# Herança

Prof. Tiago Sombra

# Herança

- Mecanismo para **estender** as classes existentes adicionando **novos métodos e campos**;

- Ex: **ContaPoupanca**
  - Suponha que você deseja implementar uma classe que irá representar uma conta poupança.
  - **Você já possui** um classe chamada **ContaBancaria** que representa um conta bancária. A classe ContaBancaria **possui vários métodos para manipulação de uma conta bancária;**

# Herança

```
class ContaBancaria{
   double saldo=0.0;
   ContaBancaria(double valor){
        saldo=saldo+valor;
   }
   public void depositar(double valor){
       this.saldo += valor;
   }
   public double getSaldo(double valor){
       return this.saldo;
   }
```

```
 public boolean sacar(double valor)
{
       if(saldo>valor){
            this.saldo -= valor;
            return true;
       }else{
            return false;
      }
   }
}
```

# Herança

- O problema é que **ContaBancaria** não possui nenhum recurso para calcular os juros sobre os depósitos;

- Por isso estamos interessados em implementar a classe **ContaPoupanca**, a qual deverá possuir características mais específicas que um conta bancária;

- Nesse ponto podemos estender a classe **ContaBancaria** herdando todas suas características(atributos e métodos) automaticamente;

# Herança

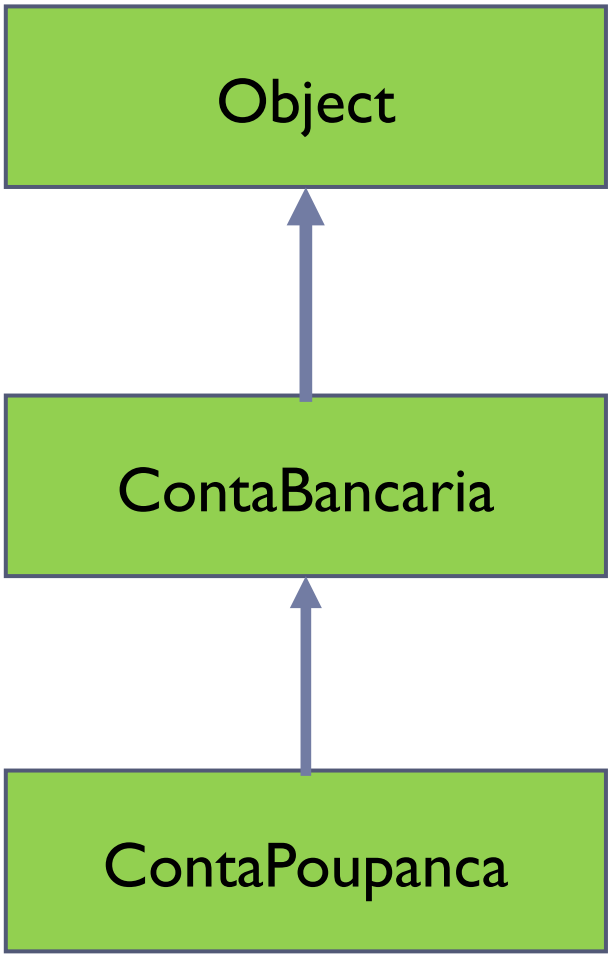
```
class ContaPoupanca extends ContaBancaria
{
    novos métodos
    novos campos de instância
}
//Conta-poupança com 10% de juros
ContaPoupanca fundoUniv = new ContaPoupanca(10);
// OK usar o método de ContaBancaria com o objeto ContaPoupanca
fundoUniv.deposito(500);
```

# Herança

- **Superclasse:** classe mais geral
- **Subclasse:** classe especializada

- Toda classe que não estende especificamente outra classe é uma subclasse da classe **Object;**
- A classe **Object** possui alguns métodos que são comuns a todos os objetos:
  - **toString**

# Herança



# Herança vs Interfaces

- A herança diferencia-se da implementação de uma interface:
  - Subclasses herdam comportamento e o estado da superclasse;
  - Interfaces definem métodos que devem ser implementados, trata-se de um esboço;

- A **vantagem da herança** é a reutilização de código:
  - **Não é necessário replicar** o esforço na implementação dos métodos;
  - Ao estender devemos apenas nos preocupar em **adicionar as novas funcionalidades**

# Herança

```
public class ContaPoupanca extends ContaBancaria
{
   private double taxadeJuros;
   public ContaPoupanca(double taxa){
     taxadeJuros = taxa;
   }
   public void adicionaJuros()
   {
     double juros = getSaldo() * taxadeJuros / 100;
     deposita(juros);
   }
}
```

# Herança

- Por que o método ***adicionaJuros*** chama os métodos ***getSaldo*** e ***deposita*** em vez de atualizar diretamente o campo saldo da superclasse?

# Herança

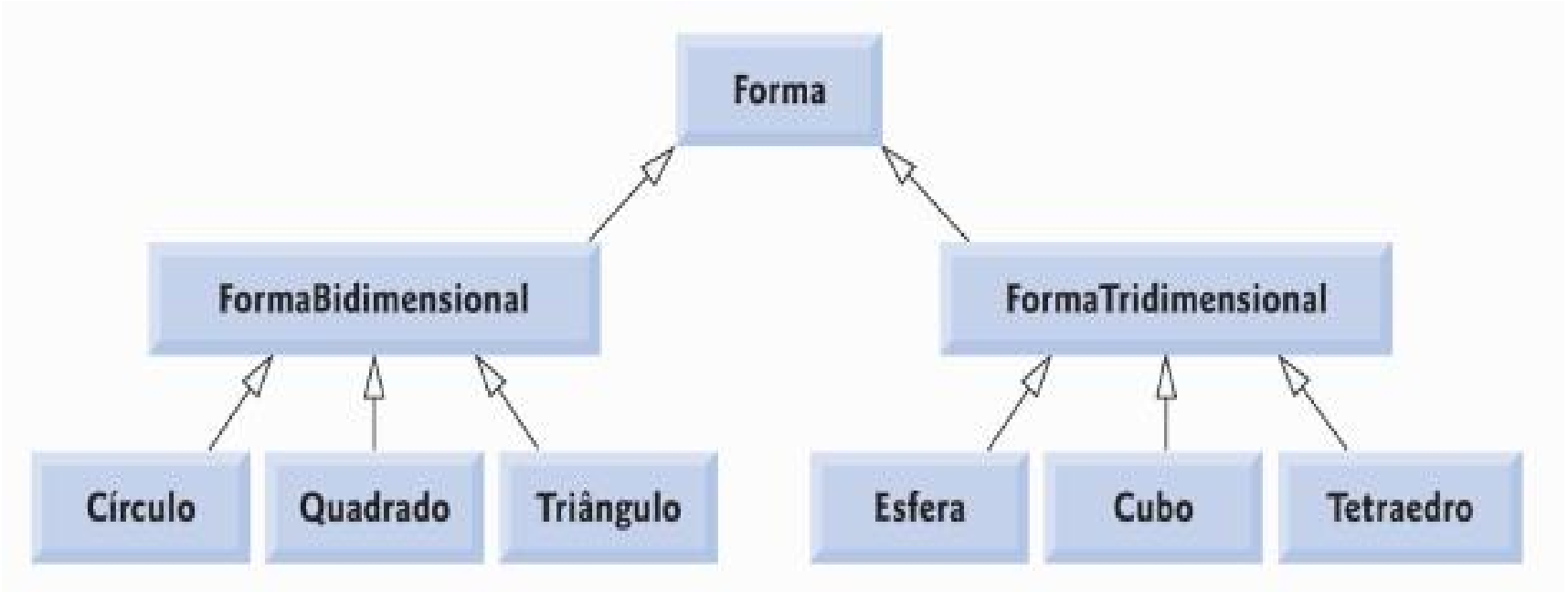
- Por que o método ***adicionaJuros*** chama os métodos ***getSaldo*** e ***deposita*** em vez de atualizar diretamente o campo balance da superclasse?
  - Consequência do encapsulamento;
  - O campo ***saldo*** é definido como ***private*** na superclasse ***ContaBancaria***
  - O método ***adicionaJuros*** é definido na subclasse, logo não tem permissão para acessar um campo privado mesmo na superclasse

# Hierarquia de Classes

- Ao **projetar uma hierarquia** de classes devemos **pensar** nos **recursos e comportamentos comuns** a todas as classes na hierarquia;

- Essas propriedades comuns devem ser implementadas em uma superclasse

# Hierarquia de Classes



# Construindo subclasses

- Para chamar o construtor da superclasse existe um instrução especial “**super**”**:**
  - Seguido dos parâmetros de construção da superclasse
  - Pode ser utilizada seguida por um “**.**”, ou seja, acessando um método da subclasse em qualquer ponto da subclasse;

# Construindo subclasses

```
public class ContaEmergencia extends ContaBancaria{
    public ContaEmergencia(double inicial){
      // Constrói a superclasse
      super(inicial);
      // Inicializa a contagem de transações
      numerodeTransacoes = 0;
   }
}
```

- Se não é realizada uma chamada super() para a superclasse essa será instanciada a partir do construtor padrão;

# Subclasses

- Ocasionalmente podemos instanciar objetos a partir de subclasses:

```
ContaEmergencia contaColegio = new ContaEmergencia(10000);

ContaBancaria umaConta = contaColegio;

Object umObjeto = contaColegio;
```

- Agora as três referências a objeto armazenadas em **contaColegio**, ***umaConta*** **e** ***umObjeto*** referenciam o mesmo objeto do tipo ***ContaEmergencia***

# Subclasses

Por que alguém iria querer conhecer menos sobre um objeto e armazenar uma referência em um campo de objeto de uma superclasse?

# Subclasses

Por que alguém iria querer conhecer menos sobre um objeto e armazenar uma referência em um campo de objeto de uma superclasse?

- Isso pode acontecer se você quiser reutilizar o código que conhece a superclasse, mas não a subclasse.

# Subclasses

Bem mais raro, mas ainda assim possível, podemos partir de um conversão de  uma superclasse para uma subclasse:

```
ContaBancaria conta = (ContaBancaria) object;
```

# Subclasses

Bem mais raro, mas ainda assim possível, podemos partir de um conversão de  uma superclasse para uma subclasse:

```
ContaBancaria conta = (ContaBancaria) object;
```

**Cuidado!!**

Se você estiver errado e **object**, na verdade, referir-se a um objeto de um tipo não-relacionado, uma exceção será lançada.

# Subclasses

- Para se proteger de coerções ruins, você pode utilizar o operador ***instanceof:***
  - Testa se um objeto pertence a um tipo específico;

```
object instanceof ContaBancaria
```

```
if (object instanceof ContaBancaria)
{
 ContaBancaria conta = (ContaBancaria) object;
}
```