Fernando Lozano
Consultor Independente
fernando@lozano.eti.br
www.lozano.eti.br

Sun Java Certified Programmer
IBM Visual Age Certified Associate Developer
MCSE - MCSD
RHCE - LPICP - SCP

Objetivo

- Entender os principais conceitos de Programação Orientada a Objetos (POO) e a sua aplicação no desenvolvimento de aplicações em Java

Pré-Requisitos

- Conhecimentos da sintaxe e tipos de dados básicos da linguagem Java, vista sob o ponto de vista da Programação Estruturada
(ver o módulo “De Pascal Para Java”)

Esquentando os Motores

- Escreva um programa em Java que, dado um número de até quatro algarismos (passado na linha de comando), escreva na tela a sua representação por extenso
- **> java Extenso 1234**
  Mil duzentos e trinta e quatro

Por Que POO?

- Gerenciar a complexidade crescente dos sistemas sendo construídos nas empresas, viabilizando o trabalho conjunto de grandes equipes e o cumprimento de metas de qualidade, prazo e orçamento
- Aumentar a produtividade dos analistas e programadores pela reutilização de código pronto e depurado escrito em sistemas anteriores ou adquiridos no mercado

Paradigmas de Programação

Programação Orientada a Eventos

Programação Orientada a Objetos

Programação Estruturada

Programação Procedural

Paradigmas de Programação

Estilo de Interface com o Usuário

Conceitos de Negócio (Entidades, Atividades, etc)

Lógica de Programação

Funcionamento da Máquina

POO e Java

- 1. Conceitos de POO
- 2. Classes e Objetos em Java
- 3. Interfaces
- 4. Relacionamentos entre classes
- 5. Outros recursos OO de Java
  - 5.1. Excessões
  - 5.2. Pacotes
  - 5.4. Reflexão
- 6. Metodologia para desenvolvimento OO

1. Conceitos de POO

- Objeto
- Encapsulamento
- Classe
- Instância
- Objetos como caixas-pretas
- Herança e Tipos
- Polimorfismo
- Sobreposição e sobrecarga
- Construtores

Objeto

- Um objeto representa qualquer coisa do mundo real que seja manipulada pelo nosso programa, ou então representa blocos de construção do próprio programa
  - O programa
  - Uma Conta-corrente
  - Um cliente
  - Uma janela
  - Um botão

Objeto

- Assim como as coisas no mundo real, os objetos tem “estado” e “comportamento”
  - Estado são informações sobre o objeto, como a sua cor, seu peso, o saldo da conta-corrente, etc
  - Comportamento são coisas que podem ser feitas com ou pelo objeto, como depositar em uma conta-corrente ou mudar a cor de uma janela
- Objetos também tem uma identificação, de modo que possamos diferenciar entre vários objetos semelhantes
  - O usuário “Fernando” e o usuário “Ricardo”

Encapsulamento

- Dizemos que um objeto em um programa “encapsula” todo o estado e o comportamento, de modo que podemos tratar o objeto como uma coisa só, em vez de ter várias variáveis e subprogramas isolados entre si
- É por isso que um programa em Java costuma ser formado por vários objetos em vez de apenas um
- Os objetos encapsulados corretamente são independentes da programa onde eles são utilizados

Classe

- Vários objetos semelhantes possuem o mesmo tipo de informação em seu estado e tem o mesmo comportamento
- Dizemos que estes objetos pertencem a uma mesma classe, ou são de um certo tipo de classe
- Assim, uma classe **ContaCorrente** pode definir todo o estado e comportamento que uma conta-corrente pode ter

Estado e Comportamento

- O *estado* de um objeto é toda a informação que ele carrega
  - Armazenamos este estado nos *atributos* do objeto
  - Estes atributos são chamados ainda de *variáveis de instância* da classe
- O comportamento de um objeto é tudo o que ele sabe fazer, e tudo o que pode ser feito com ele
  - O comportamento é definido pelos *métodos* da classe

Instância

- Dizemos que um objeto em particular de uma dada classe é uma instância desta classe
- Em Java, definimos classes e criamos variáveis que referenciam instâncias de uma classe
- O estado do objeto/instância é salvo em variáveis da instância, e o comportamento é definido por métodos na classe
- Criamos instâncias de uma classe utilizando o comando **new**

Classe x Instância

- A classe Funcionário
- A classe Carro de Passeio
- A classe Nota Fiscal

- Os funcionários Fernando e Ricardo
- O Corsa Verde placa LCW 1666
- A Nota Fiscal #21234 referente a um gaveteiro de escritório, 4 puxadores cromados e uma cadeira

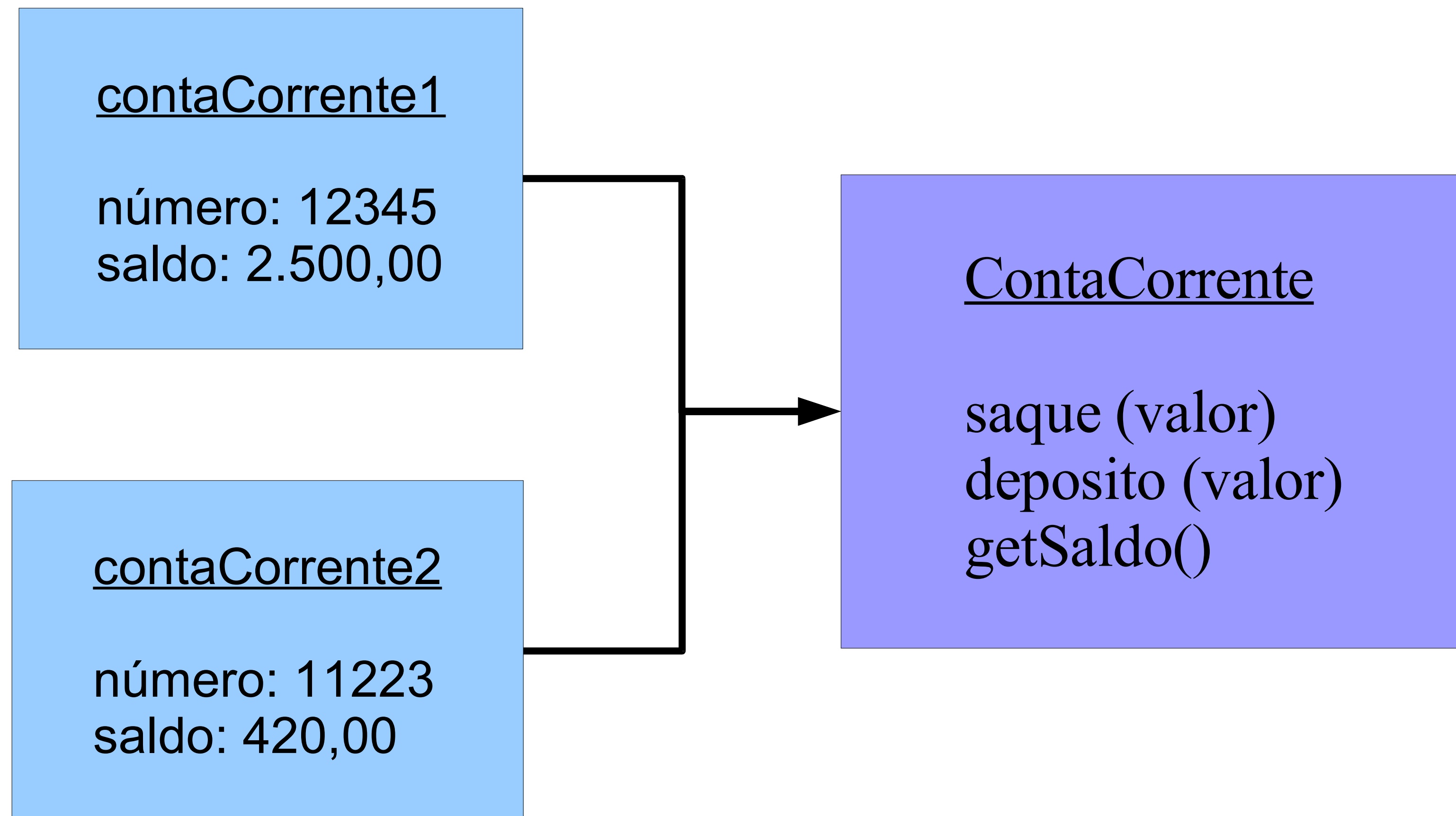
Classes em Programas de Computador

- Em um programa Orientado a Objetos, escrevemos (programamos) definições de classes em vez de programas e subrotinas
- Apenas durante a execução deste programa são criadas instâncias das classes que foram definidas
- Cada instância possui o seu próprio conjunto de atributos, independente de outras instâncias da mesma ou de outras classes
- Todas as instâncias de uma mesma classe compartilham as mesmas definições de métodos

Objetos, Classes e Instâncias

- Quando falamos de objeto, nos referimos tanto ao conceito geral de um tipo de objetos (classe) quanto a um objeto em particular de uma dada classe (instância)
- Nas linguagens de programação OO, tanto classes quanto instâncias podem ser manipuladas como objetos, utilizando a mesma sintaxe para acesso a métodos e atributos
- Os objetos de classe estão sempre disponíveis, enquanto que as instâncias devem ser criadas (devemos *instanciar* os objetos)

Classes e Instâncias no Computador



Objetos como Caixas-Pretas

- Uma boa prática é tratar os objetos como se fossem caixas-pretas. Isto significa que não permitimos a nada fora do objeto modificar o seu estado
- Desta forma evitamos que erros em uma parte do programa tenham consequências em outras partes do mesmo
- Este é o princípio do *ocultamento de informações*

Objetos como Caixas-Pretas

- Em Java, isto significa que nenhum código fora do objeto pode modificar os atributos (variáveis) internas ao objeto.
- O estado do objeto é modificado como consequência da execução dos seus métodos
- É comum criar métodos do tipo *setAtributo* e *getAtributo* quando tudo o que queremos é salvar ou recuperar uma informação dentro de um objeto

A Classe ContaCorrente

- Os arquivos **ContaCorrente.java** e **Caixa.java** demonstram como:
  - Definir uma classe
  - Instanciar objetos desta classe
  - Utilizar métodos destas instâncias

ContaCorrente.java

```
Class ContaCorrente
{
    private double saldo = 0;

    public void deposito (double valor) {
        saldo += valor;
    }

    public void saque (double valor) {
      saldo -= valor;
    }

    public double getSaldo () {
      return saldo;
    }
}
```

A Classe ContaCorrente

- Possui apenas um atributo: **saldo**
- O valor deste atributo pode ser recuperado pelo método **getSaldo**
- O saldo de uma conta-corrente é modificado em consequência das operações de **saque** e **depósito** executados sobre ele, daí os métodos com o mesmo nome
- Embora seja permitido, evite utilizar acentos em nomes de classes, métodos e atributos

Caixa.java

```
class Caixa
{
    static public void main (String[] args) {
        ContaCorrente conta = new ContaCorrente();
        System.out.println ("Saldo inicial = " + conta.getSaldo ());
        conta.deposito (100.00);
        System.out.println ("Novo saldo = " + conta.getSaldo ());
    }
}
```

O Programa Caixa

- É definido pela classe **Caixa**, que contém apenas o método estático **main**
- Métodos estáticos (**static**) são ditos *métodos de classe*, podemos executa-los (invoca-los) sem antes instanciar objetos desta classe
- O método **Caixa.main** apenas instancia um objeto da classe **ContaCorrente**, consulta o seu saldo e realiza um depósito, verificando então o novo saldo; não passa de um programa de testes, mas um sistema real teria uma classe "Caixa" responsável pela interação com o usuário

Exercício

- Modifique o programa **Caixa.java** para criar várias instâncias de **ContaCorrente** e realizar depósitos de valores diferentes em cada uma, comprovando que cada instância mantém o seu próprio estado
- Mova o método **main** da classe **Caixa** para a classe **ContaCorrente**, de modo que ela mesma contenha um “programa principal”, ou melhor, um método de teste que demonstre o seu uso e funcionalidade

Discussão

- Por que não declarar o atributo saldo como público (**public**) e realizar aritmética diretamente com o seu valor? Isto não seria mais simples?
- De quem (qual método, qual classe) seria a *responsabilidade* de verificar se existe saldo suficiente em uma conta para a realização de um saque?

Classes Também Representam Processos

- Em geral se entende facilmente a idéia de *objetos informacionais*, isto é, objetos que representam conceitos ou coisas sobre as quais armazenamos informações em nossos sistemas
- Mas para se obter resultados mais efetivos das técnicas de OO, temos que explorar principalmente os *objetos de atividade*, isto é, objetos que representam atividades, processos ou transações realizadas pelo sistema

Transferencia.java

```
class Transferencia
{
    private ContaCorrente origem;
    private ContaCorrente destino;
    private double valor;

    // métodos setXXX para os atributos
    // origem, destino e valor, exemplo:

    public void setOrigem (ContaCorrente origem) {
        this.origem = origem;
    }
    public ContaCorrenet getOrigem () {
        return origem;
    }

// continua...
```

Transferencia.java

```
// continuação...

    public void realiza () {
       origem.saque (valor);
       destino.deposito (valor);
    }
}
```

A Palavra-Chave “this”

- Quando um método de instância necessita obter uma referência à instância em particular que o está executando, ele pode usar a referência pré-definida **this**
- Há duas situações principais para o uso do this:
  - Eliminar conflitos de nome, como entre o argumento “origem” e o atributo “origem” da classe Transacao
  - Passar referências a si mesmo para outros objetos, de modo similar ao **Me** do Visual Basic ou ao **Self** do Delphi

Método de testas para a classe Transferencia

```
static public void main (String[] args) {
    ContaCorrente contaOrigem = new ContaCorrente();
    contaOrigem.deposito (1000.00);
    ContaCorrente contaDestino =  new ContaCorrente();
    contaDestino.deposito (2000.00);
    System.out.println ("Antes da transferencia");
    System.out.println ("Saldo origem = " + contaOrigem.getSaldo ());
    System.out.println ("Saldo destino = " + contaDestino.getSaldo ());
    Transferencia pagamento = new Transferencia ();
    pagamento.setOrigem (contaOrigem);
    pagamento.setDestino (contaDestino);
    pagamento.setValor (250.00);
    pagamento.realiza ();
    System.out.println ("Depois da transferencia");
    System.out.println ("Saldo origem = " + contaOrigem.getSaldo ());
    System.out.println ("Saldo destino = " + contaDestino.getSaldo ());
}
```

Conceitos x Código

- Em situações simples como o último exemplo, podemos ser tentados a incluir a lógica da operação diretamente em nosso programa, ou a simplesmente definir um método que realize a operação, sem criar a classe **Transferencia**.
- Entretanto o conceito de uma transferencia de valores entre duas contas é importante o suficiente no dia-a-dia de um banco para justificar o seu encapsulamento em uma classe isolada
- *Classes e objetos devem representar conceitos, e não apenas serem "sacos de código" em Java*

Esteja Preparado para o Futuro

- Uma aplicação bancária real teria várias outras funcionalidades associadas a uma classe "Transferencia", por exemplo:
  - Registro histórico, para fins de auditoria e emissão do extrato para o cliente
  - Compensação de fundos entre bancos diferentes
  - Restrições sobre operações (ex: caixa eletrônico ou na agência; horário comercial ou noturno)
- A classe Transferencia pode ser expandida sem efeitos colaterais negativos no restante do sistema

Discussão

- Quando definir métodos e quanto definir classes de atividade?
  - Métodos de uma classe devem modificar o estado de uma única instância desta desta classe
  - Operações que envolvam modificar o estado de várias instâncias diferentes (mesmo que sejam da mesma classe) devem ser encapsulados em classes de atividade
  - As classes de atividade podem validar se receberam toda a informação correta antes de realizar a atividade e são mais amigáveis a ferramentas visuais

Herança ou Especialização

- Uma classe pode ser *derivada* de outra classe, e desta forma *herdar* tanto seu estado quanto o seu comportamento
- A criação de *subclasses* (classes derivadas de uma *superclasse*) permite o aumento incremental da funcionalidade dos nossos objetos ou sua *especialização*
- Se precisarmos de um objeto que faça o mesmo que outro objeto faz e “mais alguma coisa”, aproveitamos o código que já está pronto e testado, definindo uma subclasse contendo apenas as novidades

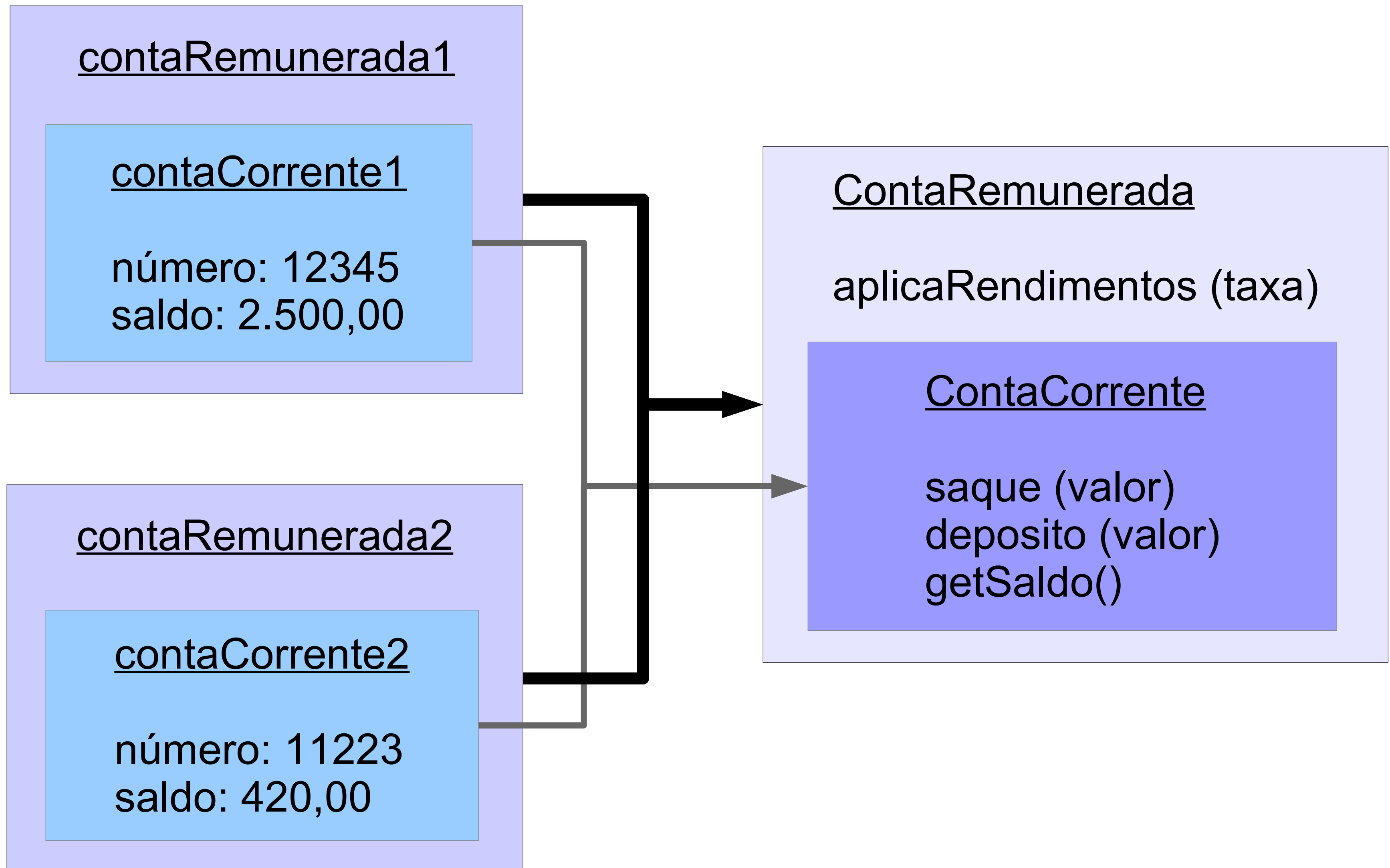
ContaRemunerada.java

```
class ContaRemunerada extends ContaCorrente
{
    public void aplicaRendimentos (double taxa) {
        deposito (getSaldo () * taxa);
    }

    // método main para teste da classe

    static public void main (String[] args) {
        ContaRemunerada conta = new ContaRemunerada();
        System.out.println ("Saldo inicial = " + conta.getSaldo ());
        conta.deposito (100.00);
        conta.aplicaRendimentos (0.05);
        System.out.println ("Novo saldo = " + conta.getSaldo ());
    }
}
```

A Classe ContaRemunerada

- Extende a classe **ContaCorrente**, de modo que ela também possui o atributo **saldo** e os métodos **saque**, **deposito** e **getSaldo**
- Acrescenta o método **aplicaRendimentos**
(é claro que uma implementação real desta classe verificaria qual taxa aplicar de acordo com a data do último saque ou deposito e recolheria impostos ou taxas de serviço)

Especialização de Classes no Computador



Boas Práticas

- Há uma grande tentação de definir *atributos protegidos* (**protected**) em superclasses, de modo que as subclasses possam manipular diretamente estes atributos, em vez de utilizar apenas os métodos fornecidos (públicos) pela superclasse
- Entretanto, esta prática aumenta o risco de modificações na superclasse refletirem no funcionamento incorreto da subclasse
- Portanto devemos ocultar informações também entre superclasses e subclasses!

Classes Como Subtipos

- Variáveis cujo tipo seja uma classe podem referenciar tanto objetos da classe declarada quanto de classes derivadas
- O Java define a classe **Object**, de modo que se possa definir variáveis capazes de referenciar objetos de qualquer classe
- Toda classe em Java extende implicitamente **Object**
- É graças à existência da classe **Object** que podemos utilizar **PrintWriter.println** ou **Vector.add** com instâncias de qualquer classe

Exercício

- Escreva um programa (chame de **Subtipo.java**) que realize uma **Transferencia** entre uma **ContaCorrente** e uma **ContaRemunerada**
- Como uma ContaRemunerada deriva de ContaCorrente, ela *efetivamente é* uma ContaCorrente, ou seja, variáveis ou argumentos do tipo ContaCorrente podem referenciar também objetos do tipo ContaRemunerada
- Portanto a classe **Transferencia** não necessita ser modificada para lidar com instâncias de ContaRemunerada

Subtipo.java

```
class Subtipo
{
    static public void main (String[] args) {
        ContaCorrente conta1 = new ContaCorrente();
        conta1.deposito (100.00);
        ContaRemunerada conta2 = new ContaRemunerada();
        conta2.deposito (200.00);
        System.out.println ("Conta Corrente = " + conta1.getSaldo ());
        System.out.println ("Conta Remunerada = " + conta2.getSaldo ());
        Transferencia transf = new Transferencia();
        transf.setOrigem (conta1);
        transf.setDestino (conta2);
        transf.setValor (50.00);
        transf.realiza ();
        System.out.println ("Conta Corrente = " + conta1.getSaldo ());
        System.out.println ("Conta Remunerada = " + conta2.getSaldo ());
    }
}
```

Polimorfismo

- Todas as referências a objetos em Java (sejam variáveis de instância, variáveis locais ou argumentos de métodos) são ditas *referências polimórficas*, pois podem referenciar objetos de várias classes diferentes.
- O polimorfismo (ou o uso de referências polimórficas) permite escrever código genérico, que não será modificado pela adição de novas classes ao sistema
- O código genérico é mais simples de criar e de manter, e evita duplicações de esforço

Referências Polimórficas

- A classe **Trasferencia** utiliza referências polimórficas, pois os atributos **origem** e **destino** podem referenciar tanto instâncias de **ContaCorrente** quanto de qualquer outra subclasse, como **ContaRemunerada**
- Se futuramente criarmos outros tipos de conta, elas poderão ser utilizadas como origem ou destino de transferências, sem que a classe **Transferencia** necessite ser modificada ou mesmo recompilada
- Java expande o conceito de polimorfismo para permitir "expansões pontuais" da aplicação

Sobrecarga

- Temos outra forma de polimorfismo em Java: podemos definir métodos com o mesmo nome, porém recebendo argumentos diferentes
- Chamamos a esta forma de polimorfismo de *sobrecarga de métodos*
- A sobrecarga pode mudar tanto a quantidade quanto os tipos dos argumentos; mas não pode mudar o tipo do valor de retorno de um método

ContaRemunerada2.java

```
class ContaRemunerada2 extends ContaCorrente
{
    private double taxaPadrao;

   // métodos get/set para o atributo taxaPadrao

    public void aplicaRendimentos () {
        aplicaRendimentos (taxaPadrao);
    }

    public void aplicaRendimentos (double juros) {
        deposito (getSaldo () * juros);
    }

    // método main para teste da classe
}
```

A Classe ContaRemunerada2

- Digamos que toda **ContaRemunerada** possua uma taxa (de rendimento) padrão, armazenada como um atributo de instância
- Em ocasiões especiais, o banco pode decidir dar um rendimento maior para uma conta, por exemplo se o titular da conta foi sorteado em uma promoção
- As duas versões do método **aplicaRendimentos** lidam respectivamente com a situação normal e com a ocasião especial

Boas Práticas

- Na definição de métodos sobrecarregados, há uma tendência a copiar o código de uma versão do método para a outra, que então é editada para realizar o comportamento pretendido
- Quando a lógica na versão original do método é modificada, facilmente se esquece de modificar também a cópia no método sobrecarregado
- Por isso deve-se escrever versões sobrecarregadas de métodos invocando sempre a versão mais “completa”, evitando duplicação de código

Inicialização de Instâncias

- Nem sempre faz sentido instanciar uma classe deixando seus atributos com valores “zero” ou “vazio”
- Podemos desejar estabelecer valores padrão para alguns atributos, pré-fixados ou calculados no momento da instanciação (ex: data/hora correntes)
- Podemos necessitar de informações externas para criar instâncias de uma classe – não faz sentido deixar os atributos vazios nem seus valores podem ser calculados automaticamente
(ex: CPF de um cliente, número de série de NF)

Métodos Construtores

- Sempre que uma classe é instanciada, o *método construtor* é invocado sobre a nova instância
- O compilador Java fornece um construtor padrão caso não seja definido nenhum outro construtor
- O construtor é um método público, sem tipo de retorno, com o mesmo nome da classe
- Caso o construtor receba argumentos, eles são especificados nos parênteses após o nome da classe no comando **new**
- Podemos sobrecarregar construtores

ContaCorrente v2

```
class ContaCorrente
{
   private double saldo;

   public ContaCorrente () { /*vazio*/  }

   public ContaCorrente (double saldoInicial) {
      deposita (saldoInicial);
   }

   // aqui seguem as definições de saldo, deposito e getSaldo
   // iguais à versão 1 da classe ...
}
```

Transferencia v2

```
class Transferencia
{
    private ContaCorrente origem;
    private ContaCorrente destino;
    private double valor;

    public Transferencia () { }

    public Transferencia (ContaCorrente origem,
                          ContaCorrente destino, double valor)
    {
        setOrigem (origem);
        setDestino (destino);
        setValor (valor);
    }

    // continua
```

Transferencia v2

```
// definição dos métodos get/set para origem, destino e valor,
// além do método realiza, confirme a versão 1 da classe ...

static public void main (String[] args) {
    ContaCorrente conta1 = new ContaCorrente(120.00);
    ContaCorrente conta2 = new ContaCorrente(130.00);
    System.out.println ("Saldo Inicial1 = " + conta1.getSaldo ());
    System.out.println ("Saldo Inicial2 = " + conta2.getSaldo ());
    Transferencia transf = new Transferencia (
            conta1, conta2, 50.00);
    transf.realiza ();
    System.out.println ("Saldo Inicial1 = " + conta1.getSaldo ());
    System.out.println ("Saldo Inicial2 = " + conta2.getSaldo ());
}
}
```

Boas Práticas

- Utilize no construtor os métodos **setXXX** definidos para a classe; assim garantimos que as regras de validação de dados definidas por eles serão aplicadas também nos construtores
- Não é obrigatório ter um construtor sem argumentos (muitas vezes ele não deve existir para garantir a integridade de dados das instâncias), mas ele é necessário para a manipulação em IDEs visuais
- Procure escrever todos os construtores em função de um único construtor "mais completo" (com mais argumentos) da mesma classe ou da superclasse

ContaRemunerada v3

```
class ContaRemunerada extends ContaCorrente
{
    private double taxaPadrao;

    public ContaRemunerada (double saldoInicial, double taxa) {
        super (saldoInicial);
        setTaxaPadrao (taxa);
    }

    public ContaRemunerada (double saldoInicial) {
        this (saldoInicial, 0.05);
    }

    public ContaRemunerada () {
        this (0);
    }

    // ...
```

This e Super

- Como **this** referencia a própria instância, ele pode ser utilizado para invocar outros construtores da mesma classe
- A palavra-chave **super** referencia a super-classe da instância; em geral é utilizada para chamar o construtor da superclasse
- Caso não seja inclusa a chamada ao construtor da superclasse, será invocado o construtor sem argumentos
- *Métodos construtores não são herdados!*

E os Destrutores?

- Linguagens com alocação de memória explícita definem *métodos destrutores*, executados automaticamente quando o objeto é descartado
- Um objeto Java pode nunca ser descartado (porque o programa foi encerrado antes da execução do coletor de lixo) por isso não faz sentido definir destrutores
- Operações de finalização ou limpeza, como fechar arquivos e conexões a Bds, devem ser realizadas por métodos comuns, chamados explicitamente para a liberação de recursos externos

Sobreposição

- Uma classe derivada pode *sobrepor* métodos herdados, modificando o seu comportamento ou anulando-os completamente
- Se um método é declarado tanto na superclasse quanto na subclasse, o método da subclasse predomina
- Em uma referência polimórfica, predomina a versão do método definida pela classe da instância referenciada, e não a versão definida na classe declarada para a referência

ContaEspecial.java

```
class ContaEspecial extends ContaCorrente
{
    private double limite;

    // construtores com e sem argumentos, veja ContaRemunerada

    // métodos get/set para o atributo limite

    public void saque (double valor) {
        if (getSaldo() + limite >= valor)
            super.saque (valor);
    }

    // método main para teste da classe
}
```

A Classe ContaEspecial

- Adiciona o atributo **limite**, que impõe um limite para o saldo negativo da conta – a ContaEspecial não permite que o cliente se endivide sem controle
- Sobrecarrega o método **saque** para garantir que o limite seja respeitado

Boas Práticas

- Sempre que sobrecarregamos um método de uma superclasse, há uma dependência em relação à forma como a superclasse realiza o mesmo método
- Esta dependência pode ser explicitada (e o código da subclasse simplificado, evitando duplicação de código) utilizando o método original como parte da lógica do método sobrecarregado
- A referência **super** permite o acesso aos métodos originais dentro de métodos sobrecarregados

Exercício

- Crie um programa que, utilizando a classe **Transferencia**, seja capaz de realizar transferências entre contas corrente, contas remuneradas e contas especiais
- Utilize construtores para simplificar os seus métodos de teste
- O polimorfismo é o conceito mais útil de POO sob o ponto de vista da produtividade dos desenvolvedores, mas também é um dos mais difícies de se assimilar

Herança e Valores Padrão

- Frequentemente se define subclasses que apenas fornecem valores padrão (*defaults*) para alguns atributos da superclasse, e ocasionalmente os métodos **setXXX** destes atributos são sobrecarregados para evitar a modificação destes atributos
- Também se define subclasses que não acrescentam ou modificam nenhuma das características herdadas, mas servem apenas para que outras classes possam reconhecer tipos diferentes de objetos com o mesmo comportamento

Exercício

- Suponha que o nosso banco considere três classes de cliente: o *cliente salário*, que não pode ter saldo negativo; o *cliente especial*, que pode ter um limite de R$1000,00 negativos em sua conta corrente; e o *cliente vip*, que recebe um crédito de R$5.000,00 como limite de saldo negativo
- Defina subclasses de **ContaEspecial** para cada um desses tipos de clientes e escreva um programa de teste que realize saques dentro e fora dos limites de cada conta

ContaClienteVIP.java

```
class ContaClienteVIP extends ContaEspecial
{
    public ContaClienteVIP () {
        this (0);
    }

    public ContaClienteVIP (double saldoInicial) {
        super (saldoInicial, 5000.00);
    }
}
```

- Note que desta vez foi fornecida a definição *completa* da classe!

Identificando Tipos de Classes

- Caso alguma operação venha a ser autorizada apenas para certos tipos de contas (clientes), podemos utilizar o operador **instanceof**

```
class AlgumaOperacao {

    private ContaCorrente contaOrigem;

    public void realiza () {
        if (contaOrigem instanceof ContaClienteVIP)
            // realiza a operação
        else
            // gera uma excessão
    }

//...
```

Instanceof e Herança

- O operador respeita a noção de que uma instância de uma subclasse também é uma instância das suas superclasses:
  - ContaClienteVIP instanceof ContaCorrente == true
  - ContaClienteVIP instanceof ContaEspecial == true
  - ContaClienteVIP instanceof ContaRemunerada == false

3. Interfaces

- Classes abstratas
- Refatoração
- Métodos abstratos
- Casses abstratas “puras” e Interfaces
- Interfaces e tipos de dados
- Casts

Classes Abstratas

- Muitas vezes temos conceitos que se aplicam a todo um conjunto de classes, determinando comportamentos que gostaríamos de herdar de uma superclasse, mas não faria sentido instanciar objetos desta superclasse
- Definimos então esta superclasse como sendo *abstrata*, de modo que ela possa fornecer estado e comportamento para classes derivadas, ou utiliza-la como tipo de dados para referências, mas não seja permitido criar instâncias
- As classes que podem ser instanciadas são denominadas *classes concretas*

Evolução do Projeto

- A definição origial da classe **ContaCorrente** possu um bug: ela permite que a realização de saques mesmo que não haja saldo para satisfazer a operação
- Corrigir a classe **ContaCorrente** irá prejudicar o funcionamento da classe **ContaEspecial**
- O que necessitamos é de uma classe para representar o *conceito abstrato* de “Conta”, fornecendo a funcionalidade genérica que será herdada por ContaCorrente, ContaEspecial e ContaRemunerada

ContaAbstrata.java

```
abstract class ContaAbstrata
{
    private double saldo = 0;

    public void deposito (double valor) {
        saldo += valor;
    }

    public void saque (double valor) {
        saldo -= valor;
    }

    public double getSaldo () {
        return saldo;
    }
}
```

A Classe ContaAbstrata

- Ela é exatamente igual à definição original de **ContaCorrente**, apenas acrescida da palavra-chave **abstract** e com o novo nome
- Seu objetivo é fornecer apenas a manutenção do saldo para as classes derivadas, que são por sua vez responsáveis por decidir quando saques e depósitos são permitidos, e por manter informações adicionais necessárias para o cálculo de rendimentos ou multas
- Em um sistema real, a **ContaAbstrata** também manteria históricos, auditoria, segurança, etc

ContaCorrente.java v3

```
class ContaCorrente extends ContaAbstrata
{

    // construtores

    public void saque (double valor) {
        if (getSaldo () >= valor)
            super.saque (valor);
    }

    // método main para teste da classe
}
```

A Classe ContaCorrente v3

- Note que ela é idêntica à classe **ContaEspecial** se for removido o atributo **limite**
- As classes **ContaRemunerada** e **ContaEspecial** passam a especializar **ContaAbstrata** em vez de **ContaCorrente**
- A classe **Transferencia** continua funcionando com os três tipos de contas, apesar das modificações, desde que suas referências polimórficas sejam ajustadas para o tipo **ContaAbstrata**
- Bons sistemas OO permitem modificações extensas sem efeitos colaterais ou “bolas de neve”

Refatoração

- No desenvolvimento de sistemas de modo geral, e em especial em sistemas OO, frequentemente decidimos mover a funcionalidade presente em uma dada classe para uma nova superclasse ou subclasse
- Ou ainda decidimos que certa lógica é necessária em vários métodos então decidimos criar um novo método privativo (**private**)
- O processo de reescrever classes e métodos para evitar a duplicação de código ou aumentar o seu reaproveitamento é chamado de *refatoração*

Refatoração x Qualidade

- A refatoração de código pode parecer perda de tempo a princípio, pois o tempo gasto poderia ser utilizado no desenvolvimento de novas funcionaldiades no sistema
- Entretanto, a refatoração aumenta a qualidade e a clareza do código, ou a sua resistência à mudanças, de modo que ela contribui para a maior qualidade e sobrevida do sistema
- Lembrando: mais de 80% do tempo e esforço gasto em um sistema (ou em um módulo do sistema) ocorre depois da entrega da sua primeira versão ao usuário!

Discussão

- Refatorar código x Compatibilidade retroativa

Métodos Abstratos

- O uso do polimorfismo exige a compatibilidade de tipo entre os objetos envolvidos, o que em geral se obtém com o uso de superclasses
- Mas há situações em que não é possível fornecer uma implementação padrão para ser herdada; criar uma classe abstrata com métodos vazios não é elegante...
- Nestes casos a implementação da superclasse abstrata é “incompleta”: as classes concretas derivadas devem complementar o comportamento herdado

Métodos Abstratos

- Um *método abstrato* fornece apenas uma *assinatura*, mas nenhuma implementação
- Métodos abstratos só podem ser definidos em classes abstratas
- Uma subclasse deve implementar todos os métodos abstratos herdados, ou deve ser ela mesma declarada como sendo abstrata

Transferencia.java v2

```
abstract class Transferencia
{
    // atributos origem, destino e valor

    protected abstract double getLimite ();

    public void realiza () {
        if (valor <= getLimite ()) {
            origem.saque (valor);
            destino.deposito (valor);
        }
    }

}
```

Transferencia v2

- O valor que pode ser transferido depende de onde é feita a transferência:
  - Pequenos valores podem ser transferidos pelo caixa eletrônico;
  - Valores maiores podem ser transferidos na agência;
  - Valores muito grandes devem ser realizados por emissão (e posterior depósito) de cheques administrativos
- O método abstrato **getLimite** permite que classes especializadas para a agência e para o caixa eletrônico e herdem a capacidade de realizar transferências e sejam capazes de validar o limite de forma segura

TransferenciaAgencia.java e Transferencia24hs.java

- ```
  class TransferenciaAgencia extends Transferencia
  {
      protected double getLimite () {
          return 20000.00
      }
  }
  ```
- ```
  class Transferencia24hs extends Transferencia
  {
      protected double getLimite () {
          return 3000.00
      }
  }
  ```

Discussão: Método Abstrato getLimite x Atributo limite

- O método abstrato **getLimite** corresponde a um atributo **limite**
- Em alguns tipo de transferência o valor do limite pode ser calculado em função de outros parâmetros
- Ex: limite de R$500,00 para saques antes das 20:00hs, limite de R$100,00 para saques depois das 20:00hs
- Métodos abstratos permitem a uma subclasse modificar um algoritmo implementado na superclasse, o que é mais poderoso do que apenas parametrizar um algoritmo pré-fixado

Discussão: Métodos Abstratos x Sobreposição

- As subclasses de **Transferência** poderiam sobrecarregar o método **realiza** para verificar se o valor está dentro dos limites permitidos para a subclasse
- Entretanto a existência de um limite é parte do conceito de “transferência entre contas”, portanto deve ser expresso de alguma forma na classe abstrata
- Manter esta funcionalidade na superclasse (validação do limite) permite que ela trate corretamente a integridade transacional, auditoria, etc

Métodos Protegidos

- Quando um método existe apenas para uso das subclasses podemos defini-lo como sendo *protegido* (**protected**)
- Desta forma podemos decidir que comportamento é exposto para qualquer outra classe e que comportamentos são expostos apenas para as subclassses
- Também podemos definir atributos protegidos, mas não é recomendado (ocultamento de informações)
- Métodos abstratos não são necessariamente protegidos

Investimentos

- Uma caderneta de poupança simples rende juros sobre a quantia que não for movimentada durante o mês, de modo que ela não tem um único saldo
- Cadernetas multidata podem render juros em várias datas de aniversário diferentes, de acordo com a realização de depósitos
- Fundos de ação possuem uma quantidade de quotas e um valor da quota
- Mas é interessante manipular investimentos da mesma forma que contas-corrente, realizando saques, depósitos, transferencias, ...

Classes Abstratas Puras

- Poderíamos definir classes **FundoDeAcoes** e **CadernetaDePoupanca** herdando de **ContaAbstrata** (para utilizar o polimorfismo) e sobrepondo todos os métodos... não usufruindo do comportamento herdado, mas apenas das definições de métodos
- Em linguagens OO podemos definir *classes abstratas puras* (onde todos os métodos são abstratos) para utilizar polimorfismo sem herdar comportamentos que não serão utilizados
- O Java extende este conceito para a definição de *interfaces*

Interfaces

- Interfaces são classes que não definem a implementação dos métodos, mas apenas as suas assinaturas (nomes e argumentos)
- Interfaces não podem ter atributos
- Todos os métodos de uma interface são implicitamente abstratos e públicos
- Uma classe pode extender apenas uma única superclasse (para evitar conflitos entre os comportamentos herdados), mas pode implementar várias interfaces (mesmo que elas definam métodos com a mesma assinatura)

Equivalência entre Interfaces e Classes Abstratas

- ```
  interface Conta
  {
      void saque (double valor);
      void deposito (double valor);
      double getSaldo ();
  }
  ```

- ```
  abstract class Conta
  {
      public abstract void saque (double valor);
      public abstract void deposito (double valor);
      public abstract double getSaldo ();
  }
  ```

Interfaces e Tipos de Dados

- Podemos declarar variáveis tipadas por interfaces da mesma forma que variáveis tipadas por classes
- O primeiro caso indica uma referência para qualquer objeto que implemente a interface
- O segundo caso indica uma referência para qualquer objeto da classe ou de suas subclasses
- Interfaces permitem definir funcionalidade baseada em conceitos que se aplicam a várias classes distintas (em hierarquias de especialização diferentes)

Extends x Implements

- Uma classe *extende* outra classe
- Uma classe *impementa* várias interfaces
- Uma interface *extende* várias interfaces
- Uma classe pode, **ao mesmo tempo**, *extender* outra classe e *implementar* várias interfaces

Investimento.java

- ```
  interface Conta
  {
      void saque (double valor);
      void deposito (double valor);
      double getSaldo ();
  }
  ```
- ```
  interface Investimento extends Conta
  {
      Date getProximoAniversario ();
      void aplicaRendimentos ();
  }
  ```
- ```
  class ContaAbstrata implements Conta
  {
      private double saldo;
      //...
  ```
  ```

A Interface Investimento

- Investimentos são tipos de Conta, pois podem sober operações de saque e deposito
- Além disso, investimentos recebem rendimentos em datas de aniversário pré-definidas
- O mesmo código pode realizar transferências entre quaisquer tipos de Contas e/ou Investimentos
- O mesmo programa de retaguarda pode realizar a aplicação de rendimentos a cada dia em todos os tipos de investimentos

CadernetaDePoupanca.java

```
class CadernetaDePoupanca implements Investimento
{
    private double saldoInicial;
    private double depositosPendentes;

    public void saque (double valor) {
        saldoInicial -= valor;
    }

    public void deposito (double valor) {
        depositosPendentes += valor;
    }

    public double getSaldo () {
        return saldoInicial + depositosPendentes;
    }

// continua
```

CadernetaDePoupanca.java

```
private java.util.Date dataAbertura;

public Date getProximoAniversario () {
    // calcula o próximo aniversario a partir da data de
    // abertura (mesmo do mês, ou o próximo dia útil)
}

public void aplicaRendimentos () {
    if (/* hoje é o dia do aniversário*/)
        deposito (saldoInicial * getTaxaMesCorrente ());
        saldoInicial += depositosPendentes;
        depositosPendentes = 0;
}

private static double getTaxaMesCorrente () {
// ...
```

ContaRemunerada v3

```
class ContaRemunerada extends ContaAbstrata
implements Investimento
{
    //...

    public void saque (double valor) {
        //...
    }

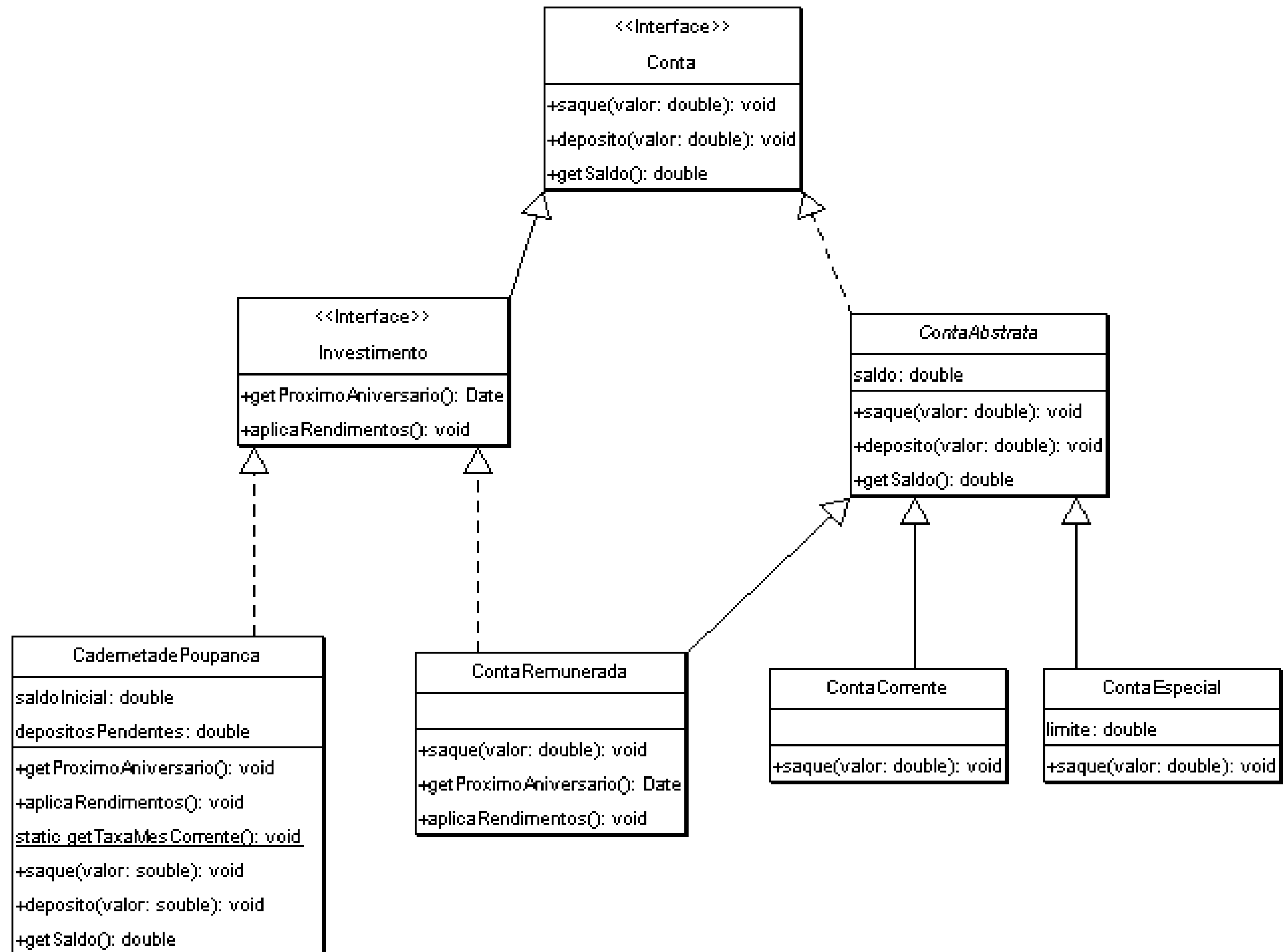
    public void aplicaRendimentos () {
        //...
    }

    // ...
}
```

Interfaces e instanceof

- Uma instância é considerada como *sendo* cada uma das interfaces implementadas pela sua classe
  - ContaEspecial instanceof Conta == true
  - ContaEspecial instanceof Investimento == false
  - ContaRemunerada instanceof Conta == true
  - ContaRemunerada instanceof Investimento == true
  - ContaRemunerada instanceof ContaAbstrata == true
  - CadernetaDePoupanca instanceof Conta == true

Tipagem Forte

- O compilador Java verifica chamadas a métodos de acordo com o tipo declarado para a variável que referência o objeto
- Assim uma variável do tipo **Conta** pode referenciar uma **ContaEspecial**, mas não pode invocar o método **setLimite**
- Somos obrigados a copiar a referência para uma variável (referência) declarada como **ContaEspecial**
- Mas o compilador também não irá aceitar esta atribuição

Atribuições e Casts

```
// compila e funciona corretamente
Conta conta = new ContaRenumerada ();
Investimento invest = new ContaRenumerada ();

// a instância é compatível, mas o compilador não aceita
invest = conta;

// a instância tem esta capacidade, mas o compilador não aceita
conta.aplicaRendimentos ();

// Temos que utilizar um cast para compatibilizar as referências
invest = (Investimento)conta;
((Investimento)conta).aplicaRendimentos();

// As duas linhas acima serão verificadas em tempo de
// execução, evitando erros de “tela azul”
```

Compatibiliade de Tipos

- As convesões “para cima” são automáticas
  - Uma subclasse é compatível com a superclasse
  - Uma classe é compatível com as interfaces implementadas
- As conversões “para baixo” devem ser explicitadas por meio de casts
  - Superclasse para subclasse
  - Interface para classe implementadora
- Veja adiante o tópico sobre coleções

Diagrama de Classes UML



Diagrama de Classes UML

- Um quadrado com duas divisões é uma interface
- Um quadrado com três posições é uma classe
- Setas triangulares expressam relacionamentos de especialização (setas cheias) ou de implementação (setas tracejadas)
- Texto em itálico indica classe ou método abstrato
- Texto sublinhado indica atributo ou método estático (de classe)

4. Relacionamentos entre Classes

- Usar ou não a Herança
- Coesão e Acoplamento
- Agregação
- Delegação
- Value Objects
- JavaBeans
- Coleções
- Camadas da aplicação e o modelo MVC

Cuidado com a Herança!

- A herança permite escrever menos código (mais produtividade) entretanto ela cria uma forte dependência entre subclasse e superclasse
- Modificações na superclasse podem comprometer o funcionamento da subclasse
- Pense bem antes de usar herança: realmente existe especialização a nível conceitual (no “mundo real”) ou é apenas uma coincidência que as classes tenham estado e/ou comportamento semelhantes?

Um Mal Exemplo

- Pense em várias janelas de uma mesma aplicação: elas tem elementos comuns, como um mesmo menu, um mesmo toolbar, uma mesma barra de status, ...
- Isto significa que devemos criar uma superclasse com esses elementos, e derivar dela classes para cada janela da nossa aplicação?
- Não, porque conceitualmente uma janela não é um subtipo (ou uma especialização) da outra janela. Janelas de uma aplicação em geral tem propósitos completamente diferentes

Parâmetros de Qualidade do Softwae

- *Coesão*: uma boa classe (ou um bom método) é coeso, o que significa que ele serve apenas um propósito bem-definido
- *Acoplamento*: uma boa classe (ou um bom método) é pouco acoplada com outras classes (ou métodos), ou seja, tem poucas (idealmente nenhuma) dependências sobre o funcionamento de outras classes (ou métodos)

Agregação

- Em geral é melhor agregar classes do que derivar classes
- Agregar classes é usar classes como partes constituintes de outras classes
- Uma instância da classe A contém instâncias da classe B: A agrega B ou A é composta por B
- Em Java, isto significa que uma classe tem atributos que armazenam objetos de outras classes
- No exemplo, cada janela pode conter um atributo toolbar, um atributo menu, etc

Agregação e Delegação

- Nem sempre o fato de uma classe agragar outra classe deve ser exposto
- Podemos utilizar o objeto agregado apenas para utilizar sua funcionalidade, *delegando* tarefas para ele
- Uma subclasse delega automaticamente para a superclasse

Exemplos

- *Agregação:*
  Toda conta ou investimento possui um histórico de operações, utilizado para gerar extratos
- *Delegação:*
  Um objeto conta delega uma solicitação de extrato para o objeto de histórico agregado
- *Composição:*
  Cada objeto histórico contém vários objetos operação, referentes a depósitos, retiradas, transferências, pagamentos, recolhimento de impostos, ...

Agregações em Java

- Definimos atributos que sejam referências a outros objetos
- Definimos atributos que sejam referências à arrays ou coleções de outros objetos
- Esses atributos podem ser inicializados (as referências instanciadas) no construtor da classe
- Lembre que instanciar uma coleção ou array não instancia os objetos referenciados por cada elemento

Exercício

- Defina as classes **Operacao**, **Historico** e **Extrato**, e inclua na interface **Conta** o método

  ```
  public Extrato getExtrato (Date dataInicio,
      Date dataFim)
  ```

- **Extrato** também é uma agregação de **Operacao**
- Cada operação de saque ou retirada adiciona uma **Operacao** ao **Histórico** agregado à **Conta**

Discussão: Porque ter classes Historico e Extrato?

- O histórico é parte de uma conta (e consequentemente de um investimento) e contem operações representando tudo o que foi feito sobre a conta desde a sua criação
- Um extrato é um relatório entregue ao cliente em um dado instante de tempo, relacionando as operações realizadas sobre uma conta em um dado período de tempo
- Como se vê, apesar de ambos serem agregados de operações, eles possuem finalidades, atributos e capacidades diferentes

Classes de Coleção

- É comum se criar classes cuja principal finalidade é gerenciar instâncias de outras classes
- Estas classes incorporam a funcionalidade de criar, destruir, localizar e organizar (ordenar, filtrar) as instâncias agregadas por ela
- Classes de coleção também são naturais para lógica de negócios que envolve várias instâncias de uma mesma classe ou de tipos compatíveis
  Ex: aplicação dos rendimentos mensais em cadernetas de poupança

Coleções em Java

- Devemos diferenciar entre dois tipos de classes de coleção:
  - Coleções de objetos necessários à realização da lógica de negócios da aplicação – estas coleções são parte dos conceitos da aplicação e representam objetos do “mundo real”
  - Coleções de objetos que representam estruturas de dados – estas coleções são ferramentas para a implementação das coleções de negócios ou da lógica de métodos de negócio específicos
- Veja mais adiante o conceito de camadas da aplicação

API de Coleções do Java

- O Java original possuía apenas as classes Vector e Hashtable para representar coleções dinâmicas, enquando que os arrays representam coleções estáticas
- O Java2 definiu uma nova API de coleções permitindo isolar a lógica de negócios de algoritmos especializados de busca e ordenação ou de compromissos de performance x consumo de memória

Interfaces de Coleção do Java

- Há dois tipos básicos de coleção: *Collections* e *Maps*
- Uma **Collection** apenas agrega vários objetos de qualquer tipo
- Um **Map** associa pares de objetos na forma (chave, valor)
- Qualquer tipo de coleção pode ser percorrida através de um objeto especializado que implementa a interface **Iterator**

Collections

- Há dois subtipos de Collection em Java:
- *Sets* representam conjuntos, não podem conter duplicatas e não é preservada a ordem dos elementos
- *Lists* representam listas ordenadas, no sentido de que é preservada a ordem de inserção dos elementos na coleção, e podem conter duplicatas

Coleções Ordenadas

- *SortedSets* são Sets capazes de retornar seus elementos segundo um critério de ordenação definido previamente
- SortedMaps são Maps capazes de retornar seus elementos segundo um critério de ordenação definido previamente sobre suas chaves

Compatibilidade Retroativa

- **Vector** foi refatorado em um **List** e **HashTable** foi refatorada como um **Map**, mas ambos estão presentes para compatibilidade retroativa
- recomenda-se utilizar as novas classes definidas no Java2, como:
  - **ArrayList** é a implementação mais comum de **List**
  - **HashSet** é a implementação mais comum de **Set**
  - **TreeSet** é a implementação mais comum de **SortedSet**

A interface Collection

- Atuam sobre elementos individuais da coleção:
  - size
  - isEmpty
  - contains
  - add
  - remove
  - iterator

- Realizam operações sobre todos os elementos da coleção:
  - containsAll
  - addAll
  - removeAll
  - clear
  - toArray

Iterators

- Toda coleção é capaz de fornecer um objeto Iterator que contém a inteligência sobre como percorrer todos os elementos da coleção
- A interface Iterator implementa os métodos:
  - hasNext: indica se chegamos ou não ao fim da coleção
  - next: fornece uma referência para o próximo objeto da coleção
- Um iterator percorre a coleção sequenciamente, do início ao fim
- Podemos ter vários objetos Iterator independentes para uma mesma coleção

Exemplo: Percorre.java

```
class Percorre {
    public static void main (String[] args) {
        // popula a coleção
        List lista = new ArrayList();
        lista.add ("Primeiro");
        lista.add ("Segundo");
        lista.add ("Terceiro");
        // percorre a coleção
        Iterator it = lista.iterator ();
        while (it.hasNext ()) {
            String elemento = (String)it.next ();
            System.out.println (elemento);
        }
    }
}
```

Sobre o Exemplo

- O loop **while** utilizado para percorrer a coleção lista poderia ser escrito da mesma forma para qualquer outro tipo de coleção
- Observe o uso de casts na chamada ao método **next**, pois as interfaces **Collection** e **Iterator** declaram seus argumentos e valores de retorno como referências a **Object**

Exercício

- Modifique o exemplo **Percorre.java** para criar e percorrer um Set em vez de um List
- Só será necessário modificar a instanciação da coleção, a sua população e o loop while permanecem inalterados
- Os elementos são retornados na ordem em que foram inseridos?
- Experimente também inserir elementos duplicados e depois percorrer um **List** e um **Set**.

A Interface List

- Adiciona à Collection os métodos
  - add (int index, Object element)
  - set (int index, Object element)
  - Object get (int index)
  - int indexOf (Object element)
  - int lastIndexOf (Object element)
  - List sublist (int from, int to)
- É melhor precorrer uma lista utilizando o Iterator do que utilizando os métodos **get** e **size**.

Utilitários para Coleções e Arrays

- A classe **java.util.Collections** fornece métodos estáticos para buscar elementos, ordenar, inverter, “embaralhar” e outras operações sobre coleções quaisquer
  - Não confundir com a *interface* **java.util.Collection**!
- A classe **java.util.Array** contém um métodos estáticos que permitem transformar um array em uma lista para usufriur das funcionalidades em **Collections**
  - Não confundir com a classe **java.lang.Array**!

A Interfafe Map

- Atuam sobre pares (chave, valor) individuais:
  - put (key, value)
  - get (key)
  - remove (key)
  - containsKey (key)
  - containsValue (value)
  - size
  - isEmpty
- Atuam sobre o conjunto de pares
  - putAll
  - clear
- Geram coleções à partir dos pares
  - keySet
  - values
  - entrySet

Como Percorrer um Map

- Obtemos um **Iterator** sobre a coleção de chaves (**keySet**) e utilizamos o método **get(key)** sobre cada chave retornada pelo Iterator
- Obtemos um **Iterator** sobre a coleção de valores (**values**)
- Obtemos um Iterator sobre a coleção de pares (**entrySet**), que retorna objetos **Map.Entry** com a seguinte interface:
  - getKey
  - getValue
  - setValue

Exemplo: PercorreMapa.java

```
class PercorreMapa
{
    public static void main (String[] args) {
        // popula o mapa
        Map mapa = new HashMap();
        mapa.put ("Um", "Primeiro");
        mapa.put ("Dois", "Segundo");
        mapa.put ("Três", Terceiro");
        // percorre o mapa
        Iterator it = mapa.keySet().iterator ();
        while (it.hasNext ()) {
            String chave = (String)it.next ();
            String valor = (String)mapa.get (chave);
            System.out.println (valor);
        }
    }
}
```

Nem Sempre Precisamos dos Casts

```
class PercorreMapa2
{
    public static void main (String[] args) {
        // popula o mapa
        Map mapa = new HashMap();
        mapa.put ("Um", "Primeiro");
        mapa.put ("Dois", "Segundo");
        mapa.put ("Três", Terceiro");
        // percorre o mapa
        Iterator it = mapa.keySet().iterator ();
        while (it.hasNext ()) {
            Object chave = it.next ();
            // println e get delcaram Object como argumento
            System.out.println (mapa.get (chave));
        }
    }
}
```

Exercícios

- Reescreva o exemplo **PercorreMapa.java** para utilizar as duas outras formas de se percorrer um Map
- Qual delas seria mais adequada para fornecer os objetos armazenados em um **Map** para um método que espere receber uma referência a **Collection** como argumento?
- Experimente: como exibir os elementos contidos dentro de um mapa ordenados pela chave ou pelo valor?

Exemplo: OrdenaMapa.java

```
import java.util.*;

class OrdenaMapa
{
    public static void main (String[] args) {
        // popula o mapa
        Map uf = new HashMap();
        uf.put ("RJ", "Rio de Janeiro");
        uf.put ("SP", "São Paulo");
        uf.put ("MG", "Minas Gerais");
        uf.put ("ES", "Espírito Santo");
        // ordena pelas siglas
        System.out.println ("Ordenado pelas chaves...");
        List chaves = new ArrayList ();
        chaves.addAll (uf.keySet());
        Collections.sort (chaves);

// continua...
```

Exemplo: OrdenaMapa.java

```
        // continuação...

        // percorre o mapa
        Iterator it = chaves.iterator ();
        while (it.hasNext ()) {
            String chave = (String)it.next ();
            String valor = (String)uf.get (chave);
            System.out.println (chave + "\t" + valor);
        }
    }
}
```

- Observe que foi necessário inserir os elementos do **Set** retornado por **keySet** em um **List** para que fosse possível ordena-los.

E para Ordenar pelos Valores?

- Acrescentar ao exemplo anterior:

```
// ordena pelos nomes
System.out.println ("Ordenado pelas valores, com chaves...");
List pares = new ArrayList ();
pares.addAll (uf.entrySet());
Collections.sort (pares, new ComparaNomes());
// percorre o mapa
Iterator it3 = pares.iterator ();
while (it3.hasNext ()) {
    Map.Entry par = (Map.Entry)it3.next ();
    System.out.println (par.getKey() + "\t" + par.getValue());
}
```

A classe ComparaNomes

- ```
  class ComparaNomes implements Comparator
  {
      public int compare(Object o1, Object o2) {
          Map.Entry e1 = (Map.Entry)o1;
          Map.Entry e2 = (Map.Entry)o2;
          String nome1 = (String)e1.getValue ();
          String nome2 = (String)e2.getValue ();
          return nome1.compareTo (nome2);
      }
  }
  ```
- Podemos definir esta classe no próprio arquivo **OrdenaMapa.java**, mas ela não será visível para nenhuma classe em outros arquivos
  ```

Coleções Personalizadas

- Além de criarmos coleções de negócio para representar conceitos da aplicação, também criamos coleções apenas para simplificar as chamadas a métodos, eliminando a necessidade de casts frequentes
- Muitas vezes estas coleções são atributos de classe (estáticos)

Exercício

- Crie a classe **MapaStrings** para facilitar o uso de um **Map** onde tanto chaves quanto valores são Strings, e modifique o exemplo PercorreMapa para utilizar esta casse
- Observe que não podemos sobrecarregar o método **get** para retornar um String!

Exercício Opcional

- Criar a classe **Contas** que contém todas as contas e investimentos definidos para o sistema
- Definir um método para localizar uma conta dado o seu "id" (número) – este atributo deve ser acrescentado à interface **Conta**!
- Esta classe torna a aplicação independente do fato das contas serem armazenadas em memória, em um arquivo, em um banco de dados ou serem acessadas via WebServices!

Igualdade x Identidade

- Cada instância de uma mesma classe é considerada um objeto único e diferente dos demais
- Mas no mundo real existe ainda o conceito de igualdade
- Tome duas canetas: podemos reconhece-las como sendo iguais (são ambas novas, da mesma cor, do mesmo modelo e fabricante, ..) mesmo assim as reconhecemos como dois objetos diferentes, que podem ser manipuladas de modos diferentes

Value Objects

- Muitas classes representam valores:
  Integer, String, Date, ...
- Objetos de valor (*value objects*) possuem o conceitos de igualdade
- Muitos objetos não possuem o conceito de igualdade, ou não permitem que hajam duas instâncias com exatamente o mesmo estado
- Ex: Nota fiscal (mesmo que sejam duas notas fiscais para o mesmo produto e para o mesmo cliente, ainda são duas notas fiscais diferentes)

Value Objects em Java

- Devem sobrepor o método **equals (Object o)**, definido na classe **Object**
- Devem sobrepor o método **hashCode()**, também definido na classe Object
- Dois objetos iguais devem retornar o mesmo **hashCode** para que possam ser manupulados corretamente dentro de Coleções
- Recomenda-se que seja implementado o método **toString()**
- Recomenda-se também um construtor que inicialize todos os atributos

Value Objects e Coleções

- **Sets** e **Maps** simples ou coleções ordenadas podem funcionar erroneamente caso sejam utilizados para armazenar instâncias de classes que não sejam Value Objects
  - **equals** determina se o objeto já está presente no **Set** ou se achave já está presente no **Map** – não podem haver chaves duplicadas, embora objetos iguais possam estar associados a chaves diferentes
  - **hashCode** é utilizado nos algoritmos de hashsing utilizados por **Set** e **Map**

Endereco.java

```
class Endereco
{
    private String logradouro;
    private int numero ;
    private String complemento;
    private String bairro;
    private String cep;

    // métodos get/set para os atributos

    public Endereco () {}

    public Endereco (String logradouro, int numero, ...) {
        this.logradouro = logradouro;
        this.numero = numero;
        ....
    }
```

Endereco.java

```
public boolean equals (Object obj) {
    if (o instanceof Endereco) {
        Endereco e = (Endereco)obj;
        return (
            ( logradouro == null && e.getLogradouro() == null )
            || ( logradouro != null &&
                logradouro.equals (e.getLogradouro ()) ) &&
            ( numero == e.getNumero () ) &&
            ...
    }
    else
        return false;
}
```

Endereco.java

```
    public int hashCode () {
        return (logradouro + numero + ...).hashCode ();
    }

    public String toString () {
        return logradouro + ", " + numero + " " + complemento
            ...
    }

}
```

Exercício

- Crie uma classe **Telefone** que funcione como um Value Object
- Ela deve conter ao menos os atributos código de área, número e ramal
- O método **toString** deve gerar o número convenientemente formatado, mas sem espaços ou pontuação supérfulos
(experimente com e sem código de área)
- Escreva um programa de teste que instancie e compare vários objetos Telefone

Discussão

- Quais destas classes seriam Value Objects?
  - CPF
  - ContaCorrente
  - Cliente
  - Produto
  - Aniversário
  - Peso

Características Opcionais de Objetos Java

- Especificadas por interfaces definidas em **java.lang**
  - *Comparable*
    Podem ser comparados. Necessária em objetos inseridos em coleções ordenadas, a não ser que seja fornecido um **Comparator**
  - *Cloneable*
    Podem ser clonados (podemos criar cópias deles, com estados independentes – não é a mesma coisa que copiar todos os atributos!)
  - *Serializable*
    Podem ser salvos (persistidos) em memória não-volátil e recuperados para o seu estado original

JavaBeans

- São classes onde:
  - Existe um construtor sem argumentos
  - Existem métodos get/set para todos os atributos
  - São serializáveis
- Podem ser utilizadas como componentes em aplicações GUI, páginas web, ...

Domínios de Objetos

- Classes em uma camada só devem interagir com classes de domínios inferiores, fomentando a reutilização e a manutenabilidade do sistema
- *Domínio da Aplicação*
  gerenciamento e reconhecimento de eventos
- *Domínio de Negócio*
  Atividade/Transação, Relacionamento, Papel ou Atributo
- *Domínio de Arquitetura*
  Interface humana, Banco de Dados e Comunicação
- *Domínio de Base*
  Classes Semânticas, Estruturais e Fundamentais

Exemplo Hipotético

- *Domínio da Aplicação*
  gerenciamento (FormDeBusca) e
  reconhecimento de eventos (FiltroDePesquisa)
- *Domínio de Negócio*
  Atividade/Transação (Transferencia), Relacionamento (Fornecedor), Papel (Cliente) ou Atributo (ItemNotaFiscal)
- *Domínio de Arquitetura*
  Interface humana (JFrame), Banco de Dados (Resultset) e Comunicação (URLConnection)
- *Domínio de Base*
  Classes Semânticas (Date), Estruturais (Vector) e Fundamentais (String)

O Modelo MVC

- Outra forma popular de divisão das classes de uma aplicação em camadas conforme o seu papel é o Modelo Model-View-Controller
  - **Model** *(Modelo)* representa as informações manipuladas pela aplicação
  - **View** *(Visualização)* representa uma visualização destas informações ou uma inteface para a sua edição
  - **Controller** *(Controlador)* determina mudanças sobre o modelo e provoca a atualização das visualizações correspondentes

MVC não é Três Camadas!

- O falado desenvolvimento em três camadas consiste na separação *física* da aplicação em três camadas, hospedadas em nós diferentes da rede:
  - **Apresentação**, que fornece a interface com o usuário
  - **Negócios**, que fornece a inteligência da aplicação
  - **Persistência**, que armazena e recupera informações de um banco de dados
- Os modelos de Domínios de Classes, MVC ou Três Camadas (*Three-Tier*) não são mutuamente excludentes!

5. Outros Recursos OO de Java

- Excessões
- Pacotes
- Reflexão

A Java API

- O maior benefício do Java é padronizar não apenas classes fundamentais, mas também classes semânticas, estruturais e da Camada de Arquitetura
- Já existem esforços em várias indústrias para padronizar classes na camada de Negócio, via EJBs ou WebServices
- O objetivo final das metodologias OO é construir novas aplicações apenas acrescentando novas classes na camada de Aplicação, reutilizando as classes pré-existentes da camada de Negócio

Tratamento de Erros

- Como um objeto informa a outro que algo não funcionou ou não é possível?
- Em linguagens estruturadas, definimos códigos de retorno ou variáveis de status
- Mas a necessidade constante de testar por esses códigos e variáveis prejudica a clareza da lógica de negócios
- Além disso, frequentemente estes testes são omitidos e bugs se propagam sem serem detectados na origem

Excessões

- Linguagens OO modernas como Java, C++ e ObjectPascal definem o conceito de excessão
- Uma *excessão* é um objeto tratado de modo especial pelo ambiente de execução (run-time): sua geração provoca o término imediato do subprograma onde foi gerado, do programa chamador e assim sucessivamente até que o programa principal seja terminado ou algum subprograma capture a excessão
- Semelhante ao conceito de *sinal* no Unix ou de *evento* no Windows

Excessões em Java

- São disparadas pelo comando **throw**
- São capturadas em blocos **try..catch** de modo a não interferir com a lógica de negócios – as excessões são tratadas à parte
- Caso não sejam capturadas por um método, devem ser declaradas na sua assinatura pela cláusula **throws**
- Não capturar nem declarar uma excessão impede a compilação do programa, de modo que possíveis falhas na execução do método tenham que ser tratadas em algum momento pela aplicação

Exceptions e Throwables

- Todas as excessões são subclasses de **Throwable**
- Mas apenas as subclasses de **Exception** são verificadas pelo compilador
- As subclasses de **Error** não necessitam ser capturadas nem declaradas
- Toda excessão carrega uma mensagem descritiva, um *stack frame* (sequência de métodos até o momento da excessão) e possivelmente outras excessões aninhadas
- Frequentemente definimos novas excessões dentro do contexto da aplicação

Exceptions x RuntimeErrors

- IOException
- FileNotFoundException
- SecurityException
- SaldoInsuficienteException
- LimiteTransferenciaEletronicaExcedidoException
- Não utilizamos **instanceof** para diferenciar subclasses de **Exception**, mas sim múltiplas cláusulas **catch** para um mesmo bloco **try**

- NullPointerException
- ClassCastException
- DivisionByZero
- NumberFormatException
- ArrayIndexOutOfBoundsException
- Podemos capturar **RuntimeErrors** utilizando blocos **try..catch**, apenas não somos obrigados a faze-lo pelo compilador

Onde e Quando capturar Exceptions?

- A lógica de negócio não deve interagir com o usuário, mas apenas gerar excessões para que a lógica de interface com o usuário (ou a lógica de um outro objeto de negócios) possa decidir como responder à excessão e/ou fornecer *feedback* adequado ao usuário
- Em geral não devemos expor excessões de uma camada inferior para objetos de camadas superiores; por isso aninhamos excessões
  (veja o método **Throwable.getCause**)

Exercício

- Modificar as classes e interfaces do exemplo de contas bancárias para gerar excessões quando saques e transferências não puderem ser realizados:
  - SaqueNaoAutorizadoException
  - SaldoInsuficienteException extends SaqueNaoAutorizadoException
  - LimiteInsuficienteException extends SaqueNaoAutorizadoException
  - LimiteOperacaoExcedido extends SaqueNaoAutorizadoException
  - TransferenciaNaoAutorizadaException e subclasses

Gerenciando Classes

- Uma aplicação típica em Java define dezenas ou milhares de classes
- Aplicações comerciais podem definir milhares de classes
- O próprio Java define milhares de classes para atividades comuns como criar interfaces gráficas ou acessar bancos de dados
- Como garantimos que os nomes das classes não entrem em colisão uns com os outros?

Pacotes

- A solução do Java é o conceito de *pacote*
- Pense em pacotes como subsistemas da sua aplicação, ou como uma biblioteca de objetos com um objetivo comum
- O Java define uma série de pacotes padrão:
  - **java.math** define classes para matemática de precisão
  - **java.sql** define classes para bancos de dados relacionais
  - **java.text** ajuda na internacionalização de programas
  - **java.util** contém várias estruturas de dados
  - **java.net** contém classes para comunicação em redes

Pacotes

- Os nomes de pacotes iniciados por “java” e “javax” são reservados para futuras expansões na API padrão do Java
- Os seus pacotes devem ser nomeados de acordo com o domínio Internet da sua empresa ou instituição, por exemplo **br.eti.lozano** ou **com.acme**
- Várias classes de um ou mais pacotes podem ser agrupados arquivo JAR, que nada mais é do que um arquivo ZIP

Utilizando Pacotes

- O comando **import** permite que um programa Java utilize classes definidas em um pacote
- Podemos incluir somente uma classe de um pacote
  include java.util.Random;
- Podemos incluir todo um pacote
  include java.util.*
- Incluir um pacote não inclui subpacotes, ou seja, incluir **java.util.*** não inclui **java.util.zip.***
- Todo programa inclui automaticamente o pacote **java.lang**, que define **String**, **Double**, **Math** e etc

Criando Pacotes

- O comando **package** indica que a classe definida em um arquivo *.java faz parte de um pacote
- Você deve criar subdiretórios correspondentes aos nomes dos seus pacotes, caso contrário as suas classes não serão encontradas pela JVM
- Note que as classes também podem ser públicas ou privativas em relação ao seu pacote, de modo que não precisamos expor todas as classes de um subsistema para o restante da aplicação!

Estrutura de Pacotes para uma Aplicação

- Deve seguir a divisão do sistema em módulos e também o conceito de camadas apresentado anteriormente:
  - com.acme.vendas.base
  - com.acme.vendas.comissao.negocio
  - com.acme.vendas.comissao.gui
  - com.acme.vendas.comissao.web
  - com.acme.vendas.comissao.entidades
  - com.acme.vendas.produtos.negocio
  - ...

Exemplo Simples

- Suponha uma organização para o sistema bancário
  - **banco.conta** contém as classes relacionadas com contas correntes e assemelhadas
  - **banco.investimento** contém as classes relacionadas com investimentos diversos
  - **banco.transacoes** contém as classes relacionadas com transferências e etc
  - **banco.historico** contém as classes relacionadas com extrato
- Leia “classes” como “classes e interfaces”

Pacotes e Diretórios

- banco
  - conta
    - Conta
    - ContaCorrente
    - ContaEspecial
  - investimento
    - CadernetaDePoupanca
    - ContaRemunerada

- banco
  - transacoes
    - Transferencia
    - TransferenciaAgencia
    - Transferencia24hs
  - historico
    - Operacao
    - Historico

Pacotes e o CLASSPATH

- Quando utilizamos vários pacotes em uma mesma aplicação, torna-se necessário incluir o diretório raiz da aplicação no CLASSPATH, caso contrário as classes não serão capazes de localizar umas às outras
- Ou então iniciamos todas as compilações à partir do diretório raiz, referenciando os arquivos *.java pelo path relativo
  - **Windows**
    ```
    > javac banco\conta\ContaCorrente.java
    > javac banco\investimento\*.java
    ```
  - **Linux**
    ```
    $ javac banco/conta/ContaCorrente.java
    $ javac banco/investimento/*.java
    ```

Pacotes e o CLASSPATH

- Na execução temos que indicar o nome completo da classe e não do arquivo *.class

```
> java banco.conta.ContaCorrente
> java banco.investimento.TransacaoAgencia
```

- A opção **-cp** (classpath) dos comandos **java** e **javac** podem quebrar o galho em algumas situações

```
> cd banco\conta
> javac -cp ..\.. *.java
> java -cp ..\.. banco.conta.ContaEspecial
```

Exercício

- Reorganizar as classes do sistema bancário na estrutura de pacotes sugerida, recompilar todas as classes e executar alguns programas de teste
- Será necessário incluir comandos **import** em algumas classes, ex: **banco.transacao.Transferencia** deve importar **banco.conta.Conta**

Extensibilidade e Desacoplamento

- Até agora sempre acessamos métodos e atributos de classes por meio de referências a tipos declarados e conhecidos em tempo de compilação
- Como IDEs, servidores de aplicações e aplicações baseadas em plug-ins interagem com classes cujos tipos não são conhecidos em tempo de compilação?
- A resposta está nos mecanismos de *reflexão* e *introspecção* definidos nos pacotes **java.lang** e **java.lang.reflect**

Instanciando uma Classe dado o seu Nome

- A classe **Class** fornece o método **forName** para carregar a classe nomeda na JVM, retornando uma referência ao *objeto que representa a classe* (lembram que classes também são objetos em Java?)
- Objetos da classe **Class** fornecem métodos para obter referências a construtores (**getConstructor**) e métodos (**getMethod**)
- Podemos ainda relacionar todos os métodos, todos os atributos, saber os valores de retorno, ...

Invocação Dinâmica de Métodos

- Utilizando as classes **Object**, **Class**, **Constructor** e **Method** podemos instanciar qualquer tipo de objeto em tempo de execução, *inclusive objetos de classes inexistentes durante a compilação do programa*, e executar qualquer método suportado pela classe ou por suas instâncias!
- Cuidado com as excessões como **ClassNotFoundException**, **NoSuchMethodException** ou **IllegalAccessException**

Exemplo: Instancia.java

```
Import java.lang.reflect.*;

class Instancia
{
    public static void main (String[] args) throws Exception {
        // obtém uma referência à classe
        Class classe = Class.forName ("java.math.BigDecimal");
        // obtém uma referência a um construtor para a classe
        Class[] tiposConstrutor = { String.class };
        Constructor ctor = classe.getConstructor (tiposConstrutor);
        Object[] argsConstrutor = { "120.5" };
        Object obj = ctor.newInstance (argsConstrutor);

// continua...
```

Exemplo: Instancia.java

```
// continuação...

        // obtém uma referência ao método toString da instância
        Class[] tiposMetodo = { Object.class };
        Method metodo = classe.getMethod ("equals", tiposMetodo);
        //Object[] argsMetodo = { new BigDecimal (args[0]) };
        Object[] argsMetodo = { new java.math.BigDecimal ("120.5") };
        Object result = metodo.invoke (obj, argsMetodo);
        // exibe o resultado
        System.out.println ("O resultado foi " + result);
    }
}
```

6. Metodologias OO

- UML não é metodologia!
- As Metodologias modernas são interativas, evitando a divisão do projeto em etapas estanques e rígidas
- Análise, Projeto, Codificação e Testes são realizados de maneira incrementalmente para cada parte do sistema, de modo a obter feedback do usuário o mais cedo possível

Sugestão de Metologia para Desenvolvimento OO

1. Parta da perspectiva do usuário do seu sistema, e:
   a. Identifique as *entidades* presentes no seu sistema, criando **classes informacionais** correspondentes a cada uma
   b. Identifique as atividades, processos, transações e fluxos de trabalho realizadas pelo sistema, criando **classes de atividade** para cada um
   c. As classes definidas nesta etapa não devem realizar nenhum tipo de interação com o usuário!

Sugestão de Metologia para Desenvolvimento OO

2. Ainda sob a perspectiva do usuário:
    a. Desenhe as telas, páginas, relatórios e formatos de arquivos ou tabelas, encapsulando cada um em sua própria **classe de visualização**
    b. Detalhe a interação do usuário com cada tela, página, etc, criando **classes controladoras** para a interação com cada uma

Sugestão de Metologia para Desenvolvimento OO

3. Agora, sob a perspectiva do desenvolvedor:
   a. Tome um *caso de uso* por vez (um tipo de interação entre o usuário e o sistema)
   b. Revise as definições de classes e interfaces utilizadas por este caso de uso
   c. Escreva casos de teste automatizados para cada requisito deste caso de uso
   d. Implemente as classes e métodos correspondentes a cada requisito
   e. O caso de uso só está terminado quando todos os testes rodarem ok

O Que Não Vimos Ainda...

- Classes aninhadas e anônimas
- Java e código nativo
- Vários pacotes padrão do Java
- Pacotes de extensão para Web Services, computação distribuída, Wireless, Web, ...
- Servidores de aplicação
- Padrões de Projeto
- Outros tipos de diagramas UML
- Outras linguagens que geram bytecodes para a JVM

Referências Recomendadas

- *Thinking in Java*, Bruce Eckel
- *Fundamentos do Desenho Orientado a Objetos com UML*, Meilir Page-Jones
- *Java Design, Building Better Apps & Applets*, Peter Coad & Mark Mayfield
- www.xprogramming.com
- www.agilemodeling.com
- www.junit.org