

# Introdução à Programação



*Herança*

# Herança

Como aproveitar  
comportamento já  
definido antes?



# Tópicos da Aula

- ◆ Hoje, aprenderemos os fundamentos de Herança
  - Programando sem herança
  - Conceito
  - Importância
  - Utilização em Java
  - A palavra reservada ***super***
  - Restrições de herança em Java
  - Verificação dinâmica de tipos
    - instanceof e Cast
  - A classe *Object*

# Classe de Poupanças: Assinatura

```
public class PoupancaD {  
    public PoupancaD (String n) {}  
    public void creditar (double valor) {}  
    public void debitar(double valor) throws ... {}  
    public String getNumero() {}  
    public double getSaldo() {}  
    public void renderJuros(double taxa) {}  
}
```

# Classes de Poupanças e Contas: Descrições

```
public class PoupancaD {  
    private String  numero;  
    private double saldo;  
    public void creditar (double valor) {  
        saldo = saldo + valor;  
    } //...  
    public void renderJuros(double taxa) {  
        this.creditar(saldo * taxa);  
    }  
}
```

Partes idênticas  
das descrições

Parte diferente  
das descrições

```
public class Conta{  
    private String  numero;  
    private double saldo;  
    public void creditar (double valor) {  
        saldo = saldo + valor;  
    } //...  
}
```

# Classe de Bancos: Assinatura

```
public class BancoD {  
    public BancoD () {}  
    public void cadastrarConta(Conta c) {}  
    public void cadastrarPoupanca(PoupancaD p) {}  
    public void creditarConta(String numero, double valor)  
        throws ... {}  
    public void creditarPoupanca(String numero, double valor)  
        throws ... {}  
  
    // ...  
}
```

Métodos diferentes para manipular  
contas e poupanças

## Classe de Bancos: Descrição

```
public class BancoD {  
    private Conta[] contas;  
    private PoupancaD[] poupancas;  
    private int indiceP, indiceC;  
    //...  
}
```

## Classe de Bancos: Descrição

```
public void cadastrarConta (Conta c) {  
    contas[indiceC] = c;  
    indiceC = indiceC + 1;  
}  
  
public void cadastrarPoupanca (PoupancaD p) {  
    poupancas[indiceP] = p;  
    indiceP = indiceP + 1;  
}
```

## Classe de Bancos: Descrição

```
private Conta procurarConta(String numero) {  
    int i = 0;  
    boolean achou = false;  
    Conta resposta = null;  
    while ((! achou) && (i < indiceC)) {  
        if (contas[i].getNumero().equals(numero))  
            achou = true;  
        else  
            i = i + 1;  
    }  
    if (achou) resposta = contas[i];  
    return resposta;  
}
```

## Classe de Bancos: Descrição

```
private Poupanca procurarPoupanca(String numero) {  
    int i = 0;  
    boolean achou = false;  
    PoupancaD resposta = null;  
    while ((! achou) && (i < indiceP)) {  
        if (poupancas[i].getNumero().equals(numero))  
            achou = true;  
        else  
            i = i + 1;  
    }  
    if (achou) resposta = poupancas[i];  
    return resposta;  
}
```

# Classe de Bancos: Descrição

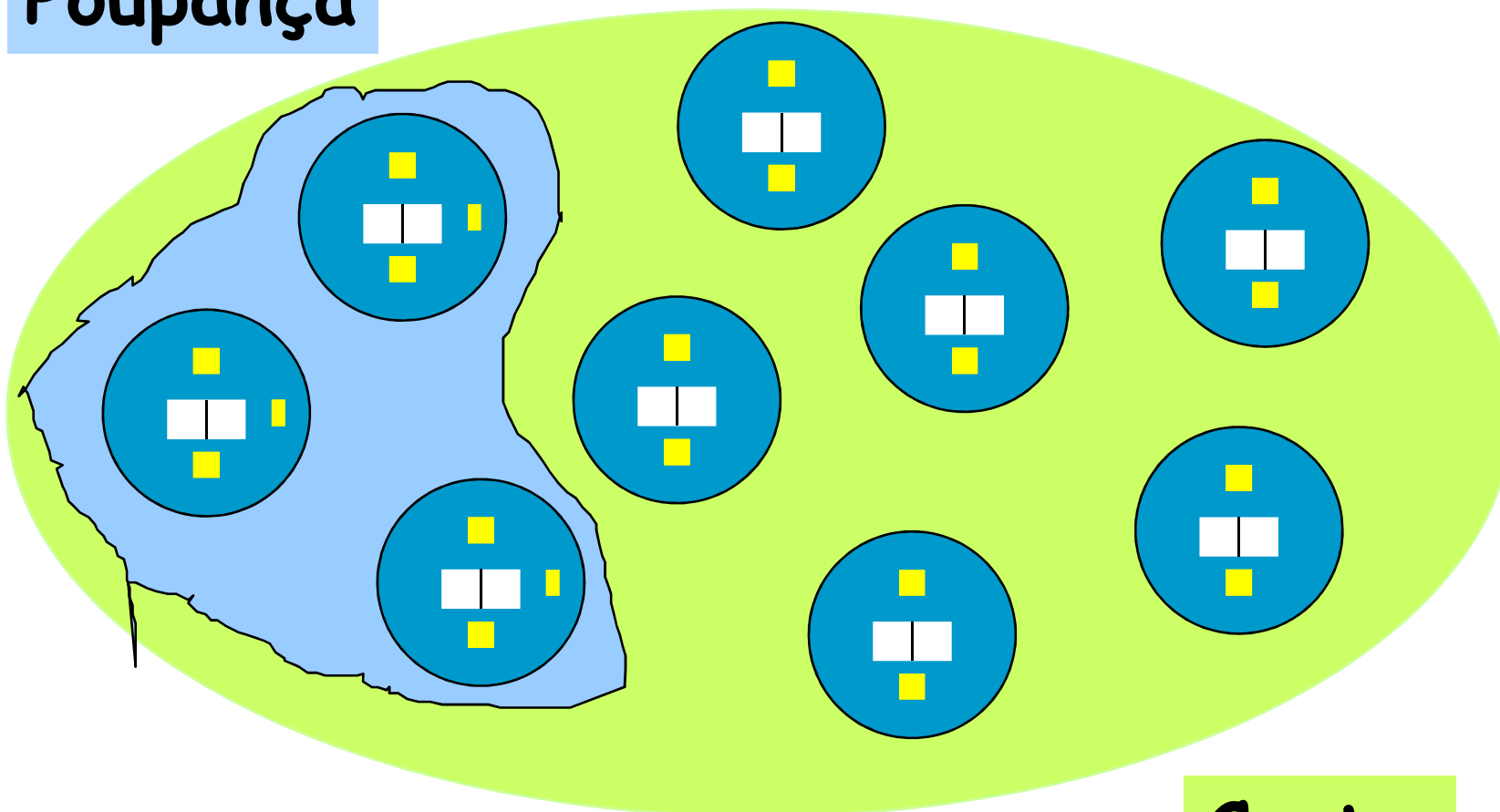
```
public void creditarConta(String numero,double valor)
    throws ... {
    Conta c;
    c = this.procurarConta(numero);
    if (c != null)
        c.creditar(valor);
    else
        //Lança exceção
}
public void creditarPoupanca(String numero,double valor)
    throws ... {
    PoupancaD p;
    p = this.procurarPoupanca(numero);
    if (p != null)
        p.creditar(valor);
    else
        //Lança exceção
}
```

# Problemas

- ◆ Duplicação desnecessária de código:
  - A definição de PoupançaD é uma simples extensão da definição de Conta
  - Clientes de Conta que precisam trabalhar também com PoupançaD terão que ter código especial para manipular poupanças
- ◆ Falta refletir relação entre tipos do “mundo real”

# Subtipos e Subclasses

Poupança



Conta

# Herança

- ◆ Necessidade de estender classes
  - Alterar classes já existentes e adicionar propriedades ou comportamentos para representar outra classe de objetos
  - Criar uma hierarquia de classes que “herdam” propriedades e comportamentos de outra classe e definem novas propriedades e comportamentos

# Herança

- ◆ **Herança** permite que novas classes possam ser derivadas de classes existentes
- ◆ A classe existente é chamada de classe pai ou superclasse
- ◆ A classe derivada é chamada de classe filha ou subclasse
- ◆ Subclasse *herda* as características da superclasse
  - Herda os atributos e métodos
- ◆ Estabelece a relação de **é-um**
  - A subclasse é uma versão especializada da superclasse

# Importância de Herança

## ◆ Comportamento

- Objetos da subclasse comportam-se como os objetos da superclasse

## ◆ Substituição

- Objetos da subclasse podem ser usados no lugar de objetos da superclasse

## ◆ Reuso de Código

- Descrição da superclasse pode ser usada para definir a subclasse

## ◆ Extensibilidade

- algumas operações da superclasse podem ser redefinidas na subclasse

# Classe de Poupanças: Assinatura

Palavra reservada para  
indicar herança

```
public class Poupanca extends Conta {  
    public Poupanca (String numero) {}  
    public void renderJuros(double taxa) {}  
}
```

## *Extends*

- ◆ *Subclasse extends superclasse*
- ◆ Mecanismo para definição de herança e subtipos
- ◆ Herança simples: só se pode herdar uma classe por vez
- ◆ Pode-se utilizar *extends* com interfaces, neste caso uma interface pode estender várias interfaces (veremos depois)

# Classes de Poupanças: Descrição

```
public class Poupanca extends Conta {  
  
    public Poupanca (String numero) {  
        super (numero);  
    }  
  
    public void renderJuros(double taxa) {  
        this.creditar(this.getSaldo() * taxa);  
    }  
}
```

Utiliza-se o construtor  
da superclasse

# Referência super

- ◆ Construtores não são herdados
  - Embora, freqüentemente, precisamos do construtor da superclasse para a definição dos construtores das subclasses
  - Necessários para inicializar os atributos que são herdados da superclasse
- ◆ A referência **super** pode ser utilizada para se referir à superclasse
  - Freqüentemente utilizada para invocar o construtor da superclasse

# Referência super

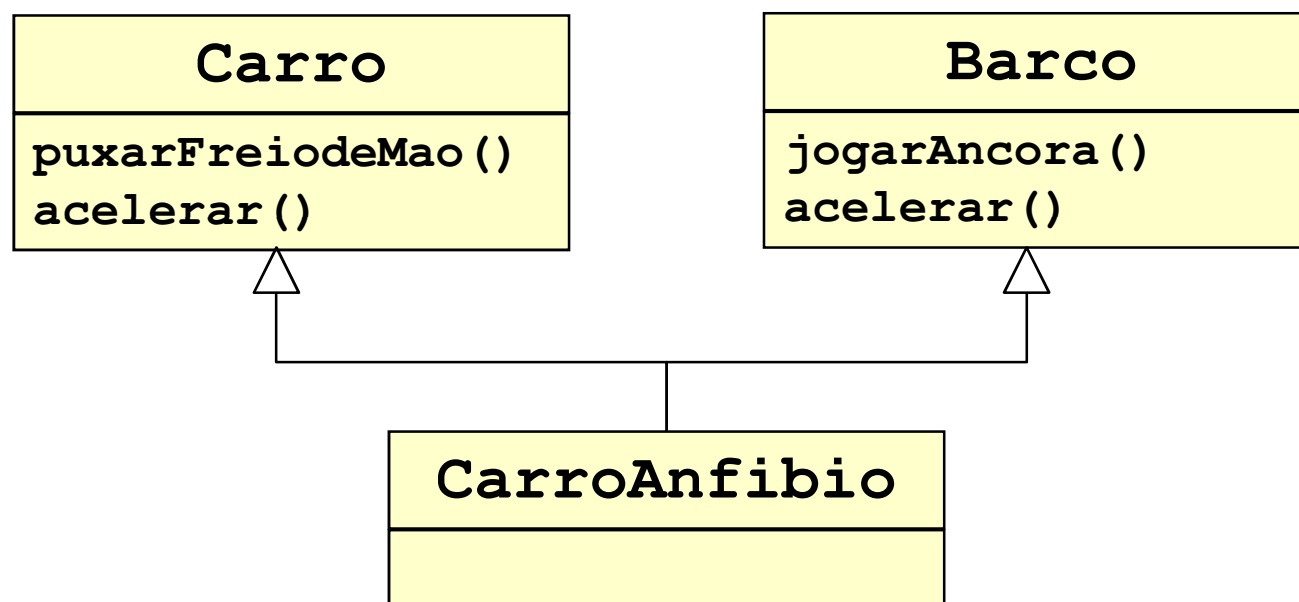
- ◆ Construtor da subclasse é responsável por chamar construtor da superclasse
- ◆ A referência super pode ser utilizada para referenciar atributos e métodos da superclasse
  - Especialmente importante no caso de redefinições de métodos

```
public class Pai {  
    public void x() {  
        ...  
    }  
}
```

```
public class Filho extends Pai {  
    // Redefinindo x  
    public void x() {  
        super.x();  
        ...  
    }  
}
```

# Herança Múltipla

- ◆ Herança múltipla permite que uma classe seja derivada de mais de uma classe



Qual versão do método acelerar é herdada por CarroAnfibio?

# Herança Múltipla e Java

- ◆ Herança múltipla faz com que subclasses herdem atributos e métodos das suas superclasses
  - Pode haver colisões de nomes de atributos e métodos
- ◆ Java não dá suporte a herança múltipla
  - Uma classe só pode ter um pai
- ◆ Para aplicações que são modeladas usando herança múltipla, Java permite a simulação deste fenômeno com o uso de interfaces (veremos depois)
  - Herança múltipla de assinaturas

## Subtipos: Substituição

- ◆ Herança permite a substituição do supertipo pelo subtipo no código
  - Onde é permitido o supertipo, o subtipo pode ser utilizado

...

```
Conta conta;  
conta = new Poupanca("21.342-7");  
conta.creditar(500.87);  
conta.debitar(45.00);  
console.println(conta.getSaldo());
```

...

## Subtipos: Substituição

```
...  
Poupanca p;  
p = new Conta("21.342-7");  
p.creditar(500.87);  
p.debitar(45.00);  
console.println(p.getSaldo());  
...0
```

**Cuidado! A recíproca não é verdadeira.**

## Substituição e Casts

- ◆ Nos contextos onde objetos do tipo **Conta** são usados pode-se usar objetos do tipo **Poupanca**
- ◆ Nos contextos onde objetos do tipo **Poupanca** são usados pode-se usar variáveis do tipo **Conta** que armazenam referências do tipo **Poupanca**, desde que se faça o uso explícito de ***casts***

## Subtipos: Utilizando Casts

```
...  
Conta conta;  
conta = new Poupanca ("21.342-7");  
...  
conta.renderJuros (0.01);  
((Poupanca) conta).renderJuros (0.01);  
...
```

Gera erro de  
compilação

Cast

# Cast e Conversão de Dados

- ◆ Às vezes é conveniente converter um tipo de dado em outro
  - Ex: tratar um inteiro como um valor real
- ◆ Um dos modos de fazer isto é a utilização de **cast**

```
int total = 4;  
float resultado = (float)total;
```

- ◆ `total` não é convertido, apenas o valor que será armazenado em `resultado` será
- ◆ Cast em variáveis do tipo referência **NÃO** fazem nenhum tipo de conversão do objeto referenciado pela referência
  - Apenas informam ao compilador que o objeto referenciado pela variável, naquele momento, é do tipo declarado pelo `cast`

## Subtipos: Verificação Dinâmica com `instanceof`

- ◆ A palavra `instanceof` permite a verificação de tipos em tempo de execução (verificação dinâmica)

```
...  
Conta c = this.procurar("123.45-8");  
if (c instanceof Poupanca)  
    ((Poupanca) c).renderJuros(0.01);  
else  
    console.println("Poupança inexistente!")  
...
```

# Verificação Dinâmica de Tipos

## ◆ *Casts* e `instanceof`:

- `((Tipo) variável)`
- *variável* `instanceof` Tipo
- O tipo de *variável* deve ser supertipo de Tipo
- *Casts* geram exceções quando `instanceof` retorna *false*
- *Casts* são essenciais para verificação estática de tipos (compilação)

**Casts devem ser utilizados com cuidado**

# Classe de Bancos: Assinatura

```
public class Banco {  
    public Banco () {}  
    public void cadastrar(Conta conta) {}  
    public void creditar(String numero, double valor) {}  
    public void debitar(String numero, double valor) {}  
    public double getSaldo(String numero) {}  
    public void transferir(String contaOrigem,  
                           String contaDestino,  
                           double valor) {}  
}
```

## Subtipos: Substituição

...

```
Banco banco = new Banco();
```

```
banco.cadastrar(new Conta("21.345-7"));
```

```
banco.cadastrar(new Poupanca("1.21.345-9"));
```

```
banco.creditar("21.345-7", 129.34);
```

```
banco.transferir("21.345-7", "1.21.345-9", 9.34);
```

```
System.out.print(banco.getSaldo("1.21.345-9"));
```

...

Podemos  
cadastrar contas

Podemos cadastrar  
poupanças

**Mesmo método pode ser aplicado a Conta  
e Poupanca**

# Classe Object

- ❖ A classe `Object` é definida na biblioteca padrão de Java (pacote `java.lang`)
- ❖ Todas as classes de Java são derivadas de `Object`
- ❖ Se uma classe não explicita que é subclasse de uma outra classe, então ela é filha de `Object`
- ❖ Esta classe tem alguns métodos úteis que são herdados por todas as classes
  - `Ex: toString()`
    - Retorna o nome da classe e algumas outras informações
- ❖ Toda a vez que uma classe define o método `toString()`, na verdade esta classe está redefinindo o método herdado de `Object`

## Resumindo ...

### ◆ Herança

- Necessidade de Herança
- Herança em Java
- A palavra reservada ***super***
- Restrições de herança em Java
- Verificação dinâmica de tipos
  - instanceof e Cast
- A classe *Object*