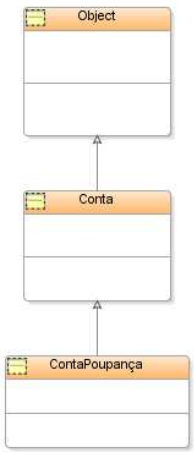
Herança

Java

O mecanismo fundamental da POO conhecido como “herança” de baseia no fato de que podemos criar classes a partir de uma classe já existente. Essas classes, chamadas de classes derivadas, são uma extensão da classe já existente e herdam automaticamente atributos e métodos dessa classe.

As classes derivadas de uma classe A são chamadas de **“subclasses de A”** e a classe A é chamada de **“superclasse”** das suas derivadas. Subclasses podem ser superclasses de outras classes e assim por diante, o mecanismo de herança então se propaga pelos sucessivos níveis de classes derivadas. Java considera uma classe de nome **“Object”** como superclasse de todas as outras classes, como se fosse a classe mais alta na hierarquia de classes. Se uma classe não é explicitamente a extensão de nenhuma outra, ela é derivada da classe “Object”.

Vamos supor um exemplo de uma classe genérica que modela contas bancárias de nome “Conta”, superclasse de uma outra de nome “ContaPoupanca”. Teríamos então o esquema abaixo, segundo a notação de UML:



Para mantermos o exemplo simples, vamos imaginar que em nosso modelo uma conta bancária é um objeto que tem um só atributo, seu saldo, e dois métodos de serviço, um para depósito e outro para saque, além de métodos de acesso.

O código Java dessa classe, considerando dois construtores, seria:

```
class Conta
{
  // atributo saldo
  private double saldo;
  // construtores
  public Conta() { saldo = 0; }
  public Conta(double s) { saldo= s; }
  // acesso
  public double getSaldo() { return saldo; }
  // servicos
  public void deposito(double v) { saldo += v; }
  public void saque(double v) { saldo -= v; }
}
```

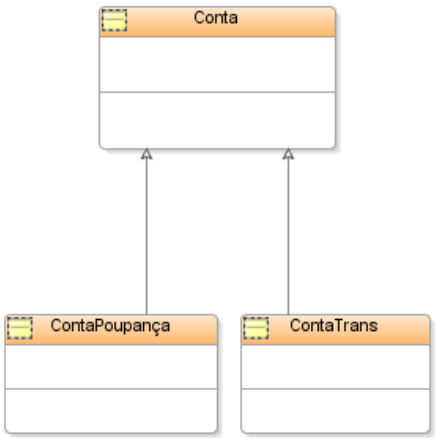
Automaticamente a classe Conta é considerada derivada de Object, mas para declararmos nossas classes derivadas de outras usamos a palavra ***extends*** e o nome da superclasse. Supondo nosso modelo de classe de conta de poupança com um atributo para a taxa de juros, um método que reajusta o saldo com os juros, métodos de acesso e um construtor, teríamos:

```
class ContaPoupanca extends Conta
{
  // atributo
  private double taxa;
  // construtor
  public ContaPoupanca(double t)
  { taxa = t; }
  // acesso
  public double getTaxa()
  { return taxa; }
  public void setTaxa(double t)
  { taxa = t; }
  // servico
  public void reajuste()
  {
    double valor = getSaldo() * taxa / 100;
    deposito(valor);
  }
}
```

Uma classe derivada pode ter a redefinição de um método por ela herdado, exatamente com o mesmo nome e parâmetros. Neste caso a classe derivada terá duas versões do método, uma herdada e a outra criada por ela mesma. Para acessar a versão herdada será necessário o uso de um prefixo ***super*** antes do nome do método. Isso vale também para atributos homônimos herdados.

Vamos ver isso no exemplo de uma outra subclasse de Conta, uma conta que cobra um valor fixo por transação efetuada (depósito ou saque) além de um certo número de transações gratuitas. Essa cobrança não é feita no momento da transação, mas sim por um método de serviço que pode ser acionado periodicamente e reiniciará o contador de transações. Enquanto o contador de transações será um atributo normal desse tipo de conta, o valor de desconto por transação e o número de transações gratuitas serão atributos estáticos, compartilhados por todos os objetos da classe, e cujos valores serão fixos (constantes), o que se consegue com o atributo final especificado na definição desses dados.

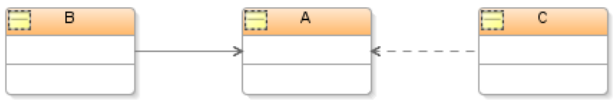
O diagrama com essa nova classe, desconsiderando a classe Object, seria:



```
class ContaTrans extends Conta
{
  // atributos
  private static final double CUSTOTRANS = 1.0;
  private static final int TRANSGRATIS = 2;
  private int cntTrans;
  // construtor
  public ContaTrans(double s)
  {
    super(s); // construtor da superclasse
    cntTrans = 0;
  }
  // servicos
  public void cobranca()
  { if (cntTrans > TRANSGRATIS)
    { double valor = CUSTOTRANS * (cntTrans – TRANSGRATIS);
      super.saque(valor);
      cntTrans = 0;
    }
  }
  public void deposito(double v)
  { cntTrans++;
    super.deposito(v);
  }
  public void saque(double v)
  { cntTrans++;
    super.saque(v);
  }
}
```

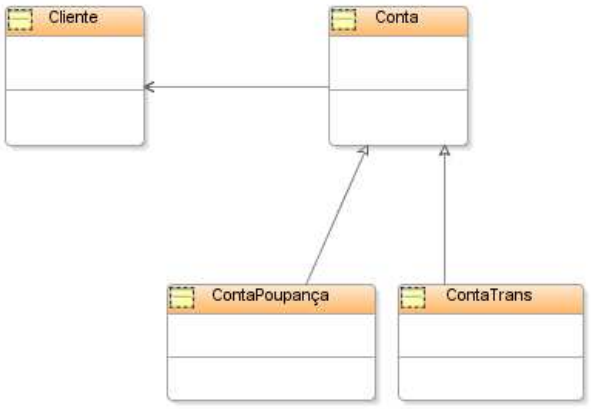
Em Java uma classe só pode ser derivada diretamente de uma única classe, o que se chama de “herança simples”. Em C++, por exemplo, uma classe pode ser derivada de mais de uma classe, o que se chama “herança múltipla”. A restrição em Java tem o objetivo de simplificar e otimizar o processamento dos programas.

Nos programas orientados a objetos, as classes podem estar relacionadas de outras formas além de serem derivadas umas das outras. Podemos ter classes “associadas” a outras classes sem que sejam derivadas delas. Uma classe B pode, por exemplo, ter um atributo (um objeto) que é uma **referência** a um objeto de uma classe A. Dizemos neste caso que a classe B está associada à classe A e depende de A. Por outro lado, uma classe C pode não ter atributos da classe A mas pode ter métodos que recebem parâmetros que são referências a objetos da classe A. Dizemos então neste segundo caso que a classe C **depende** de A. Nos diagramas UML poderiam essas classes estar assim representadas:



Vamos incrementar nosso modelo associando a classe Conta a uma classe Cliente, com um único atributo, o nome do cliente, um construtor e métodos de acesso. A classe Conta passará a ter um atributo que será uma referência a um objeto da classe Cliente, e seus construtores deverão receber esse objeto cliente como parâmetro.

O nosso diagrama pode ser então desenhado:



E as classes Cliente e Conta ficariam:

```
class Cliente
{
  // atributo
  private String nome;
  // construtor
  public Cliente(String n) { nome = n; }
  // acesso
  public void setNome(String n) { nome = n; }
  public String getNome() { return nome; }
}

class Conta
{
  // atributos
  private double saldo;
  private Cliente cli;
  // construtores
  public Conta() { saldo = 0; }
  public Conta(double s) {saldo= s; }
  // acesso
  public Cliente getCliente() { return cli; }
  public void setCliente(Cliente c) { cli = c; }
  public double getSaldo() { return saldo; }
    // servicos
  public void deposito(double v) { saldo += v; }
  public void saque(double v) { saldo -= v; }
}
```