- Para sobrescrever um método de superclasse, uma subclasse deve declarar um método **com a mesma assinatura** do método de superclasse.
  - **Exemplo:** O método toString da classe **Pessoa** sobrescreve o método toString da classe Object.

# A anotação @Override

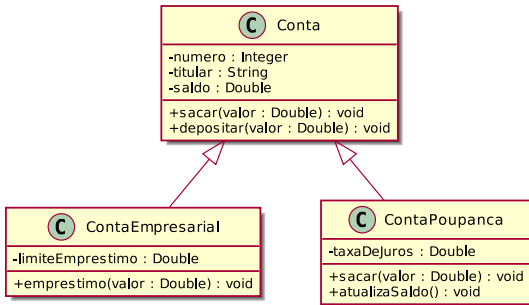
- Para sobrescrever um método de superclasse, uma subclasse deve declarar um método **com a mesma assinatura** do método de superclasse.
  - **Exemplo:** O método toString da classe **Pessoa** sobrescreve o método toString da classe Object.
- Usa-se a anotação **@Override** antes da declaração de um método para indicar que o método sendo declarado deve sobrescrever o método de uma superclasse existente.
  - Essa anotação força o compilador a capturar erros comuns.

# A anotação @Override

**Dica de prevenção de erro:** Embora seja opcional, declare métodos sobrescritos com `@Override` para assegurar em tempo de compilação que suas assinaturas foram definidas corretamente.

Sempre é melhor encontrar erros em tempo de compilação em vez de em tempo de execução.

# Exemplo



Suponha que a operação de saque possui uma taxa no valor de 5.0.  
Entretanto, se a conta for do tipo poupança, esta taxa não deve ser cobrada.

Como resolver isso?

**Resposta:** Sobrescrevendo o método `sacar` na subclasse `ContaPoupanca`

# Regras para sobreposição de métodos

1. A sobreposição de um método em uma subclasse não elimina o acesso ao método de mesma assinatura na classe ancestral – este pode ser acessado, de dentro da classe herdeira, com a palavra-chave `super`, contanto que não tenha sido declarado como `private`.

# Regras para sobreposição de métodos

1. A sobreposição de um método em uma subclasse não elimina o acesso ao método de mesma assinatura na classe ancestral – este pode ser acessado, de dentro da classe herdeira, com a palavra-chave `super`, contanto que não tenha sido declarado como `private`.
2. Métodos declarados em uma subclasse com o mesmo nome mas assinaturas diferentes dos métodos da superclasse não sobrepõem estes métodos.



# Regras para sobreposição de métodos

1. A sobreposição de um método em uma subclasse não elimina o acesso ao método de mesma assinatura na classe ancestral – este pode ser acessado, de dentro da classe herdeira, com a palavra-chave `super`, contanto que não tenha sido declarado como `private`.
2. Métodos declarados em uma subclasse com o mesmo nome mas assinaturas diferentes dos métodos da superclasse não sobrepõem estes métodos.
3. Métodos podem ser sobrepostos com diferentes modificadores de acesso, contanto que os métodos sobrepostos tenham modificadores de acesso menos restritivos.
  - **Exemplo:** podemos declarar um método na superclasse com o modificador de acesso `private` e sobrepor este método em uma subclasse com o modificador de acesso `public`, mas não podemos fazer o contrário.

# Regras para sobreposição de métodos

4. Métodos estáticos declarados em classes ancestrais não podem ser sobrepostos em classes descendentes, nem mesmo se não forem declarados como estáticos.

# Regras para sobreposição de métodos

4. Métodos estáticos declarados em classes ancestrais não podem ser sobrepostos em classes descendentes, nem mesmo se não forem declarados como estáticos.
5. Qualquer método da classe herdeira pode chamar qualquer método da classe ancestral que tenha sido declarado como `public`, `protected` ou sem declaração explícita de modificador. Métodos declarados como `private` não são acessíveis diretamente.

# Regras para sobreposição de métodos

4. Métodos estáticos declarados em classes ancestrais não podem ser sobrepostos em classes descendentes, nem mesmo se não forem declarados como estáticos.
5. Qualquer método da classe herdeira pode chamar qualquer método da classe ancestral que tenha sido declarado como `public`, `protected` ou sem declaração explícita de modificador. Métodos declarados como `private` não são acessíveis diretamente.
6. Métodos declarados como `final` são herdados por subclasses, mas não podem ser sobrepostos (a não ser que a sua assinatura seja diferente).

# A palavra-chave super



# A palavra-chave `super`

- As subclasses podem ter acesso a métodos das superclasses, usando a palavra-chave `super`.
- O acesso a métodos de classes ancestrais é útil para aumentar a reutilização de código.
  - **Atenção:** Métodos `private` não são acessíveis nas subclasses.

# A palavra-chave `super`

- Existem duas maneiras de se reutilizar métodos de classes que não tenham sido declarados como `private`:

# A palavra-chave `super`

- Existem duas maneiras de se reutilizar métodos de classes que não tenham sido declarados como `private`:
  1. Se a execução do método for a mesma para a superclasse e a subclasse, então instâncias da subclasse podem chamar diretamente o método como se fosse delas mesmas — é o caso do método `depositar(valor)` definido na classe `Conta`, e que também pode ser invocado por instâncias de suas subclasses.



# A palavra-chave `super`

- Existem duas maneiras de se reutilizar métodos de classes que não tenham sido declarados como `private`:
  1. Se a execução do método for a mesma para a superclasse e a subclasse, então instâncias da subclasse podem chamar diretamente o método como se fosse delas mesmas — é o caso do método `depositar(valor)` definido na classe `Conta`, e que também pode ser invocado por instâncias de suas subclasses.
  2. Se um método na classe ancestral realiza operações necessárias, é preferível que ele seja chamado, ao invés de duplicarmos o código. Isso reduz a manutenção de código.
    - Vamos ver um exemplo a seguir.

# Exemplo

Suponha que, na classe `ContaEmpresarial`, a regra para saque seja realizar o saque normalmente da superclasse, e só depois descontar mais 2.0.

```
@Override  
public void sacar(Double valor) {  
    super.sacar(valor);  
    saldo -= 2.0;  
}
```

# Regras para uso de super

Algumas regras para uso da palavra-chave `super` para chamar métodos de classes ancestrais como sub-rotinas são:

1. Construtores são chamados pela palavra-chave `super` seguida dos argumentos a serem passados para o construtor entre parênteses. Se não houver argumentos, a chamada deve ser feita como `super()`.

# Regras para uso de super

Algumas regras para uso da palavra-chave **super** para chamar métodos de classes ancestrais como sub-rotinas são:

1. Construtores são chamados pela palavra-chave **super** seguida dos argumentos a serem passados para o construtor entre parênteses. Se não houver argumentos, a chamada deve ser feita como **super()**.
  - **Obs.:** O construtor de uma subclasse **SEMPRE** chama o construtor de uma superclasse, mesmo que a chamada não seja explícita.  
Quando a chamada não é explícita, o construtor chamado é o construtor vazio – se este construtor não estiver definido, haverá um erro de compilação.

# Regras para uso de super

2. Construtores de superclasses só podem ser chamados de dentro de construtores de subclasses e **DEVEM** ser declarados na primeira linha de código do construtor da subclasse.
  - **Obs.:** Métodos não podem chamar construtores de superclasses.

## Regras para uso de super

2. Construtores de superclasses só podem ser chamados de dentro de construtores de subclasses e **DEVEM** ser declarados na primeira linha de código do construtor da subclasse.
  - **Obs.:** Métodos não podem chamar construtores de superclasses.
3. Métodos são chamados pela palavra-chave `super` seguida de um ponto e do nome do método (seguido dos possíveis argumentos).
  - **Exemplo:** `super.toString()`

## Regras para uso de super

2. Construtores de superclasses só podem ser chamados de dentro de construtores de subclasses e **DEVEM** ser declarados na primeira linha de código do construtor da subclasse.
  - **Obs.:** Métodos não podem chamar construtores de superclasses.
3. Métodos são chamados pela palavra-chave `super` seguida de um ponto e do nome do método (seguido dos possíveis argumentos).
  - **Exemplo:** `super.toString()`
4. Somente os métodos e construtores da **superclasse direta** podem ser chamados usando a palavra-chave `super`

## Regras para uso de super

2. Construtores de superclasses só podem ser chamados de dentro de construtores de subclasses e **DEVEM** ser declarados na primeira linha de código do construtor da subclasse.
  - **Obs.:** Métodos não podem chamar construtores de superclasses.
3. Métodos são chamados pela palavra-chave `super` seguida de um ponto e do nome do método (seguido dos possíveis argumentos).
  - **Exemplo:** `super.toString()`
4. Somente os métodos e construtores da **superclasse direta** podem ser chamados usando a palavra-chave `super`
5. Se um método de uma classe ancestral for herdado pela classe descendente, ele pode ser chamado diretamente sem necessidade da palavra `super`.



# Atributos, Métodos e Classes final



# Atributos, Métodos e classes final

- Uma variável ou atributo declarado com o modificador `final` é constante
  - Ou seja, depois de inicializada não pode ser modificada.
- Um método declarado com o modificador `final` não pode ser sobrescrito na classe filha.
- Uma classe declarada com o modificador `final` não pode ser herdada.
  - A declaração de uma classe como `final` efetivamente impede o mecanismo de herança — o compilador não compilará uma classe declarada como herdeira de uma classe final.
  - Exemplo: A classe `String` do java é final

# final — Pra quê?

- **Segurança:** dependendo das regras de negócio, às vezes é desejável garantir que uma classe não seja herdada, ou que um método não seja sobreposto.
  - Geralmente convém acrescentar **final** em métodos sobrepostos, pois sobreposições múltiplas podem ser uma porta de entrada para inconsistências.

# Introdução ao Polimorfismo



**Definição:** Em programação orientada a objetos, **polimorfismo** é a capacidade de uma referência de classe se associar a instâncias de diferentes classes em tempo de execução.

**Definição:** Em programação orientada a objetos, **polimorfismo** é a capacidade de uma referência de classe se associar a instâncias de diferentes classes em tempo de execução.

```
Conta x = new Conta(1020, "Alex", 1000.0);  
Conta y = new ContaPoupanca(1023, "Maria", 1000.0, 0.01);  
x.sacar(50.0);  
y.sacar(50.0);
```

```
Conta x = new Conta(1020, "Alex", 1000.0);  
Conta y = new ContaPoupanca(1023, "Maria", 1000.0, 0.01);  
x.sacar(50.0);  
y.sacar(50.0);
```

**Conta:**

```
public void sacar(Double valor) {  
    saldo -= valor + 5.0;  
}
```

**ContaPoupanca:**

```
public void sacar(Double valor) {  
    saldo -= valor;  
}
```

# Polimorfismo

Usando herança, podemos escrever métodos que recebam instâncias de uma superclasse  $C$ , e os mesmos métodos serão capazes de processar instâncias de qualquer classe que herde da classe  $C$ , já que qualquer classe que herde de  $C$  é um tipo de  $C$ .

```
public static void imprimeDadosConta(Conta c) {  
    System.out.println("Dados da conta:");  
    System.out.println("Número: " + c.getNumero());  
    System.out.println("Titular: " + c.getTitular());  
    System.out.println("Saldo: " + c.getSaldo());  
}
```



# Importante Entender

- A associação do tipo específico com o tipo genérico é feita em tempo de execução (upcasting).
- O compilador não sabe para qual tipo específico a chamada do método `imprimeDadosConta` está sendo feita (ele só sabe que é uma variável do tipo `Conta`)

```
public static void imprimeDadosConta(Conta c) {  
    System.out.println("Dados da conta:");  
    System.out.println("Número: " + c.getNumero());  
    System.out.println("Titular: " + c.getTitular());  
    System.out.println("Saldo: " + c.getSaldo());  
}
```

# Importante Entender

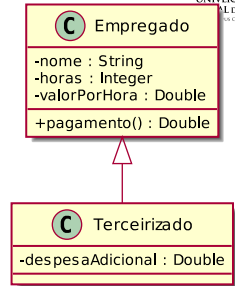
- O compilador não sabe para qual tipo específico a chamada do método **sacar** está sendo feita  
(ele só sabe que é uma variável do tipo **Conta**)

```
Conta x = new Conta(1020, "Alex", 1000.0);  
Conta y = new ContaPoupanca(1023, "Maria", 1000.0, 0.01);  
x.sacar(50.0);  
y.sacar(50.0);
```

# Exercício



- Uma empresa possui funcionários próprios e terceirizados. Para cada funcionário, deseja-se registrar nome, horas trabalhadas e valor por hora. Funcionários terceirizados possuem ainda uma despesa adicional.
- O pagamento dos funcionários corresponde ao valor da hora multiplicado pelas horas trabalhadas, sendo que os funcionários terceirizados ainda recebem um bônus correspondente a 110% de sua despesa adicional.
- Fazer um programa para ler os dados de  $N$  funcionários ( $N$  fornecido pelo usuário) e armazená-los em uma lista. Depois de ler todos os dados, mostrar nome e pagamento de cada funcionário na mesma ordem em que foram digitados.
- Construa o programa conforme o diagrama ao lado. Veja exemplo na próxima página.



Entre o numero de funcionarios: 3

**Dados do Empregado 1:**

Terceirizado (s/n)? n

Nome: Alex

Horas: 50

Valor por hora: 20.00

**Dados do Empregado 2:**

Terceirizado (s/n)? s

Nome: Bob

Horas: 100

Valor por hora: 15.00

Despesa adicional: 200.0

**Dados do Empregado 3:**

Terceirizado (s/n)? n

Nome: Maria

Horas: 60

Valor por hora: 20.00

**PAGAMENTOS:**

Alex - R\$ 1000.00

Bob - R\$ 1720.00

Maria - R\$ 1200.00

Entre o numero de funcionarios: **3**

### Dados do Empregado 1:

Terceirizado (s/n)? **n**

Nome: **Alex**

Horas: **50**

Valor por hora: **20.00**

### Dados do Empregado 2:

Terceirizado (s/n)? **s**

Nome: **Bob**

Horas: **100**

Valor por hora: **15.00**

Despesa adicional: **200.0**

### Dados do Empregado 3:

Terceirizado (s/n)? **n**

Nome: **Maria**

Horas: **60**

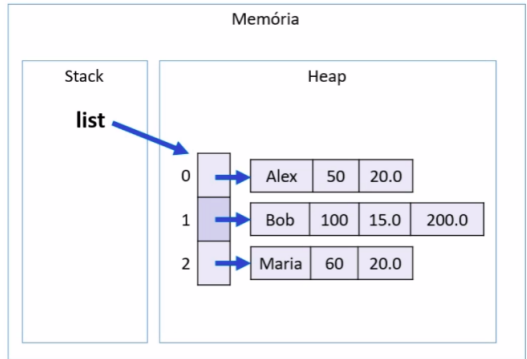
Valor por hora: **20.00**

### PAGAMENTOS:

Alex - R\$ 1000.00

Bob - R\$ 1720.00

Maria - R\$ 1200.00



FIM

