



Módulos do curso

1. Introdução

2. O Ambiente de Desenvolvimento

3. Sintaxe Básica da linguagem

4. Classes, Objetos, Atributos e Métodos

5. Tipos de Referência, String e Array

6. Herança, Polimorfismo e Ligação Dinâmica

7. Pacotes

8. Modelagem em camadas

9. Classes Abstratas e Interfaces

10. Tratamento de Exceções

11. API Java e Classes Wrappers

Qualiti Software Processes
Java Básico

| 2



Copyright © 2002 Qualiti. Todos os direitos reservados.

Conteúdo Abordado

- ▶ **Módulo 1- Introdução**
 - História da Tecnologia Java
 - Versões da Tecnologia Java
 - Adoção da Tecnologia Java
 - Visão geral da Plataforma Java

- ▶ **Módulo 2- O Ambiente de desenvolvimento**
 - Visão geral do JSDK e seus componentes
 - Conhecendo e Configurando uma IDE (Vídeo Aula)

Copyright © 2002 Qualiti. Todos os direitos reservados.

Conteúdo Abordado

- ▶ Módulo 3- Sintaxe Básica da linguagem
 - Identificadores e tipos primitivos
 - Operadores
 - Conversão de tipos
 - Estruturas de controle
- ▶ Módulo 4- Classes, Objetos, Atributos e Métodos
 - Definição de classes, objetos, métodos e atributos
 - Modificadores de atributos e métodos
 - Criação e remoção de objetos

Copyright © 2002 Qualiti. Todos os direitos reservados.

Conteúdo Abordado

- ▶ Módulo 5- Tipos de Referência, String, Enum e Array
 - Referências
 - Manipulação de Strings
 - Manipulação de tipos Enumerados
 - Manipulação de Array
 - Enhanced For
 - Modelagem do repositório de contas
- ▶ Módulo 6- Herança, Polimorfismo e ligação dinâmica
 - Herança
 - Polimorfismo (Overloading e Override de métodos)
 - Ligação dinâmica (Dynamic Biding)

Copyright © 2002 Qualiti. Todos os direitos reservados.

Conteúdo Abordado

- ▶ Módulo 7 - Modelagem em Camadas
- ▶ Módulo 8 - Classes Abstratas e Interfaces
 - Utilizando classes abstratas em Java
 - Utilizando interfaces em Java
 - Comportamento do operador "instanceof"
 - Classes abstratas vs Interfaces

Copyright © 2002 Qualiti. Todos os direitos reservados.

Conteúdo Abordado

- ▶ Módulo 9- Tratamento de exceções
 - Conceitos básicos sobre tratamento de erro
 - Tipos de Exceções em Java
 - Usando exceções em Java
- ▶ Módulo 10- Pacotes
 - Conceitos sobre Modularização
 - Sintaxe e uso de pacotes em Java

Copyright © 2002 Qualiti. Todos os direitos reservados.

Conteúdo Abordado

- ▶ Módulo 11 - APIs de Java e Classes Wrappers
 - Overview sobre APIs Java
 - Classes Wrappers
 - Conceito de Unbox e Outbox em Java

Copyright © 2002 Qualiti. Todos os direitos reservados.



Módulo 1

Introdução

Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes



Um pouco de história...

- ▶ Origem de Java
 - Java surgiu do projeto Green, em 1991, na Sun Microsystems, que tinha como objetivo prevê qual seria a “próxima onda” em computação.
- ▶ A primeira versão de produção da plataforma foi lançada em 1997, chamada Sparkler (versão 1.1.4).
- ▶ O uso de Java cresceu rapidamente com o crescimento da Internet
- ▶ Atualmente mais de 5.000.000 de desenvolvedores possuem o software de desenvolvimento Java

Copyright © 2002 Qualiti. Todos os direitos reservados.

A tecnologia Java surgiu em um pequeno e secreto projeto da Sun Microsystems chamado “The Green Project” em 1991.

O time de desenvolvimento chamado “Green Team” (um trocadilho com “Dreaming Team” ou time dos sonhos) era composto por 13 pessoas lideradas por James Gosling. Esse time se alojou em um escritório anônimo, longe da sede da Sun, e cortou comunicação regular com a empresa trabalhando por 18 meses tentando prever a “nova onda” em computação. O resultado principal desse projeto foi antecipar a convergência digital entre aparelhos eletrônicos de uso pessoal e computadores (O exemplo mais popular seria os celulares que estão computacionalmente cada vez mais poderosos) e criar uma linguagem independente de plataforma de hardware que tinha como codinome “Oak”.

Como estudo de caso o Green Team desenvolveu um dispositivo que tinha como alvo a indústria de TV digital (que hoje é o foco das atenções de muitas indústrias de eletrônicos). Apesar de tudo isso o projeto se tornou um grande fiasco. A indústria de TV não conseguia enxergar o grande potencial dessa inovação e não estava interessada em “comprar” a idéia. Era uma grande inovação, porém fora de seu tempo.

Nessa mesma época (em torno de 1993), a internet estava sendo introduzida e o James Gosling enxergou que ela possuía as características necessárias para utilização da plataforma criada pelo Projeto Green, era heterogenia em relação as arquiteturas computacionais que se interligavam. Ele uniu forças com a Netscape e habilitou o Netscape Navigator a executar programas em Java. Isso mudou completamente a visão da Sun em relação a Gosling, que provou o valor do projeto.

Em 1995 Java foi anunciada oficialmente em um evento importante e mudou o alvo original da linguagem que era de pequenos dispositivos. Hoje como sabemos Java cobre também a área de pequenos dispositivos com sua plataforma J2ME, que se mostra em franca expansão.

Versões lançadas até hoje

- ▶ JDK 1.1.4 (Sparkler) 12 de setembro de 1997
- ▶ J2SE 1.2 (Playground) 4 de dezembro de 1998
- ▶ J2SE 1.3 (Kestrel) 8 de maio de 2000
- ▶ J2SE 1.4 (Merlin) 13 de fevereiro de 2002
- ▶ J2SE 5.0 (1.5.0) (Tiger) 29 de setembro de 2004
- ▶ Java SE 6 (1.6.0) (Mustang) 11 de dezembro de 2006
- ▶ Java SE 7 (1.7.0) (Dolphin) Previsto para 2010

Copyright © 2002 Quali. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 11



Lista completa das versões lançadas até hoje:

JDK 1.1.4 (Sparkler) 12 de setembro de 1997
JDK 1.1.5 (Pumpkin) 3 de dezembro de 1997
JDK 1.1.6 (Abigail) 24 de abril de 1998
JDK 1.1.7 (Brutus) 28 de setembro 1998
JDK 1.1.8 (Chelsea) 8 de abril 1999
J2SE 1.2.0 (Playground) 4 de dezembro de 1998
J2SE 1.2.1 (sem nome) 30 de março de 1999
J2SE 1.2.2 (Cricket) 8 de Julho de 1999
J2SE 1.3.0 (Kestrel) 8 de maio de 2000
J2SE 1.3.1 (Ladybird) 17 de maio de 2001
J2SE 1.4.0 (Merlin) 13 de fevereiro de 2002
J2SE 1.4.1 (Hopper) 16 de setembro de 2002
J2SE 1.4.2 (Mantis) 26 de Junho de 2003
J2SE 5.0 (1.5.0) (Tiger) 29 de setembro de 2004
Java SE 6 (1.6.0) (Mustang) 11 de dezembro de 2006
Java SE 7 (1.7.0) (Dolphin) Previsto para 2010

Perceba que toda a família 1.x era o chamado Java 1, em 1998 houve mudanças na linguagem que justificaram o nome dado pela Sun de Java 2 usado até pouco tempo. A partir da versão 6 da plataforma, a Sun voltou a denominar a linguagem somente por Java e enfatizando o SE (Standard Edition) e deve ser mantida essa nomenclatura a longo prazo.

Outro ponto que deve ser levado em consideração é que até o lançamento do Tiger (Java 5) os nomes das versões não eram divulgados, hoje parece ser uma tendência dar mais importância a esses nomes em detrimento aos números de versão.

Adoção de Java

- ▶ O número de desenvolvedores Java passou de **3 milhões**
- ▶ Já há mais desenvolvedores Java que desenvolvedores C++
- ▶ Há um forte movimento *OpenSource* com Java
 - Bibliotecas completas
 - Aplicações e ferramentas gratuitas
 - Padrões abertos
- ▶ Java é hoje uma linguagem largamente utilizada para o desenvolvimento de **aplicações desktop, web e para dispositivos móveis.**

Copyright © 2002 Qualiti. Todos os direitos reservados.



Para quem gosta de alguns números, hoje a tecnologia Java já está presente em mais de 2.5 bilhões de dispositivos.

Desses 800 milhões são PCs, mais de 1.2 bilhões de telefones e handhelds e 1.65 bilhões de smart cards. Isso sem contar os inúmeros outros dispositivos como set-top boxes, impressoras, web cams, jogos, sistemas de navegação de carros, terminais de loteria, equipamentos médicos, estações de pagamento de estacionamento, etc.

Adoção de Java

- Algumas empresas que apóiam a Tecnologia Java



Várias empresas fornecedoras de produtos que dão o suporte ao desenvolvimento, distribuição e gerenciamento de aplicativos estão construindo cada vez mais produtos (ferramentas IDEs, servidores de aplicação, entre outros) que dão suporte ao desenvolvimento de aplicações Java.

O apoio dessas empresas, ao desenvolver todo um ferramental que adere à especificação de Java, acaba estimulando o uso da linguagem uma vez que a adoção para o desenvolvimento de aplicações é facilitada com o uso de ferramentas e, por conseguinte, acaba conquistando e abrangendo um número maior de usuários.

A Tecnologia Java

A Tecnologia Java é composta por dois componentes fundamentais:

▸ A linguagem de programação Java

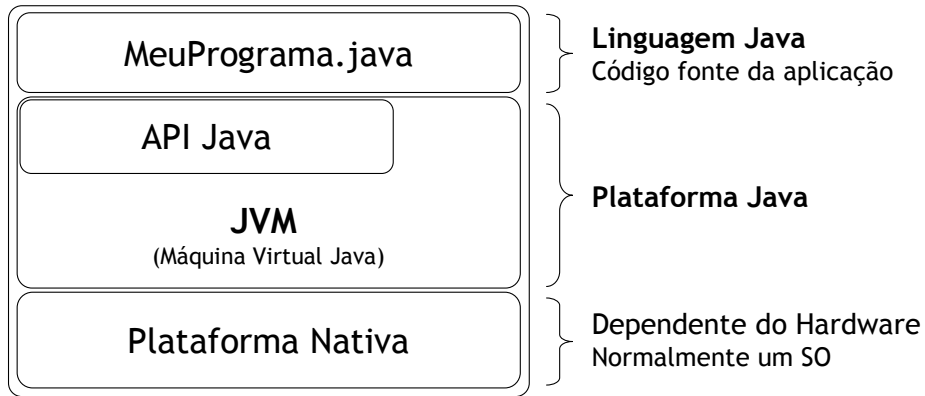
▸ A Plataforma Java

Que por sua vez é composta por:

- Java Virtual Machine - JVM (Máquina Virtual Java)
- Java Application Programming Interface (API Java)

Copyright © 2002 Qualiti. Todos os direitos reservados.

A Tecnologia Java



Copyright © 2002 Qualiti. Todos os direitos reservados.

Todos os componentes da plataforma Java estão envolvidos quando desenvolvemos um programa para ela. O desenvolvedor se utiliza da linguagem de programação em conjunto com a API Java quando escreve seu programa e o executa na JVM que por sua vez traduz as instruções para o código nativo da plataforma onde está sendo executada.

A Linguagem Java

O objetivo principal da linguagem Java é ser Simples, poderosa e eficiente. Gosling definiu algumas características que mostram que java cumpre esses requisitos:

- ▷ Simples e Orientada a Objetos
- ▷ Robusta e Segura
- ▷ Neutra em relação a arquitetura e Portável

Copyright © 2002 Qualiti. Todos os direitos reservados.

1. Simples e Orientada a Objetos

Java foi desenvolvida para ser uma linguagem com uma sintaxe simples, resumida e de fácil aprendizado. Os principais conceitos da linguagem são absorvidos rapidamente pelo desenvolvedor o que em pouco tempo o deixa apto a produção.

Além disso Java é Orientada a Objetos e traz com isso todos os benefícios do paradigma em uma implementação limpa e eficiente, sem preciosismos como Small Talk ou conceitos obscuros como C++.

Java ainda tem a vantagem de ter uma sintaxe base parecida com a de C++ o que torna ainda mais simples de ser aprendida por desenvolvedores que conhecem essa linguagem.

2. Robusta e Segura

A linguagem Java foi desenhada para criar softwares de alta confiabilidade. Ela possui uma checagem extensiva em tempo de compilação junto com uma outra checagem em tempo de execução. O sistema de gerenciamento de memória é extremamente simples (para o desenvolvedor). Objetos são criados pelo operador new e coletados automaticamente pelo Garbage Collector quando já não são mais necessários. Não existem ponteiros, aritmética de ponteiros nem desalocação explícita de memória.

Java é voltada para trabalhar em ambientes de redes e distribuídos o que eleva a importância por segurança e sigilo. Por causa da arquitetura da máquina virtual Java não podem ser invadidos por programas maliciosos. Java, ainda, está sempre atualizada com o estado da arte dos algoritmos e protocolos de criptográficos.

3. Neutra em Relação a Arquitetura e Portável

Quando falamos no conceito Neutra em Relação a Arquitetura queremos dizer que o código compilado Java pode rodar em diversas plataformas de hardware sem que seja preciso modificar ou recompilar o código (Write once, run anywhere). Essa característica é requisito fundamental de sistemas realmente portáveis.

Outra técnica utilizada que permite que java seja completamente portátil é que ela define o tamanho, formato e modelo de memória utilizados pelos seus tipos de dados e esse formato é respeitado por toda implementação da JVM que traduz esses tipos para o sistema operacional subjacente.

A linguagem Java (cont.)

- ▶ Alto Desempenho
- ▶ Interpretada, Multi-Threaded e Dinâmica
- ▶ Distribuída

Copyright © 2002 Quali. Todos os direitos reservados.

4. Alto Desempenho

Para melhorar a performance quando da execução de programas, Java oferece alternativas para eliminar ou diminuir o tempo gasto com a interpretação do bytecode.

Compiladores Just-In-Time (JIT) podem ser utilizados para melhorar a performance de programas Java. Compiladores desta natureza, compilam o código já para a linguagem de máquina sendo utilizada, em vez do bytecode (código neutro). Dessa forma, quando o programa Java for executado, ele não precisará antes ser interpretado para a linguagem de máquina já que este passo foi feito durante a compilação, ao utilizar um compilador JIT.

5. Interpretada, Multi-Threaded e Dinâmica

O interpretador Java é capaz de executar bytecodes em qualquer plataforma computacional. Para isso só é necessário portar o interpretador para essa plataforma. Por ser interpretada a linguagem Java possui um sistema de linkagem simples, incremental e leve. Essas características permitem um ciclo de desenvolvimento (escrita -> teste -> ajuste) muito mais rápido e eficiente que as plataformas convencionais com linkagem estática (como C e C++) onde a compilação e linkagem consomem um tempo considerável. Ainda, por ter linkagem completamente dinâmica, é possível carregar classes que não fazem parte da aplicação original, de forma plug-and-play, fazendo com que novos componentes façam parte do programa sem a necessidade de recompilar o mesmo.

Java possui uma API para desenvolvimento Multi-Thread simples é fácil de utilizar, isso faz com que um recurso extremamente importante esteja mais acessível ao desenvolvedor. Além disso também fornecer um conjunto de mecanismos para permitir o desenvolvimento de aplicações cujos objetos participam ou fazem parte de acesso concorrente e compartilhado. Java é, atualmente, a única linguagem Orientada a Objetos que possui suporte nativo ao tratamento de concorrência.

6. Distribuída

Com Java é possível desenvolver aplicações distribuídas, as quais podem ser executadas em diferentes máquinas e aparelhos ou dispositivos eletrônicos. Esta característica só é possível porque Java fornece APIs específicas (RMI, CORBA, EJB, JDBC, etc) que dão suporte a este tipo de desenvolvimento.

Por exemplo é possível, utilizando Java, desenvolver aplicações corporativas cujas características podem envolver, entre outras, o suporte à Web/Internet, acesso a bancos de dados remotos, arquivos remotos, componentes distribuídos em máquinas remotas, etc.

A plataforma Java

- ▶ O que é a plataforma Java?
 - É uma infra-estrutura para programação baseada no poder das redes de computadores e na idéia de que uma mesma aplicação possa executar em diferentes máquinas, aparelhos e dispositivos eletrônicos.

“Write once, run anywhere.”

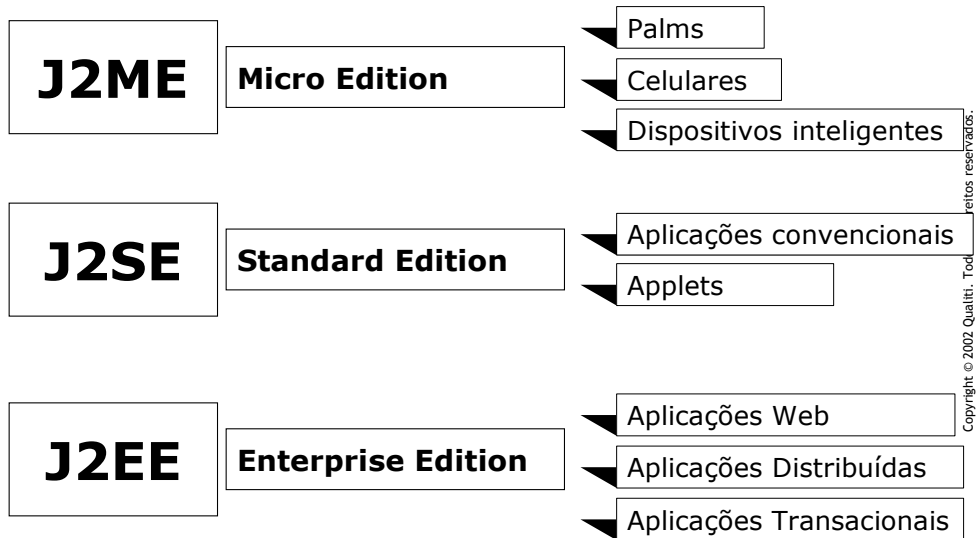
A plataforma Java fornece um conjunto de ferramentas (toolkits) que dão suporte ao desenvolvimento dessas aplicações.

A linguagem Java é formada por um conjunto de APIs (Application Programming Interfaces), que dão suporte ao desenvolvimento de diferentes tipos de aplicações, desde aplicações *stand alone* até aplicações distribuídas em diferentes máquinas, aparelhos e dispositivos.

Outra característica dessa linguagem, é o princípio WORA (*Write Once, Run Anywhere*). Com base neste princípio, é possível escrever um programa na linguagem Java e executá-lo em qualquer plataforma de sistema operacional.

Ao contrário de algumas linguagens de programação, C, por exemplo, cujos programas, dependendo do sistema operacional sendo utilizado, precisam ser recompilados ou mesmo modificados, com Java, uma vez compilado o programa o mesmo pode ser executado em qualquer plataforma de sistema operacional, sem a necessidade de recompilação. Existem diferentes versões do interpretador Java, uma para cada sistema operacional e esta característica, associada a outros fatores, permite que Java adote o princípio WORA.

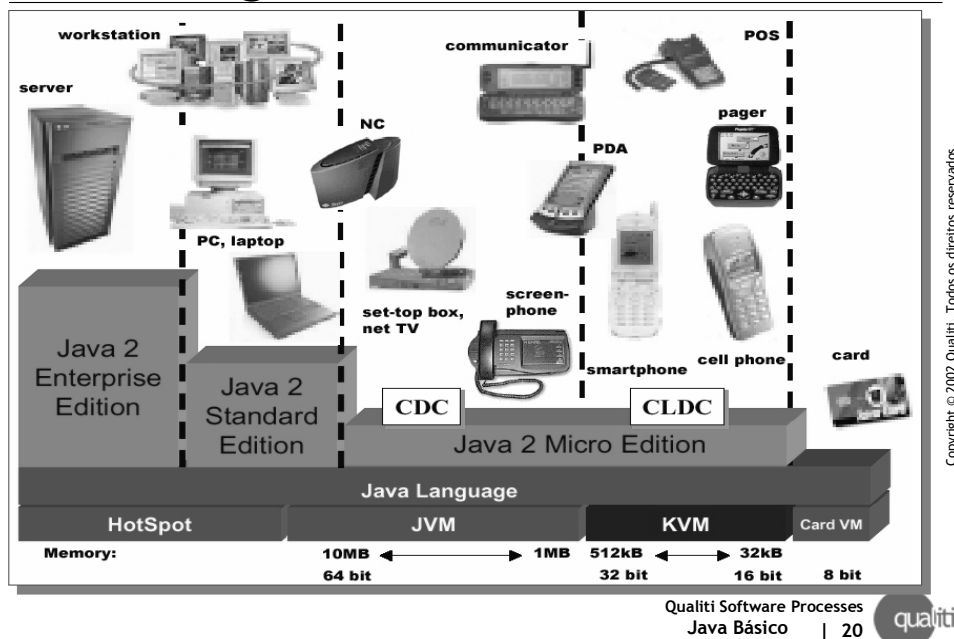
Plataformas Java



A plataforma Java permite o desenvolvimento de diferentes tipos de aplicações (*stand alone*, web e distribuída). Para melhor organizar os recursos (APIs) que devem ser utilizados para cada tipo de desenvolvimento, a plataforma Java categorizou os recursos e os organizou em três plataformas diferentes:

- Java 2 Micro Edition (J2ME), a qual contém APIs para o desenvolvimento de aplicações em executam em dispositivos móveis;
- Java 2 standard Edition (J2SE), a qual é utilizada em aplicações Java convencionais; e
- Java 2 Enterprise Edition (J2EE), contém APIs para o desenvolvimento de aplicações Web, distribuídas e transacionais.

Abrangência da Plataforma Java



Esta figura ilustra a grande variedade de equipamentos e dispositivos, para os quais aplicações Java podem ser desenvolvidas. As plataformas Java estão descritas da direita para a esquerda e os dispositivos associados ao domínio de cada uma também estão especificados.

Da direita para a esquerda, aparecem os dispositivos de menor capacidade para os de maior capacidade de processamento. A divisão em plataformas também segue esse princípio.

Todas as plataformas Java, têm como base a plataforma J2SE:

- A plataforma J2ME consiste de um subconjunto das APIs da plataforma J2SE, mais algumas classes específicas desse domínio. Isto é obvio porque dispositivos, como Palms, smartphones e pagers, são dispositivos com pouca capacidade de processamento e mesmo, pouco espaço para visualização gráfica, portanto não precisam e nem vão utilizar todas as APIs da J2SE.
- A plataforma J2EE consiste de todas as APIs da plataforma J2SE, acrescentando novas APIs, uma vez que aplicações dessa natureza podem apresentar tudo que aplicações Java convencionais apresentam e mais recursos específicos do domínio enterprise tais como, distribuição, aspectos transacionais, entre outros.

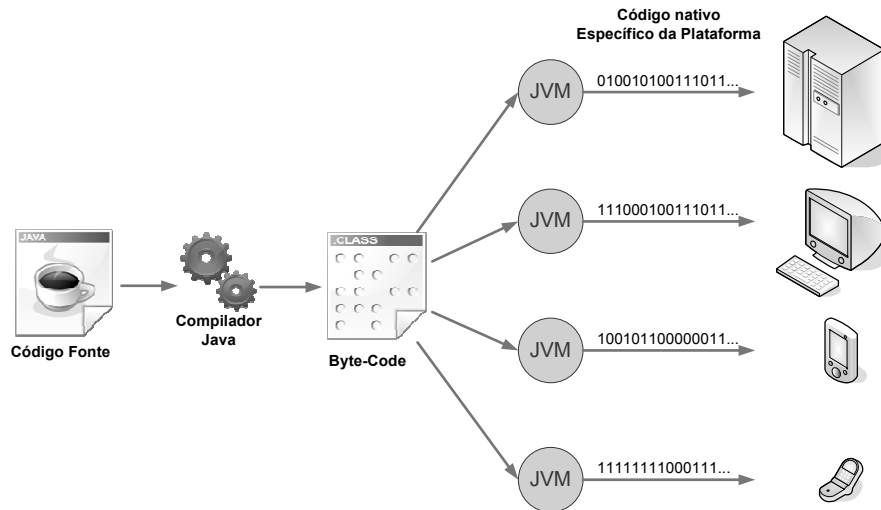
A plataforma Java

- ▶ Java Virtual Machine (JVM)
 - Componente da plataforma Java que assegura a independência das aplicações entre diferentes plataformas
 - É uma espécie de tradutor existente nos dispositivos para traduzir comandos da linguagem Java para a linguagem da máquina nativa.
- ▶ Plataformas compatíveis com Java
 - Implementam o tradutor da linguagem Java para sua linguagem de máquina

Copyright © 2002 Qualiti. Todos os direitos reservados.

Aplicações Java executam dentro de uma máquina virtual Java (JVM - *Java Virtual Machine*). A JVM, nada mais é do que um ambiente dentro do sistema operacional, no qual programas Java são executados.

Como funciona um compilador e um interpretador Java?



Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 22



Depois que um programa Java é desenvolvido, o mesmo precisa ser compilado antes de ser executado. O **compilador** gera um arquivo denominado *bytecode*, o qual é um arquivo neutro e independente de plataforma.

Quando um programa Java é executado, é o arquivo gerado, o *bytecode* que deve ser manipulado, e não o arquivo que contém o código desenvolvido pelo programador.

Como o *bytecode* gerado é independente de plataforma (para garantir a portabilidade), antes de ser executado, o mesmo deve ser interpretado para a linguagem de máquina do sistema operacional no qual o mesmo deve ser executado. Neste caso, entra em cena o **interpretador**, o qual irá traduzir as informações geradas no *bytecode* em informações legíveis para a máquina. Por este motivo, é necessário que o interpretador Java utilizado seja específico para o sistema operacional.

Este processo de interpretação acontece imediatamente antes do código ser executado.

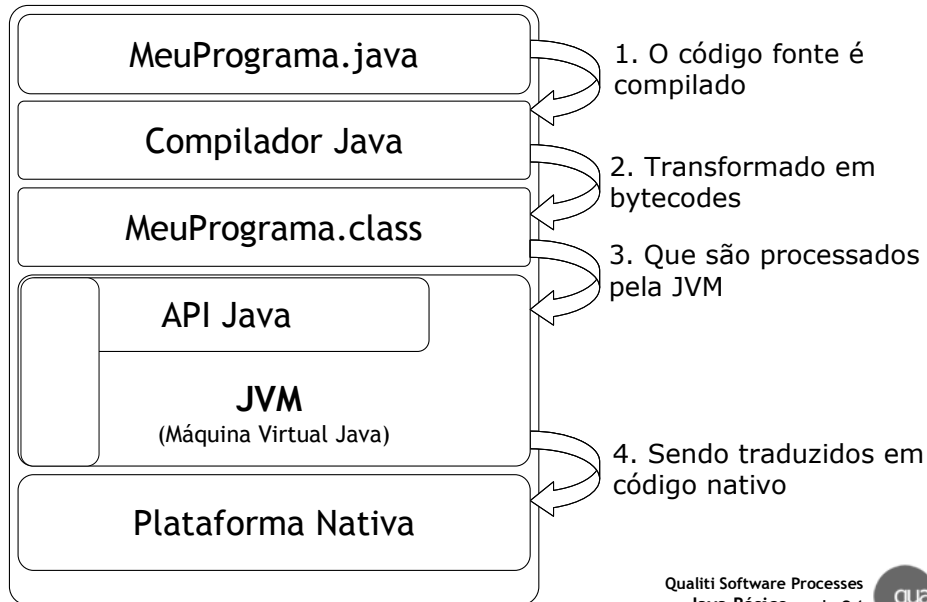
Bytecodes

- ▶ O que são bytecodes?
 - Instruções de código de máquina específicas para a máquina virtual Java
 - Não chega a ser código de baixo nível, chamamos de código intermediário.

- ▶ A máquina virtual transforma os bytecodes em instruções da máquina que está executando o programa

Copyright © 2002 Qualiti. Todos os direitos reservados.

Fluxo da Plataforma Java



Referências do módulo

- ▶ Java Tutorial
 - <http://java.sun.com/docs/books/tutorial/>
- ▶ Java Language Environment
 - <http://java.sun.com/docs/white/langenv/>
- ▶ Java Virtual Machine
 - <http://java.sun.com/javase/technologies/hotspot/>
- ▶ Documentação da Versão 6 da linguagem
 - <http://java.sun.com/javase/6/docs/>
- ▶ Guia sobre a Linguagem Java
 - <http://java.sun.com/javase/6/docs/technotes/guides/language/>
- ▶ Melhorias da Versão 4 e 5
 - <http://java.sun.com/javase/6/docs/technotes/guides/language/enhancements.html>

Copyright © 2002 Qualiti. Todos os direitos reservados.



Módulo 2

O Ambiente de Desenvolvimento

Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes



JSDK

Copyright © 2002 Qualiti. Todos os direitos reservados.

O que é o JSDK?

Java Software Development Kit

- ▶ Conjunto de ferramentas, bibliotecas e exemplos para o desenvolvimento em Java
- ▶ Ferramentas para desenvolvimento de aplicações Java usam o JSDK como base
 - Eclipse, Net beans, IntelliJ IDEA, Borland JBuilder, WSAD, entre outros

Copyright © 2002 Quali. Todos os direitos reservados.

A plataforma Java disponibiliza, além da linguagem, um conjunto de recursos, entre os quais APIs e ferramentas para desenvolvimento. Por este motivo, quando se fala em Java costuma-se utilizar o termo JSDK - Java Software Development Kit.

As ferramentas disponibilizadas no JSDK são, na realidade, ferramentas de linha de comando. Para tornar o desenvolvimento Java mais produtivo, existem algumas ferramentas Gráficas no mercado (outras também *free*), as chamadas IDEs (Integrated Development Environments), as quais proporcionam uma interface gráfica para edição e manipulação de programas Java. Estas ferramentas todas utilizam o JSDK como base para permitir o desenvolvimento Java.

Entre as IDEs mais utilizadas no mercado, podem ser citadas: Eclipse, Netbeans, Borland Jbuilder, IntelliJ IDEA, WSAD, entre tantas outras. Atualmente existe uma distribuição do JSDK integrado com a IDE Netbeans distribuído oficialmente pela Sun, essa distribuição é gratuita.

Ferramentas e Recursos disponibilizados com o JSDK

- ▶ Ferramentas para compilação, execução e depuração de programas Java
- ▶ Bibliotecas compiladas
- ▶ Código fonte completo das bibliotecas
- ▶ Exemplos em Java

Copyright © 2002 Qualiti. Todos os direitos reservados.



Juntamente com o JSDK, podem também ser encontrados códigos exemplo para teste e o código fonte completo das classes que constituem a linguagem Java.

Exemplos de Ferramentas do JSDK

▶ Essenciais

- **javac**: compila programas Java
- **java**: executa programas Java

▶ Importantes

- **javadoc**: gera documentação automática
- **jar**: manipula arquivos “Java Archive” (JAR)

▶ Outras

- **javap**
- **appletviewer**

Copyright © 2002 Qualiti. Todos os direitos reservados.



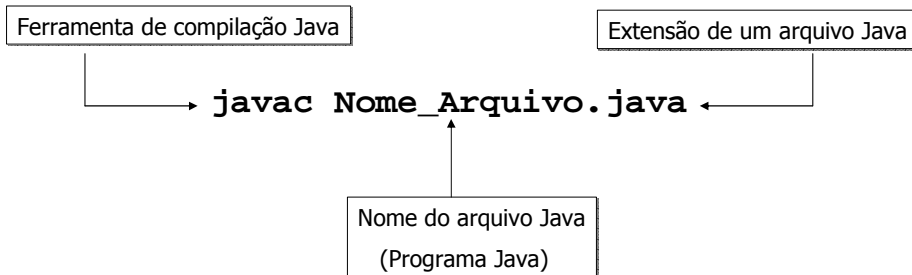
O JSDK disponibiliza uma grande quantidade de ferramentas com diferentes propósitos:

- Existem àquelas que são indispensáveis e devem ser utilizadas SEMPRE: java e javac;
- Àquelas que são importantes, utilizadas com certa frequência: javadoc e jar; e
- Àquelas que existem, mas não são utilizadas com muita frequência: javap, appletviewer, etc.

Os próximos slides detalham a função e comportamento de cada uma. Todas as ferramentas do JSDK são utilizadas via linha de comando, por este motivo, nos próximos slides não é feita distinção entre ferramenta e comando.

Comando “javac”

- Compila arquivos **.java** transformando-os em **.class** (bytecodes)



Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 31



O comando **javac** deve ser utilizado para compilar programas Java:

```
javac Nome_Arquivo.java
```

onde, **javac** representa o compilador; **Nome_Arquivo** é o nome do programa java. Um programa Java deve ter a extensão **.java**.

Após a execução desta linha de comando, o computador irá gerar o *bytecode*, que possui o mesmo nome do arquivo Java, mas a extensão é diferente(**.class**).

Assim, para o exemplo abaixo:

```
javac Endereco.java
```

O compilador irá gerar um arquivo bytecode, denominado **Endereco.class**.

É este arquivo que será manipulado pelo interpretador Java.

Um programa Java pode ser compilado através de diferentes linhas de comando como por exemplo:

- **javac Nome_Arquivo.java** → compila um arquivo especificado pelo nome.
- **javac *.java** → compila todos os arquivos **.java** localizados no mesmo diretório.
- **javac Nome_Arquivo1.java Nome_Arquivo2.java** → compilação de mais de um arquivo Java específico.

Comando “java”

- ▷ Interpretador Java
- ▷ Utilizado para:
 - Executar arquivos **.class** (gerados após a compilação, a partir dos arquivos **.java**)
 - Executar arquivos **.jar**
- ▷ Um arquivo Java precisa conter um método **main** para poder ser executado:

```
public static void main (String[] args) {  
  
    /*código Java que deve ser executado */  
  
}
```

O comando **java** é utilizado para interpretar o arquivo **.class** gerado após a compilação e, por conseguinte executar o programa Java.

Para que um arquivo Java possa ser executado, o mesmo deve conter um método **main**, o qual indica ao interpretador que o arquivo em questão, além de ser interpretado para código de máquina, também precisa ser executado. Dentro deste método **main** devem ser descritos os comandos que devem ser executados. Por exemplo:

```
public static void main (String args[]){  
    System.out.println("Imprimir esta mensagem na tela!!");  
}
```

Todo método **main** deve ter o formato descrito acima, a única coisa que varia de um programa para outro é o conteúdo deste método. No exemplo citado, quando o arquivo que contém este método for executado, através do comando **java**, a mensagem “Imprimir esta mensagem na tela” deve aparecer na linha de comando.

Exemplos do comando “java”

Interpretador Java

`java Nome_Arquivo`

Nome do arquivo
(a extensão .class
não deve ser fornecida)

`java Conta`

`java -jar bancoDatasus.jar`

Qualiti Software Processes
Java Básico | 33



Copyright © 2002 Qualiti. Todos os direitos reservados.

O formato para a utilização do comando `java` é como a seguir:

`java Nome_Arquivo`

Como já explicado, o arquivo manipulado pelo interpretador é o .class gerado após a compilação. Quando este arquivo é utilizado pelo interpretador, a extensão .class é desconsiderada. Por este motivo, não é necessário especificá-la na linha de comando. Por exemplo:

`java Endereco`

Este comando irá executar o arquivo Endereco.class gerado durante a compilação (realizada a partir do arquivo Endereco.java).

Outras opções podem ser utilizadas em conjunto com o interpretador. Por exemplo, para visualizar a versão da máquina virtual utilizada, o comando

`java -version`

pode ser utilizado.

Comando “javadoc”

- ▶ Extrai toda a documentação no estilo “javadoc” (`/** ... */`)
- ▶ Gera um site completo com a documentação

Ferramenta para geração de documentação html

→ **javadoc** Nome_Arquivo.java

Nome do arquivo com a extensão .java

Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 34



A documentação de programas, independente da linguagem que são escritos, é uma prática muito importante para permitir a organização dos arquivos de código e mesmo, proporcionar um mecanismo para garantir a compreensão do código escrito (a partir de comentários realizados no mesmo).

Java oferece uma ferramenta para gerar documentação de código no formato HTML. Neste caso, as informações geradas na documentação são obtidas a partir de comentários realizados no próprio código. O que Java possibilita é a melhor organização destas informações, documentando-as no formato HTML. A geração deste documento é feita automaticamente com o auxílio da ferramenta **javadoc**.

Este comando pode possuir o seguinte formato:

```
javadoc Nome_Arquivo.java
```

Onde, **javadoc** é a ferramenta de geração de documentos HTML e **Nome_Arquivo.java** corresponde ao arquivo para o qual a documentação HTML deve ser gerada e de onde as informações devem ser extraídas.

Comando “jar”

Para criar um arquivo .jar

```
jar cf arquivo.jar *.class  
jar cf arquivo.jar *.java
```

Para listar o conteúdo de um arquivo .jar

```
jar tf arquivo.jar
```

Para extrair o conteúdo de um arquivo .jar

```
jar xf arquivo.jar
```

Copyright © 2002 QualiTi. Todos os direitos reservados.

A ferramenta `jar` é utilizada para empacotar arquivos Java. Em geral, arquivos `.jar` empacotam arquivos `.class` somente. O empacotamento de arquivos `.class` pode ser útil quando se deseja disponibilizar um programa para execução, mas o código não precisa ser fornecido uma vez que nenhuma alteração será realizada no mesmo. De outra forma, um `.jar` pode conter também arquivos `.java`.

Outra utilização de empacotamento `jar` é quando se deseja disponibilizar um conjunto de classes para reutilização por outras aplicações. Estas outras aplicações não irão manipular diretamente essas classes empacotadas e nem vão “executar” essas classes, mas sim, utilizá-las para o desenvolvimento de alguma aplicação. Neste caso, essas classes empacotadas servem como bibliotecas que podem ser utilizadas por outras aplicações.

Outras ferramentas

- ▶ Existem muitas outras ferramentas disponibilizadas no SDK.
- ▶ Hoje quase não é mais necessário trabalhar com elas via linha de comando já que as IDEs disponibilizam uma interface gráfica para usar essas ferramentas.
- ▶ Uma lista completa das ferramentas e de suas opções de uso pode ser encontrada em: <http://www.java.sun.com/>

Copyright © 2002 Qualiti. Todos os direitos reservados.



Configurando uma IDE

- ▶ Uma IDE automatiza muito do trabalho de compilação, debug, execução e empacotamento de uma aplicação.
- ▶ Principal objetivo: Aumentar a produtividade no desenvolvimento de programas.
- ▶ Dividiremos a tarefa de montar o ambiente em 4 passos:
 1. Download e Instalação das ferramentas.
 2. Conhecendo nossa IDE.
 3. Criando um projeto.
 4. Criando, compilando e executando nosso primeiro programa Java.

Copyright © 2002 QualiTi. Todos os direitos reservados.



Referências do módulo

- ▶ Java Tutorial
 - <http://java.sun.com/docs/books/tutorial/>
- ▶ Java Language Environment
 - <http://java.sun.com/docs/white/langenv/>
- ▶ Java Virtual Machine
 - <http://java.sun.com/javase/technologies/hotspot/>
- ▶ Parâmetros de execução da JVM
 - <http://java.sun.com/javase/technologies/hotspot/vmoptions.jsp>
- ▶ Documentação da Versão 6 da linguagem
 - <http://java.sun.com/javase/6/docs/>

Copyright © 2002 Qualiti. Todos os direitos reservados.


Módulo 3

Sintaxe básica da linguagem

Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes





~~Comentários~~

```
/**
 * Este é um comentário especial do tipo "javadoc" que é usado para
 * geração automática de documentação
 */
-----
-----
-----
```


código Java

comentários Java

arquivo Java

- ▷ // Este é um comentário de uma única linha
- ▷ /* Este comentário pode ocupar várias linhas sem problemas */
- ▷ /** Este é um comentário especial do tipo
* "javadoc" que é usado para geração
* automática de documentação
*/

Qualiti Software Processes
Java Básico | 40



Copyright © 2002 Qualiti. Todos os direitos reservados.

O uso de comentários é fundamental para a documentação do código sendo desenvolvido. Os mesmos são úteis para tornar claros ou explicar aspectos do código que não são tão óbvios, por exemplo: explicação de código complexo ou não-usual, explicação de peculiaridades dos requisitos, explicação do porque o programa utiliza uma solução em vez de outra (mais convencional), sugestão de possíveis maneiras para incrementar ou modificar o programa, realização de advertências sobre certas "armadilhas" existentes no programa, por exemplo: a modificação em uma variável, pode implicar na modificação de alguma outra parte do código.

Seja qual for o motivo de um comentário, este não deve ser muito extenso. Comentários devem ser simples, explicativos e curtos (sem impactar no entendimento daquilo que o mesmo se propõe a explicar).

De modo a auxiliar na inclusão desses aspectos, Java fornece alguns mecanismos para documentação de código, representado por três formas de comentários:

- // → comentário de uma linha;

- /* ... */ → comentário de bloco; e

- /**
* → Comentário javadoc
*/

~~Identificadores~~

- ▶ Identificam elementos de um programa Java
 - Variáveis, métodos, atributos, rótulos, ...
- ▶ Regras para identificadores
 - Devem iniciar por uma letra, um “underscore” (`_`) ou cifrão (`$`). Caracteres subsequentes podem ser letras, dígitos, sublinhados ou `$`.
 - São “*Case sensitive*”:
 - Maiúsculas são diferenciadas de minúsculas

Copyright © 2002 QualiTi. Todos os direitos reservados.

Identificadores representam os nomes atribuídos aos elementos que fazem (ou podem fazer) parte de um programa Java, entre eles, atributos, variáveis, rótulos, etc.

Existem regras específicas em Java que devem ser seguidas para a definição de identificadores. Por exemplo:

- Nomes de identificadores devem começar com letras, `_` (*underscore*) ou `$`.
- Identificadores Java são *case sensitive*, ou seja, “idade” é diferente de “Idade” que, por sua vez, também é diferente de “IDADE”.

~~Identificadores~~

► Identificadores válidos

- soma
- temp01
- _numClientes
- \$fortuna
- nomeLongoDeVariavel

► Identificadores inválidos

- 102dalmatas
- 123
- #x

Copyright © 2002 Qualiti. Todos os direitos reservados.



~~Palavras reservadas~~

abstract	continue	for	new	switch
assert***	default	goto*	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum****	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp**	volatile
const*	float	native	super	while

* Não usado
** Adic. em 1.2
*** Adic. em 1.4
**** Adic. em 5.0

true
false
null } **Não são palavras reservadas**

Qualiti Software Processes
Java Básico | 43



Copyright © 2002 Qualiti. Todos os direitos reservados.

Palavras reservadas em Java não podem ser utilizadas como nome de identificadores. Elas formam o conjunto de “construções ou elementos” definidos pela linguagem, que podem ser utilizados em conjunto os elementos (atributos, métodos, etc.) definidos pelo programador.

Essa regra também vale para “true”, “false” e “null”. Eles não são palavras reservadas e sim literais predefinidos (uma espécie de constantes globais) não podendo ser usados como identificadores.

~~Tipos Primitivos~~

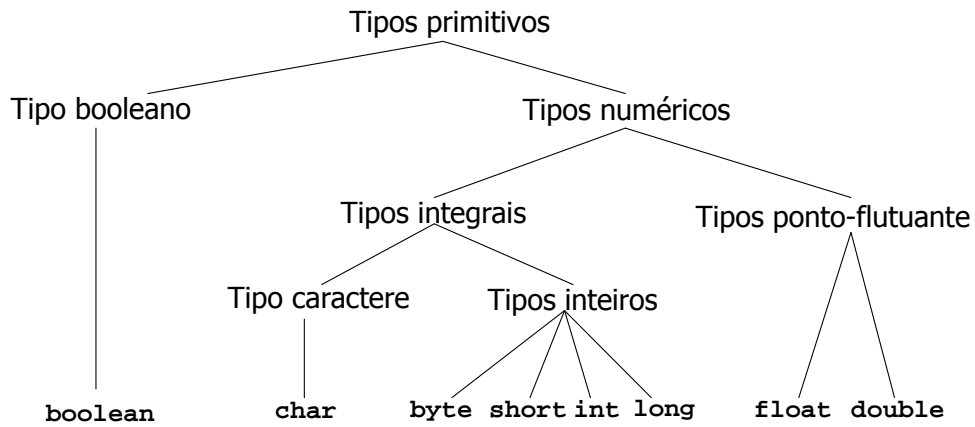
boolean	true ou false
char	caractere (16 bits Unicode)
byte	inteiro (8 bits)
short	inteiro (16 bits)
int	inteiro (32 bits)
long	inteiro (64 bits)
float	ponto flutuante (32 bits)
double	ponto flutuante (64 bits)

Copyright © 2002 QualiTi. Todos os direitos reservados.

A linguagem Java define 8 tipos primitivos, cada qual com diferentes tamanhos e valores permitidos:

- *boolean*: somente pode conter os valores *true* ou *false*, apesar de não se poder definir ao certo (depende da implementação da VM) normalmente ocupam apenas 1 bit em memória.
- *char*: especifica dados com tamanho de no máximo 16 bits e seus valores podem ser especificados usando um caractere entre aspas simples 'w' ou como dígitos hexadecimal Unicode, os quais são precedidos por \u, 'u4567'.
- *byte*, *short*, *int* e *long*: são tipos primitivos integrais com tamanhos de no máximo 8, 16, 32 e 64 bits, respectivamente. Por exemplo: 57, 100, 300 e 1234, respectivamente. O valor de um tipo primitivo *long* pode também ser realizado utilizando a letra "l" (maiúscula ou minúscula) ao final, por exemplo 10l ou 10L.
- *float* e *double*: representam números ponto-flutuante com tamanho de no máximo 32 e 64 bits, respectivamente. Exemplo de um valor *float*, 10.2f; exemplo de um valor *double* 21.7. Os valores ponto flutuante de Java obedecem ao padrão IEEE 754 32-bits e 64-bits respectivamente.

~~Hierarquia dos Tipos~~



Copyright © 2002 Qualiti. Todos os direitos reservados.

~~Declaração de Variáveis~~

```
int LIMITE_MAXIMO;
```

```
double saldo = 100.5;
```

```
float saldo = 100.5f;
```

```
int quantidade, idade;
```

O "f" nesse caso é necessário pois literais ponto flutuante são por padrão do tipo double.

► Padrão para nome de variáveis:

► começam com letras minúsculas;

► em caso de palavras compostas a primeira letra da palavra seguinte é maiúscula.

Padrão de codificação. Não é restrição da sintaxe de Java.

```
int quantidadeMaxima;
```

Qualiti Software Processes
Java Básico | 46



Copyright © 2002 Qualiti. Todos os direitos reservados.

Uma declaração de variável Java deve ser precedida do tipo correspondente à variável sendo declarada. Ao final, o ; (ponto-e-vírgula) deve ser acrescentado para finalizar a declaração. Por exemplo:

```
int idade;
```

Variáveis de um mesmo tipo podem ser declaradas de uma única vez:

```
double saldo, salario;
```

Ou também podem ser inicializadas (valores literais atribuídos à variável), durante a sua declaração. Por exemplo:

```
int idade = 25;
```

~~String~~

- ▶ Em Java é uma classe e não um tipo primitivo
- ▶ Suporte especial dado pela linguagem permite tratar Strings como tipo primitivo
- ▶ //Pode ser inicializada como tipo primitivo
`String s = "Olá pessoal!";`
- ▶ No decorrer do curso veremos mais detalhes sobre sua utilização

Copyright © 2002 QualiTi. Todos os direitos reservados.

Tipos Enumerados

- ▶ Conceito adicionado a linguagem a partir da versão 5.0 (Tiger).
- ▶ Assim como Strings são classes e não tipos primitivos.
- ▶ Também possuem suporte especial da linguagem para que seja tratado como tipo primitivo.
- ▶ No decorrer do curso estudaremos mais sobre esse tipo.

Copyright © 2002 Qualiti. Todos os direitos reservados.

Operadores

Copyright © 2002 Qualiti. Todos os direitos reservados.

Tipos de operadores

- ▷ Aritméticos
- ▷ Concatenação
- ▷ Relacionais
- ▷ Lógicos
- ▷ Atribuição
- ▷ Unários
- ▷ Condicional (ternário)

Copyright © 2002 Qualiti. Todos os direitos reservados.



~~Operadores aritméticos~~

+ - * / %

- ▶ O operador / é também utilizado para calcular divisões inteiras
- ▶ O operador % calcula o resto de uma divisão inteira

$1/2 \Rightarrow 0$

$16\%5 \Rightarrow 1$

$1\%2 \Rightarrow 1$

$16/5 \Rightarrow 3$

Qualiti Software Processes
Java Básico | 51



Copyright © 2002 Qualiti. Todos os direitos reservados.

Os operadores aritméticos em Java equivalem aos operadores já conhecidos da matemática convencional (adição, subtração, etc.), com a diferença que operadores aritméticos em Java devem seguir as regras de limitação para representação de números em computadores.

Em Java divisão de um número inteiro por 0, pode gerar uma exceção (*ArithmeticException*). Divisão entre dois números inteiros, em Java, tem como resultado valores integrais, ou seja, se houver uma fração resultante da divisão entre dois números inteiros, a mesma é desconsiderada pelo interpretador Java. Por exemplo, a seguinte divisão $3/2$ resulta em 1, em vez de 1,5.

O operador de módulo (%) tem como resultado o resto de uma “divisão inteira”. Por exemplo, o resultado da seguinte expressão $15\%4$ resulta em 2 ($15/4 = 3$, resto 2); do mesmo jeito $1\%2$ resulta em 1 ($1/2 = 0$, resto 1).

O operador de subtração é utilizado como na matemática convencional.

O operador de soma também é utilizado como na matemática convencional, se aplicado à tipos primitivos numéricos.

~~Operador de concatenação~~

► + (aplicado a Strings)

```
String nomeCompleto = nome + sobrenome;
```

A concatenação também faz uma conversão implícita para String

```
mensagem = "Este é o cliente número " + x;
```

```
System.out.println("Total: " + total);
```

Qualiti Software Processes
Java Básico | 52



Copyright © 2002 Qualiti. Todos os direitos reservados.

O operador aritmético “+” pode também ser utilizado como um operador de concatenação trabalhando com uma String como se fosse um tipo primitivo.

O operador de concatenação é utilizado para concatenar Strings. Por exemplo:

```
int idade = 15;  
String mensagem = "A idade do cliente = " + idade;
```

Se a variável mensagem for impressa na tela, a saída será igual à “**A idade do cliente = 15**”. Isto quer dizer que, em Java, “qualquer coisa” que for concatenada com uma String, será transformada ao final em uma String também. Por exemplo:

```
String valor = 10+"2";  
resulta em 102.
```

Operadores Relacionais

► Operadores Relacionais

>	Maior que
<	Menor que
>=	Maior que ou igual
<=	Menor que ou igual
==	Igual
!=	Diferente (ou <i>não igual</i>)

Copyright © 2002 QualiTi. Todos os direitos reservados.

Operadores de comparação sempre têm com o tipo de retorno um resultado booleano (*true* ou *false*). Por exemplo, para os valores

```
int p = 7;  
int q = 20;
```

os seguintes testes

```
p < q  
q > 7  
p != q
```

irão retornar como resultado o valor *true*.

Operadores lógicos

- ▶ Operadores booleanos
 - short-circuit (Avaliação parcial)
 - && (E lógico)
 - || (OU lógico)

Copyright © 2002 QualiTi. Todos os direitos reservados.

Operadores booleanos são divididos em duas categorias : *short-circuit* e *bitwise*. Uma das diferenças entre os dois é que o primeiro é aplicado somente a valores booleanos.

• *short-circuit*: são dois os tipos de operadores *short-circuit*: && (AND lógico) e || (OR lógico). O fato deste operador ser aplicado somente a tipos booleanos possibilita que uma operação lógica possa ser simplificada (reduzida) uma vez que é possível determinar, antecipadamente, a partir do valor da expressão do lado esquerdo, o resultado final da mesma. Por exemplo:

- ***false*** AND X = ***false***
- ***true*** OR X = ***true***

Neste caso, a expressão do lado direito não precisaria ser calculada, uma vez que o resultado da mesma não afetaria o resultado final da operação lógica. Por outro lado, usando estes operadores, caso o operando do lado esquerdo, não defina o valor total da expressão, o operando do lado direito é calculado.

É importante tomar cuidado com uso deste operador se o valor resultante da expressão do lado direito causar efeitos colaterais ao código sendo processado.

Operadores lógicos

- ▶ Operadores booleanos
 - Bitwise (Avaliação completa)
 - & (E lógico ou bit-a-bit)
 - | (OU lógico ou bit-a-bit)
 - ^ (OU-EXCLUSIVO bit-a-bit)

Copyright © 2002 QualiTi. Todos os direitos reservados.

Operadores *bitwise* são aplicados somente a tipos integrais de dados e são compostos dos operadores & (AND)', ^ (Exclusive-OR) e || (OR). Quando utilizados sobre tipos integrais, o cálculo da operação é realizado bit a bit.

- 1 AND 1 = 1, qualquer outra combinação produz 0.
- 1 XOR 0 ou 0 XOR 1 = 1, qualquer outra combinação produz 0.
- 0 OR 0 = 0, qualquer outra combinação produz 1.

Esses operadores também podem ser aplicados sobre tipos booleanos. O comportamento deles com tipos booleanos é exatamente o mesmo realizado com tipos integrais, a diferença é que o valor dos operandos é tratado como um único bit: **true** corresponde ao bit 1 e **false** corresponde ao bit 0.

Ao contrário dos operadores *short-circuit*, o operando do lado direito é SEMPRE calculado, o que evita problemas quando a expressão gera efeito colateral.

Atribuição

► Atribuição

=
+=, -=, *=, /=

`x = 0;`

`x += 1;`



`x = x + 1;`

`a = b = c = -1;`

`y -= k;`



`y = y - k;`

`y -= x + 5;`



`y = y - (x + 5);`

Qualiti Software Processes
Java Básico | 56



Copyright © 2002 Qualiti. Todos os direitos reservados.

O operador de atribuição deve ser utilizado quando se deseja atribuir o valor de uma variável ou expressão a outra variável. Por exemplo:

`x = 5;`

Onde, o valor ou a expressão do lado direito é atribuída à variável do lado esquerdo.

Existem variações desse operador, úteis para simplificar a escrita de um atribuição quando o valor atribuído à uma variável corresponde ao valor da própria variável e mais outros aspectos. Por exemplo, em vez de escrever `x = x + 5;` em Java esta mesma expressão pode ser escrita da seguinte forma:

`x += 5;`

Para os operadores aritméticos -, +, * e /, existe uma forma simplificada de uso (-=, +=, *=, /=). Por exemplo:

`x *= 25;`

equivale à

`x = x * 25;`

`y -= x + 5;`

equivale à

`y = y - (x + 5);`

Unários

▶ ++, --

```
y = ++x
```

```
y = --x
```

```
y = x++
```

```
y = x--
```

Avaliar bem a ordem
do operador em relação
ao operando

▶ -, !

```
y = -x
```

```
y = !x
```

Copyright © 2002 QualiTi. Todos os direitos reservados.

Os operadores unários ++ e - são utilizados para modificar o valor de uma expressão acrescentando ou subtraindo o valor da mesma em uma unidade. Por exemplo:

```
x = 5;
```

```
y = ++x;
```

equivale a $y = x + 1 \Leftrightarrow y = 5 + 1$, que resulta no valor 6

```
y = --x
```

equivale a $y = x - 1 \Leftrightarrow y = 5 - 1$, que resulta no valor 4

No primeiro exemplo, o valor de x é primeiramente incrementado para, somente depois, ser atribuído à y. Da mesma forma, no segundo exemplo, o valor de x é primeiramente decrementado, para depois ser atribuído à y. Este processamento ocorre desta forma porque o operador unário é declarado ANTES da expressão a ser avaliada. Por este motivo, estes operadores são chamados de pré (incremento e decremento).

Estes operadores podem ser utilizados ainda como operadores de pós (incremento e decremento), os quais primeiramente atribuem o valor da expressão à variável do lado esquerdo para, somente DEPOIS, incrementar ou decrementar o valor da expressão. Por exemplo:

```
x = 5;
```

```
z = x++;
```

equivale a $y = x \Leftrightarrow y = 5$, x resulta em 6

```
z = x--
```

equivale a $y = x \Leftrightarrow y = 5$, x resulta em 4

O operador (-) pode ser utilizado para negar uma expressão: $y = - (2+x)$

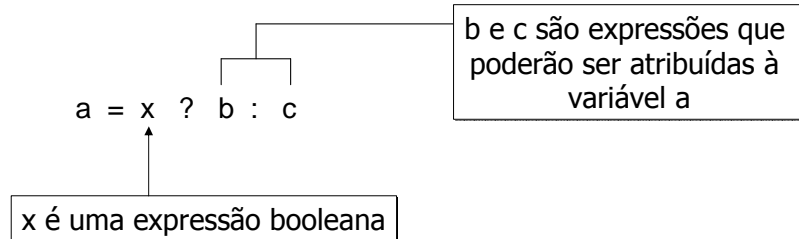
O operador (!) é utilizado para inverter o valor de uma expressão booleana:

```
int x = 7;
```

```
boolean resultado = ! (x < 4), resulta em true
```

~~Condicional~~

- Conhecido também como operador ternário
 - ?:



Copyright © 2002 QualiTi. Todos os direitos reservados.

Este operador, na realidade, pode ser utilizado como uma forma simplificada de expressões condicionais `if/else`. A semântica é a mesma, com a diferença que, dependendo se a condição for satisfeita, a ação a ser realizada deve gerar um resultado e o mesmo deve ser atribuído à uma variável. Por exemplo:

```
if (x) {  
    a = b;  
}else {  
    a = c;  
}
```

equivale à `a = x ? b : c`

Ou seja, se a condição `x` (booleana) for satisfeita, a variável `a` recebe o valor da expressão `b`, do contrário recebe o valor da expressão `c`. Portanto, os tipos das expressões `a`, `b` e `c` devem ser compatíveis para que a atribuição possa ser realizada.

Uso dos Operadores

- ▶ Ordem de avaliação dos operadores
- ▶ Associatividade

Copyright © 2002 Qualiti. Todos os direitos reservados.



Ordem de avaliação dos operadores

► Ordem de precedência (maior para menor):

- Unário Posfixado `expr++ expr--`
- Unário Prefixado `++expr --expr +expr -expr ~ !`
- Multiplicativos `* / %`
- Aditivos `+ -`
- Shift `<< >> >>>`
- Relacionais `< > <= >= instanceof`
- Igualdade `== !=`
- Bit-a-bit AND `&`
- Bit-a-bit exclusive OR `^`
- Bit-a-bit inclusive OR `|`
- Lógico AND `&&`
- Lógico OR `||`
- Ternário `? :`
- Atribuição `= += -= *= /= %= &= ^= |= <= >= >>=`

Copyright © 2002 Qualiti. Todos os direitos reservados.

Associatividade

- ▶ Quando os operadores possuem a mesma precedência, avalia-se primeiro o operador mais a esquerda
 - Exemplo: $a + b + c$ equivale a $(a + b) + c$
- ▶ (exceção) Todos os operadores binários de Java são associativos a esquerda, exceto a atribuição
 - Exemplo: $a = b = c$ equivale a $a = (b = c)$
- ▶ Precedência e associatividade podem ser redefinidas através de parênteses
 - Exemplo: $a * (b + c)$, $a + (b + c)$

Copyright © 2002 QualiTi. Todos os direitos reservados.



Para atribuições, os operadores são associativos da direita para a esquerda. Desta forma,

$a = b = c$ **NÃO** equivale à $(a = b) = c$

A última expressão gera um erro de compilação. Desta forma,

$a = b = c$ **EQUIVALE** à $a = (b = c)$

Conversão de Tipos

Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes



~~Introdução~~

- ▶ Conversões entre tipos, e *Casts*, acontecem freqüentemente quando programamos em Java

```
double d1 = 10.0d;  
System.out.println("Soma: " + (d1 + 10));  
...  
byte b = (byte) 32.0d;  
...
```

Copyright © 2002 Qualiti. Todos os direitos reservados.

As vezes, quando trabalhando com tipos de dados é necessário realizar algumas conversões entre tipos diferentes, de modo que o programa possa se comportar da maneira desejada.

Java oferece mecanismos para permitir este tipo de manipulação.

~~Conversões ocorrem~~

- ▶ Entre tipos primitivos
 - Atribuição
 - Passagem de parâmetros
 - Promoções aritméticas

- ▶ Entre objetos
 - Atribuição
 - Passagem de parâmetros
 - Inbox e Outbox nas conversões implícitas de primitivos em classes e classes em primitivos (Veremos mais detalhes a diante).

Copyright © 2002 QualiTi. Todos os direitos reservados.

Java permite conversão entre seus diferentes tipos de dados:

- Conversão entre tipos primitivos (int, log, etc.)
- Conversão entre tipos referência (objetos Java, vistos nos próximos módulos)

Para prover esse mecanismo de conversão, Java introduz três abordagens: conversão por atribuição, conversão por passagens de parâmetros e conversão através de promoção aritmética, esta última aplicada somente a tipos primitivos.

~~Conversões entre tipos primitivos~~

- ▶ Widening conversion
 - Conversão para um tipo de maior capacidade
- ▶ Narrowing conversion
 - Conversão para um tipo de menor capacidade
 - Pode haver perda de informação
- ▶ Essas conversões podem ser
 - Implícitas
 - Explícitas

Copyright © 2002 Qualiiti. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 65



Conversões entre tipos primitivos podem acontecer de forma implícita ou explícita, dependendo do tipo atribuído e do tipo que está recebendo o atributo.

Se a conversão for realizada para um tipo de maior capacidade, por exemplo, de int para long, a conversão é implícita e é denominada *widening conversion*:

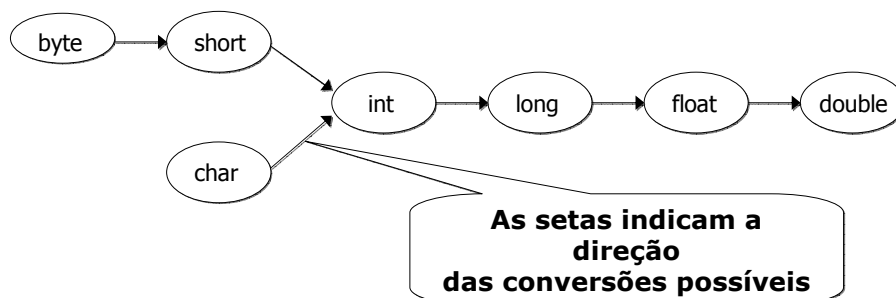
```
int i = 100;  
long l = l;    OK
```

Caso a conversão seja realizada para um tipo de menor capacidade, por exemplo de int para short, é necessário que a mesma seja feita explicitamente, utilizando um operador de *cast* (type), o qual irá "forçar" e informar ao compilador que a conversão deve ser realizada. Conversões dessa natureza são ditas *Narrowing Conversion* e podem causar perda de informação:

```
int i = 100;  
short s = i;      ERRO COMPILAÇÃO!!  
short s = (short)i; OK!!
```

~~Atribuição e passagem de parâmetros~~

- ▶ É sempre possível quando a conversão ocorre de um tipo "menor" para um tipo "maior" (*widening conversion*)



Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 66



A figura ilustra os tipos de conversões que podem ser realizadas entre tipos primitivos de forma implícita, sem a necessidade de *casting*. Desta forma:

byte → (*byte*, *short*, *int*, *long*, *float* e *double*)

short → (*short*, *int*, *long*, *float* e *double*)

char → (*char*, *int*, *long*, *float* e *double*)

int → (*int*, *long*, *float* e *double*)

long → (*long*, *float* e *double*)

float → (*float* e *double*)

double → (*double*)

Qualquer outra conversão entre tipos primitivos que não esteja listada acima, deve ser realizada explicitamente utilizando o operador de *cast* para o tipo desejado.

~~Promoção aritmética~~

- ▶ Acontece quando valores de tipos diferentes são operandos de uma expressão aritmética
- ▶ Operadores binários:
 - O tipo do operando de menor tamanho (bits) é promovido para o tipo do operando de maior tamanho
 - Os tipos *short*, *char* e *byte* são sempre convertidos para o tipo *int* em operações envolvendo apenas esses tipos

Copyright © 2002 QualiTi. Todos os direitos reservados.

Outro mecanismo para conversão entre tipos primitivos é a promoção aritmética. Nesta, antes de avaliar a expressão, o compilador realiza a conversão entre tipos da seguinte forma: promovendo o operando de menor tamanho para o operando de maior tamanho. Somente depois desse processo, o compilador realiza a avaliação das expressões.

~~Promoção aritmética~~

- ▶ Operadores unários:
 - Os tipos *short*, *char* e *byte* são sempre convertidos para o tipo *int* (exceto quando usados ++ e --)
 - Demais tipos são convertidos de acordo com o maior tipo sendo utilizado na mesma expressão

```
short s = 9;  
int i = 10;  
float f = 11.1f;  
double d = 12.2d;  
if (-s * i >= f / d) {  
    ...  
}
```

Qualiti Software Processes
Java Básico | 68



Copyright © 2002 Qualiti. Todos os direitos reservados.

No exemplo é apresentada uma comparação entre duas expressões cujos operandos possuem tipos primitivos diferentes. O compilador antes de realizar a compilação, avalia os operandos do lado esquerdo e direito de modo a realizar as devidas promoções aritméticas para permitir que as expressões possam ser comparadas.

No operando do lado esquerdo, o valor da variável *s* (do tipo *short*) é promovida para um valor do tipo inteiro, uma vez que a variável *i* é declarada com este tipo. Neste caso o operando do lado esquerdo representa uma expressão do tipo *int*.

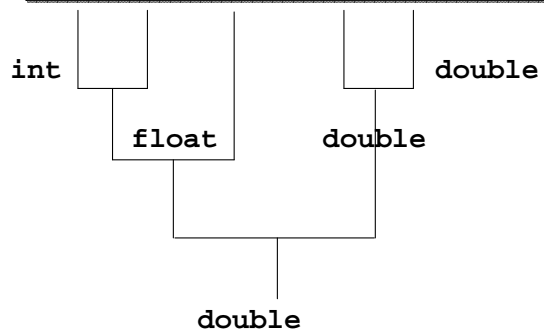
No operando do lado direito, a variável *f* (do tipo *float*) é promovida para o tipo *double*, representada pela variável *d*. Neste caso, o operando do lado direito representa uma expressão do tipo *double*.

Para finalmente realizar esta comparação, o operando do tipo *int* deve ser convertido em um valor do tipo do operando *double*. Assim o compilador vai realizar a comparação entre duas expressões do tipo *double*.

~~Exemplo de promoção aritmética~~

```
byte b = 1;  
int i = 1;  
float f = 5.2f;  
double d = 7.5;
```

```
((b + i) * f) / (d + i)
```



Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 69



~~O operador “cast”~~

► Usado para conversões

► Sintaxe

(*<tipo>*) *<expressão>*

```
int a = 1234;  
long b = a;  
short d = 10;
```

**conversão
implícita**

```
int c = (int) b;  
short c = (short) a;
```

**conversão
explícita**

Copyright © 2002 Qualiti. Todos os direitos reservados.

O operador de *casting* é necessário para converter o valor do tipo da expressão de tamanho maior, em uma valor correspondente ao tamanho da variável ao qual ele está sendo atribuído.

~~Cast entre tipos primitivos~~

- Casts podem ser realizados entre quaisquer tipos primitivos, exceto boolean

```
double d = 10.0d;  
int i = (int) d;
```

**Casts envolvendo o tipo boolean
não são permitidos!**

O único tipo primitivo que não pode participar de quaisquer conversões é o tipo booleano (boolean). Esta é uma regra da linguagem Java que deve ser seguida, ou seja, somente tipos primitivos, exceto tipo boolean, e tipos referência podem participar de conversões entre tipos.

Estruturas de Controle

Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes



Estruturas de Controle

- ▶ ~~if~~
- ▶ ~~if-else~~
- ▶ ~~if-else-if~~
- ▶ ~~switch-case~~
- ▶ while
- ▶ do-while
- ▶ ~~for~~

Copyright © 2002 Qualiti. Todos os direitos reservados.

A linguagem Java define estruturas de controle para especificar a ordem ou condição a partir da qual as instruções de um programa devem ser processadas.

As estruturas de controle Java podem ser categorizadas em três tipos: estruturas de sequência, estruturas de seleção e estruturas de repetição.

- Estrutura de sequência: é um mecanismo inerente à linguagem Java pois, à menos que especificado de forma diferente, em Java, as instruções de um programa são executadas na ordem em que aparecem.

- Estrutura de seleção: é um mecanismo onde uma determinada ação é realizada dependendo se uma dada condição for satisfeita ou não. Java oferece três estruturas de seleção **if**, **if-else** (**if-else-if**) e **switch-case**.

- Estrutura de repetição: uma instrução ou conjunto de instruções é processada iterativamente, dependendo se uma condição é satisfeita ou não. Java fornece três estruturas de repetição: **while**, **do-while** e **for**.

Um programa Java pode conter muitas diferentes combinações das estruturas de controle, com o objetivo de realizar uma dada tarefa.



► Declaração condicional mais simples em Java

```
if ( expressão booleana ){  
    comando  
}
```

```
if ( nomeUsuario == null ){  
    nomeUsuario = "indefinido";  
}
```

```
if ( vendas >= meta ){  
    desempenho = "Satisfatório";  
    bonus = 100;  
}
```

Copyright © 2002 Qualiti. Todos os direitos reservados.



A palavra chave `if` identifica a estrutura de seleção mais simples utilizada em Java. Esta estrutura é acompanhada de uma expressão booleana entre parênteses e pode ou não conter um bloco com chaves (`{ }`), delimitando os comandos que devem ser executados.

Se existe uma única linha de comando a ser executada, esta estrutura não precisa conter um bloco delimitador. Entretanto, é uma boa prática utilizar os comandos entre chaves para evitar problemas futuros, por exemplo, novos comandos podem ser inseridos e as chaves delimitando os blocos podem ser esquecidas. Neste caso, haverá uma interpretação errada do código a ser executado.

Com a estrutura `if`, caso a mesma não seja delimitada por chaves (`{ }`), a primeira linha de comando declarada após o `if` será executada. O restante de linhas não serão reconhecidas como fazendo parte da estrutura. Este comportamento é o mesmo para a maioria das estruturas de controle Java.

As linhas de comando dentro do bloco só serão executadas se a condição booleana declarada for satisfeita. Por exemplo:

```
if (nome == null){  
    nome = "indefinido";  
}  
...
```

No exemplo acima, caso a condição `nome==null` seja `false`, o controle do programa será repassado para a próxima linha imediatamente após a chave que fecha o bloco e o mesmo não será executado.

~~if-else~~

```
if (expressão booleana){  
    comando1  
}else{  
    comando2  
}
```

```
if (vendas >= meta){  
    desempenho = "Satisfatório";  
    bonus = 100+ 0.01*(vendas-meta);  
}else {  
    desempenho = "Não Satisfatório";  
    bonus = 0;  
}
```

```
if ( media < 5 ){  
    resultado = "Reprovado";  
}else {  
    resultado = "Aprovado";  
}
```

Qualiti Software Processes
Java Básico | 75



Copyright © 2002 Qualiti. Todos os direitos reservados.

A palavra chave **if-else** define a estrutura de seleção mais comum utilizada em Java.

Com esta estrutura, uma ação é realizada se a condição booleana for satisfeita. Caso contrário, uma outra ação deve ser realizada, como nos exemplos citados acima.

Nos dois casos, quando o processamento a ser realizado for formado por mais de uma linha de comando, o mesmo deve ser delimitado por chaves representando um bloco.

Do mesmo jeito que acontece com a estrutura de seleção **if**, para a estrutura **if-else**, quando o processamento contém uma única linha de comando é aconselhável utilizá-lo dentro de um bloco. Esta abordagem, além de evitar a inserção de erros e ambigüidades facilita a extensibilidade.

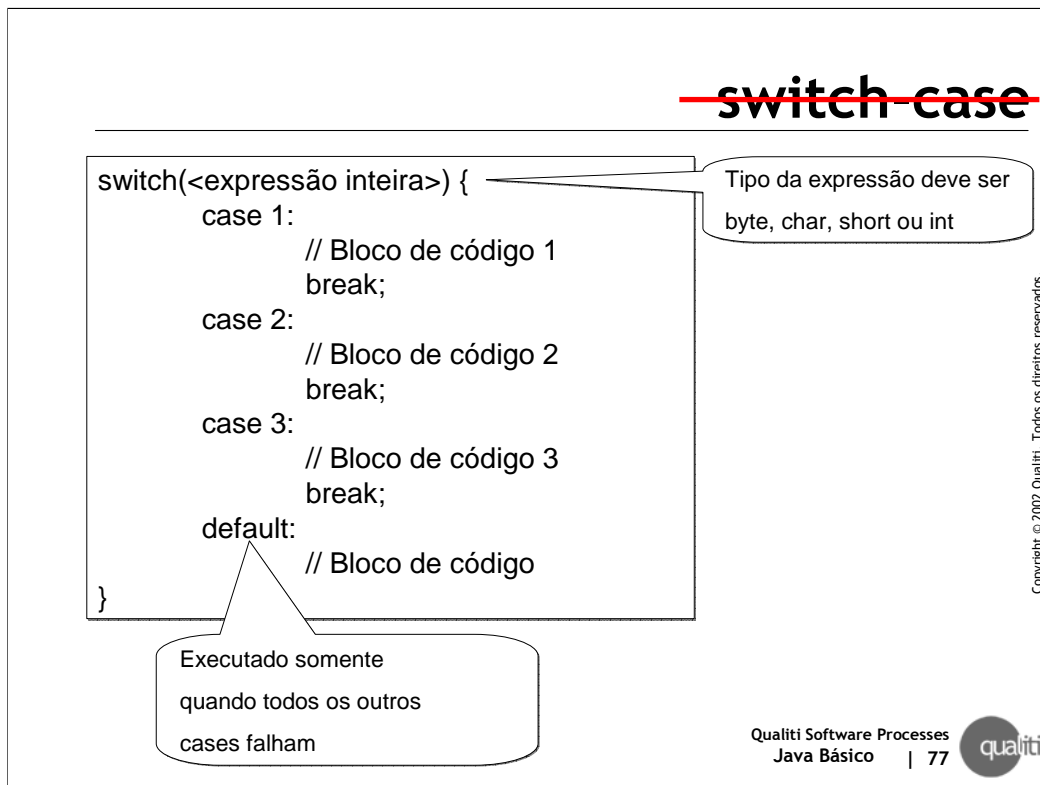
~~if-else-if~~

```
if (expressão booleana){  
    bloco de comandos  
}else if (expressão booleana){  
    bloco de comandos  
}else {  
    bloco de comandos  
}
```

```
if ( vendas >= 2*meta ){  
    desempenho = "Excelente";  
    bonus = 1000;  
} else if ( vendas >= 1.5*meta ){  
    desempenho = "Boa";  
    bonus = 500;  
} else if ( vendas >= meta ){  
    desempenho = "Satisfatório";  
    bonus = 100;  
} else {  
    desempenho = "Regular";  
    bonus = 0 ;  
}
```

Copyright © 2002 Qualiti. Todos os direitos reservados.

Uma variação de **if-else** é a estrutura **if-else-if**. Esta palavra chave define estruturas **if-else** aninhadas. Neste tipo de estrutura muitas condições podem ser testadas e diferentes ações podem ser tomadas, colocando estruturas **if-else** dentro de outras estruturas **if-else**.



A palavra chave **switch-case** define a estrutura de seleção múltipla em Java.

Nesta estrutura, em vez de uma condição booleana, a condição é representada como uma variável ou expressão constante e integral. Ou seja, a mesma só pode ser representada por valores do tipo *char*, *byte*, *short*, *int*.

A estrutura *switch* é composta por uma série de estruturas *case*, as quais representam os possíveis valores que a expressão inteira pode assumir. Cada estrutura *case*, por sua vez, é formada por blocos com comandos para processamento. Assim, dependendo do valor da expressão fornecida, um dos blocos *case* será executado.

Além disso, a estrutura **switch-case** também pode fornecer um bloco que deve ser processado caso a expressão fornecida não corresponda à nenhuma das opções definidas no *case*. O bloco em questão deve ser definido com o nome *default*.

A estrutura **switch-case** deve ainda utilizar uma instrução *break*, ao término de cada bloco *case* ou no ponto onde seja desejável que o processamento termine. Esta instrução faz com que o fluxo de controle prossiga imediatamente com a primeira instrução após a estrutura **switch-case**.

Se a instrução *break* não for definida em uma estrutura **switch-case**, quando a expressão fornecida pelo usuário “casar” com um dos *cases* disponíveis, todas as instruções a partir do *case* selecionado serão executadas na sequência em que aparecem, uma vez que não existe nenhuma condição de parada.

~~switch case~~

```
boolean analisarResposta(char resposta) {  
    switch(<resposta>) {  
        case 's':  
        case 'S': return true;  
        case 'n':  
        case 'N': return false;  
        default:  
            System.out.println("Resposta inválida!");  
            return false;  
    }  
}
```

return também pode ser
usado para sair do case

Copyright © 2002 Qualiti. Todos os direitos reservados.

Um outro mecanismo que pode ser utilizado para indicar término de execução pode ser a instrução **return**. Quando declarada, esta instrução termina o processamento do bloco *case* retornando o valor desejado. Entretanto, sai do método em que o switch está sendo executado.

Por exemplo, no código acima, a estrutura de seleção é definida dentro de um método que possui um tipo de retorno, o qual deve ser do tipo booleano. Neste caso, quando um **return** é utilizado desta forma, além de terminar a execução das instruções de processamento, o valor retornando pela instrução **return** é passada para o invocador do método `analisarResposta()`.

while

```
while ( expressão booleana ){  
    bloco de comandos  
}
```

Teste é feito no início

```
int contador = 0;  
while ( contador < 10 ) {  
    System.out.println(contador);  
    contador++;  
}
```

```
x = 10;  
while ( x < 10 )  
    x = x + 1;
```

pode ser executado
0 vezes!

```
while ( true )  
    System.out.println("Casa Forte");
```

loop infinito

Qualiti Software Processes
Java Básico | 79



Copyright © 2002 Qualiti. Todos os direitos reservados.

A palavra chave **while** define uma das estruturas de repetição Java. Nesta estrutura, uma condição booleana é testada antes de entrar no bloco **while**, o qual deve conter instruções para processamento.

Enquanto a condição booleana for satisfeita, os comandos dentro do bloco **while** serão processados iterativamente. No entanto, é importante observar que existe a possibilidade de que os comandos declarados dentro de um bloco **while** nunca sejam executados, caso a condição booleana não seja satisfeita logo na primeira iteração.

Para evitar ambigüidades, é aconselhável que chaves delimitadoras de bloco sejam utilizadas em conjunto com a estrutura **while**.

do-while

```
do{  
    bloco de comandos  
}while( expressão booleana );
```

Teste é feito no final

```
int contador = 0;  
do {  
    System.out.println(contador);  
    contador++;  
}while ( contador < 10 );
```

comandos são executados pelo menos uma vez

```
String resposta;  
do {  
    resposta = "Resposta Incorreta!";  
}while( ehInvalida(resposta) );
```

Copyright © 2002 QualiTi. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 80



A estrutura de repetição **do-while** é semelhante à estrutura **while**. A diferença principal é que na estrutura **while**, os comandos declarados no mesmo podem não ser processados, caso a condição booleana não seja satisfeita. Com a estrutura **do-while**, o bloco de comandos é processado PELO MENOS uma vez, já que a condição booleana é testada somente ao final.

~~for~~

```
for ( inicialização; condição; incremento ){  
    bloco de comandos  
}
```

```
for ( int contador = 0; contador < 10; contador++ ){  
    System.out.println(contador);  
}
```

```
void tabuada() {  
    int x,y;  
    for ( x=1, y=1; x<=10; x++, y++ ) {  
        System.out.print(x + " X " + y + " = ");  
        System.out.println(x*y);  
    }  
}
```

Qualiti Software Processes
Java Básico | 81



Copyright © 2002 Qualiti. Todos os direitos reservados.

A palavra chave **for** define uma estrutura de repetição cujas instruções são executadas em um *loop*, dependendo de alguns fatores: uma condição booleana, a qual deve ser satisfeita para que as instruções declaradas dentro do *loop* possam ser processadas; uma ou mais variáveis de controle para inicializar o *loop*; e uma ou mais expressões representando o incremento ou decremento das variáveis de controle, as quais são modificadas a cada passagem de laço, determinando o número de iterações realizadas no mesmo.

break

- Usado para terminar a execução de um bloco **for**, **while**, **do** ou **switch**

```
int procurar(String nome) {  
    int indice = -1;  
    for (int i = 0; i < 50; i++) {  
        if (i < MAXIMO ) {  
            indice = i;  
            break;    //Interrompe o loop  
        }  
    }  
    return indice;  
}
```

Qualiti Software Processes
Java Básico | 82



Copyright © 2002 Qualiti. Todos os direitos reservados.

A instrução **break** é utilizada em Java para alterar o fluxo de controle do processamento das instruções. Esta instrução pode ser definida dentro de estruturas **for**, **while**, **do** ou **switch** para terminar a execução de instruções de processamento contidas nestas estruturas.

Com o uso do **break** nas estruturas de controle, o fluxo do processamento é passado para a primeira instrução imediatamente após o término do bloco dessas estruturas.

continue

- Termina a execução da iteração atual do loop e volta ao começo do loop.

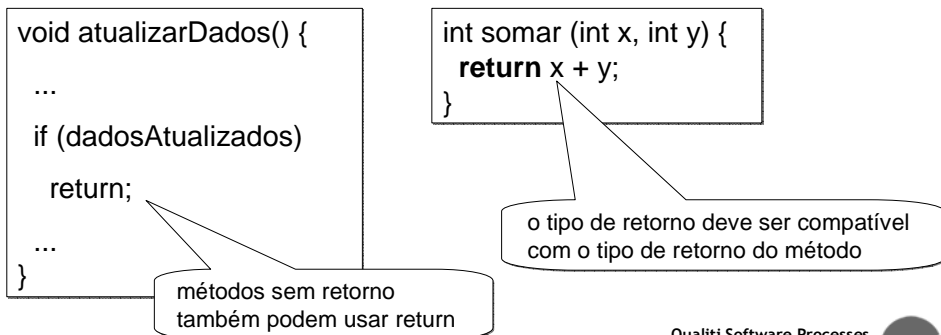
```
for (int k = 0; k < 10; k++) {  
    if (k == 5) {  
        continue; //Envia à reavaliação da condição  
    }  
    System.out.println("k = " + k);  
}
```

Copyright © 2002 Qualiiti. Todos os direitos reservados.

A instrução **continue** também pode ser utilizada em estruturas de repetição (**do**, **while** e **for**) para alterar o fluxo de controle da execução. Neste caso, quando esta instrução é processada, o fluxo de controle é repassado para a próxima iteração no *loop*.

return

- ▶ Termina a execução de um método e retorna a chamada ao invocador
- ▶ É obrigatório quando o tipo de retorno do método não é `void`
- ▶ O resultado da avaliação de *expressão* deve poder ser atribuído ao tipo de retorno do método



Qualiti Software Processes
Java Básico | 84



Copyright © 2002 Qualiti. Todos os direitos reservados.

A instrução **return** se comporta como a instrução **break**, se utilizada dentro de estruturas de controle. Ou seja, pode ser utilizada para terminar a execução de instruções dentro de uma estrutura de controle qualquer, por exemplo **if**, **while**, etc, passando o controle de execução para a linha imediatamente após o bloco correspondente à estrutura de controle.

Outro uso desta instrução, desta vez **OBRIGATÓRIO**: deve ser declarada ao final de métodos que retornam algum valor. Ao final do processamento, o valor resultante deve ser especificado do lado direito da expressão **return**, como no exemplo a seguir:

```
int subtrair (int x, int y) {  
    int resultado = x - y;  
    return resultado;  
}
```

No exemplo acima, o método de nome **subtrair** tem como tipo de retorno um valor do tipo **int**.

Maiores detalhes sobre a definição de métodos serão vistos no módulo **Classes e Objetos: Atributos e Métodos**.

Referências do módulo

- ▶ Java Tutorial
 - <http://java.sun.com/docs/books/tutorial/>
- ▶ Documentação da Versão 6 da linguagem
 - <http://java.sun.com/javase/6/docs/>

Copyright © 2002 Qualiti. Todos os direitos reservados.



Módulo 4

Classes e Objetos: atributos e métodos

Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes



Programação Orientada a Objetos

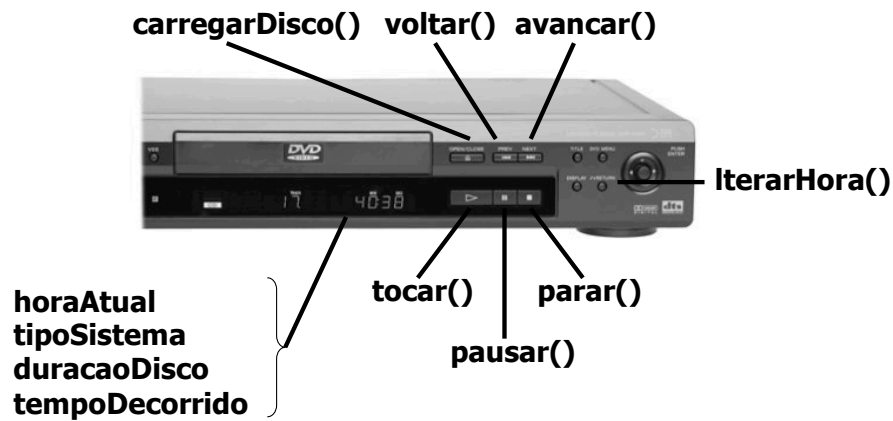
- ▷ Olhar o mundo como se tudo pudesse ser representado por objetos
- ▷ Estruturação do programa é baseada na representação de objetos do mundo real (estados + comportamento)
- ▷ Vantagens
 - Facilidade de manutenção
 - Maior extensibilidade
 - Maior reuso

Copyright © 2002 QualiTi. Todos os direitos reservados.

Com o paradigma de Programação orientada a objetos, um programa é formado por objetos. Estes objetos, em geral, representam objetos do mundo real e possuem certas propriedades que podem ser utilizadas e operações que podem ser realizadas sobre os mesmos. As propriedades ditam os estados que os objetos podem assumir em um dado momento e as operações especificam o comportamento que estes objetos podem executar, ou melhor, as operações que podem ser realizadas sobre os mesmos.

Com programação orientada a objetos, cada objeto deve descrever somente propriedades e comportamento inerentes à sua natureza. Os objetos podem ter relacionamentos e a comunicação feita entre eles deve ser realizada através de mensagens. Na prática, estas mensagens funcionam como chamadas de métodos entre objetos.

Objeto DVD



Copyright © 2002. Quali. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 88



Exemplo de um objeto do mundo real que pode ser modelado como um objeto no paradigma de orientação a objetos. O objeto do exemplo, DVD, possui propriedades (`tempoDecorrido`, `horaAtual`, `duracaoDisco`, `tipoSistema`) e operações (`voltar`, `pausar`, `avancar`, etc) que podem ser realizadas sobre o mesmo.

Classes e Objetos

- ▶ Classes especificam a estrutura e o comportamento dos objetos
- ▶ Classes são como "moldes" para a criação de objetos
- ▶ Objetos são **instâncias** de classes.

Copyright © 2002 QualiTi. Todos os direitos reservados.



Classes são utilizadas para especificar a estrutura do objeto a ser criado. Ela serve como um molde para a criação deste, fornecendo exatamente quais as propriedades que um objeto desta classe deve conter, bem como o comportamento desejado ao mesmo.

Isto implica que um **objeto** não existe sem uma classe, uma vez que o mesmo só pode ser criado a partir desta. Ou seja, um objeto é uma instância de uma classe.

Objetos

- Um objeto representa uma entidade do mundo real
- Todo objeto tem

Identidade

Estado

Comportamento

Copyright © 2002 Qualiti. Todos os direitos reservados.

Objetos

- ▷ **Identidade**
 - Todo objeto é único e pode ser distinguido de outros objetos
- ▷ **Estado**
 - Todo objeto tem estado, que é determinado pelos dados contidos no objeto
- ▷ **Comportamento**
 - O comportamento de um objeto é definido pelos serviços/operações que ele oferece

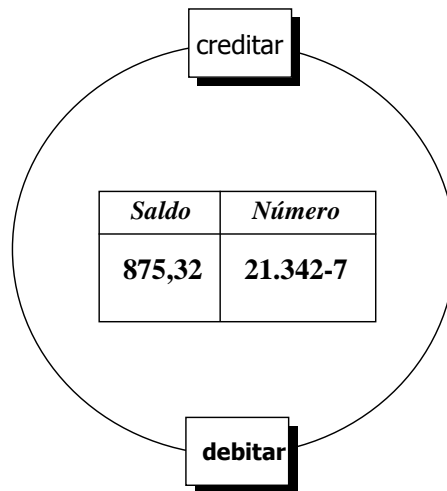
Copyright © 2002 QualiTi. Todos os direitos reservados.

Todo objeto deve possuir uma **identidade**, um **estado** e **comportamento**.

Do mesmo jeito que ocorre com objetos do mundo real, objetos em orientação a objetos também possuem identidade única; por mais que o estado de um objeto possa ser, em um dado momento, igual ao estado de um outro objeto da mesma classe, a identidade é única para cada um.

O estado de um objeto é representado pelos atributos definidos na classe desse objeto. E o comportamento especificado a um objeto é definido como método (s) na classe do objeto correspondente.

Objeto Conta Bancária



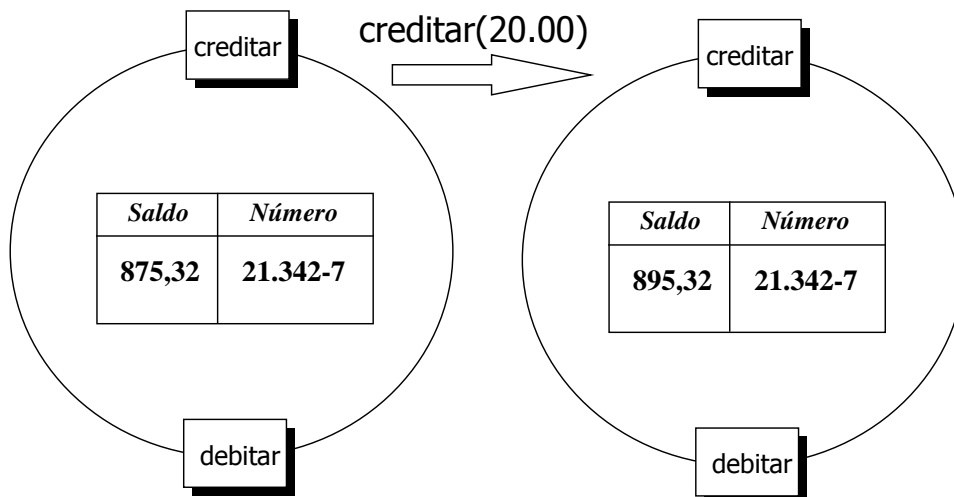
Copyright © 2002 QualiTi. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 92



Representação de um objeto Conta Bancária, com estado: saldo=875,32 e número 21.342-7; e comportamento: operações creditar e debitar.

Estados do Objeto Conta



Copyright © 2002 QualiTi. Todos os direitos reservados.

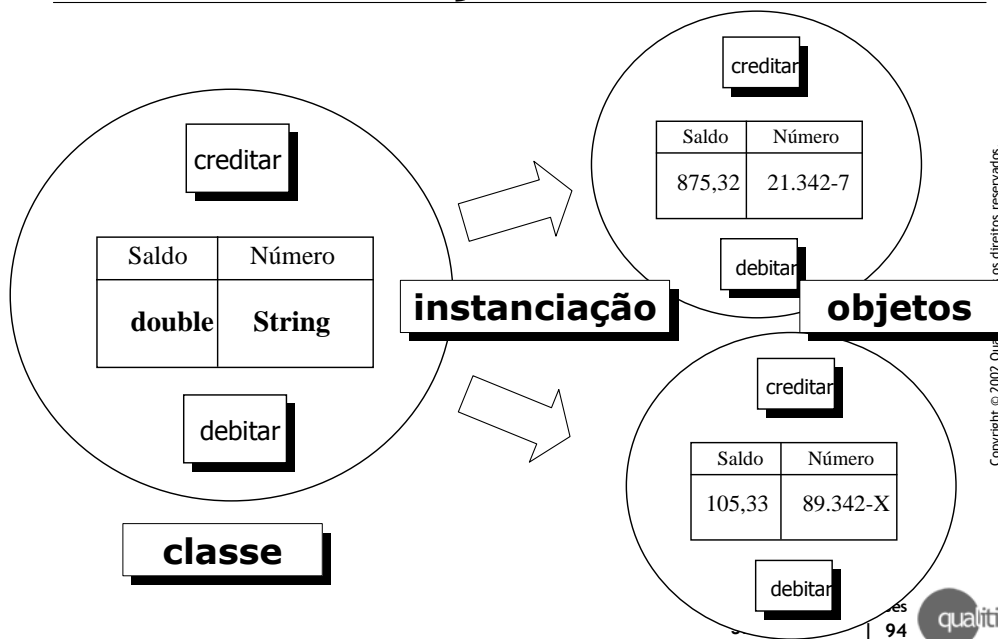
Qualiti Software Processes
Java Básico | 93



O mesmo objeto Conta Bancária assumindo mudando de estado, após a execução de uma operação `creditar()` sobre o mesmo. O saldo era 875,32 e após a execução do método `creditar` sobre os dados do objeto Conta, o saldo passou a ser 895,32.

Em geral, operações de um objeto podem manipular os dados do próprio objeto, o que pode ocasionar a alteração do estado do mesmo.

Classe e objeto Conta Bancária



Representação de uma classe (estrutura e comportamento) e suas instâncias (objetos).

Definição de Classes em Java

```
class NomeDaClasse {  
    CorpoDaClasse  
}
```

O corpo de uma classe pode conter

- atributos
- métodos
- construtores
- outras classes...

Estrutura básica de uma classe Java

```
class NomeDaClasse {  
    atributo1;  
    atributo2;  
    ...  
    método1 {  
        Corpo do método1  
    }  
    método2 {  
        Corpo do método2  
    }  
}
```

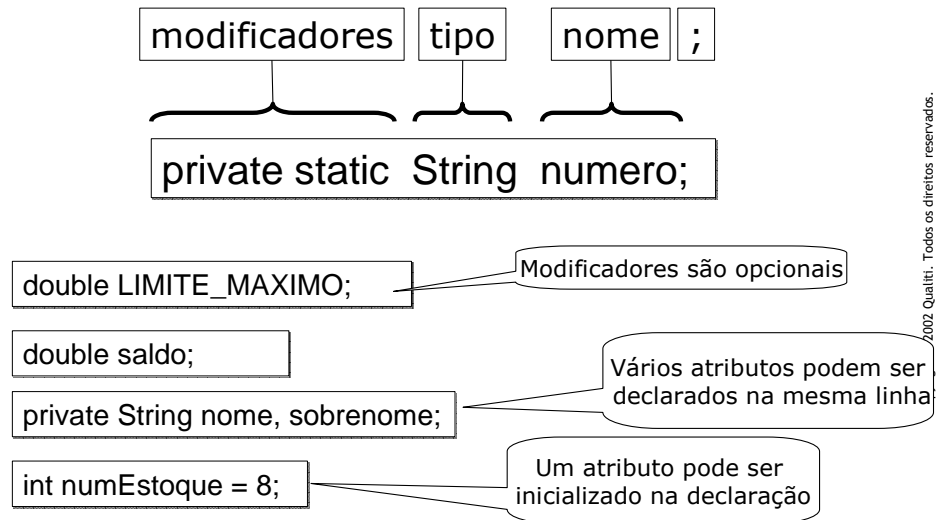
```
class Conta {  
    CorpoDaClasse  
}
```

Copyright © 2002 Qualiti. Todos os direitos reservados.

Atributos

Copyright © 2002 Qualiti. Todos os direitos reservados.

Declaração de atributos



Atributos só podem ser declarados no corpo de uma classe Java e são visíveis por qualquer outro membro desta classe. Eles representam o estado do objeto.

Exemplos de atributos

```
class Livro {  
    int anoDePublicacao;  
    int numeroDePaginas;  
    String titulo;  
    ...  
}
```

```
class Conta {  
    String numero;  
    double saldo;  
    ...  
}
```

```
class Cabine {  
    int nivel;  
    String codigo;  
    int codCategoria;  
    int lotacaoMaxima;  
    ...  
}
```

Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 99



Exemplos de diferentes classes contendo diferentes atributos.

Métodos

Copyright © 2002 Qualiti. Todos os direitos reservados.

O que são métodos?

Métodos são operações que realizam ações e modificam os valores dos atributos do objeto responsável pela sua execução

Copyright © 2002 QualiTi. Todos os direitos reservados.

Para imprimir comportamento em um dado objeto é necessário a definição de operações que representem tal comportamento. Essas operações em uma classe Java são chamadas **métodos**.

Uma classe Java pode conter tantos métodos quanto necessário para expressar o comportamento relativo ao mesmo. É importante tomar cuidado para evitar inserir comportamento não relacionado ao objeto em questão.

Declaração de métodos

modificadores tipo de retorno nome (parâmetros) {...}

```
private double obterRendimento(String numConta, int mes){  
    //corpo do método  
}
```

```
int soma(int a, int b) {  
    return a + b ;  
}
```

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

Copyright © 2002 Qualiti. Todos os direitos reservados.

O corpo do método

- ▶ O corpo do método contém os comandos que determinam as ações do método
- ▶ Esses comandos podem
 - realizar simples atualizações dos atributos de um objeto
 - retornar valores
 - executar ações mais complexas como chamar métodos de outros objetos
- ▶ O corpo do método também pode conter declarações de variáveis
 - Variáveis cuja existência e valores são válidos somente dentro do método em que são declaradas.

Copyright © 2002 Qualiiti. Todos os direitos reservados.

Um método deve conter comandos para a execução da operação fornecida pelo mesmo e pode também conter variáveis que auxiliem na execução dessa operação.

É comum haver confusão entre dois conceitos diferentes: atributos e variáveis: ao contrário de atributos, as variáveis não representam o estado de um objeto e devem ser declaradas somente em métodos. Variáveis não possuem significado e nem são visíveis fora do método onde são declaradas.

O estado das variáveis só existe durante a execução do método no qual a mesma é declarada. Por Exemplo:

```
public void imprimirNome(){  
    String nome = "Gabriel Souza";  
    System.out.println("Nome do cliente =" + nome);  
}
```

Supondo que o método `imprimirNome()` esteja declarado em uma classe qualquer, a variável `nome` é visível somente dentro deste método e o seu valor ("**Gabriel**") só existe durante a execução deste método.

Exemplo de Método

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

Copyright © 2002 Qualiti. Todos os direitos reservados.

Métodos e tipo de retorno

```
class Conta {  
    String numero;  
    double saldo;  
  
    String getNumero() {  
        return numero;  
    }  
  
    double getSaldo() {  
        return saldo;  
    }  
    ...  
}
```

Os métodos que retornam valores como resultado usam o comando **return**

Copyright © 2002 Qualiti. Todos os direitos reservados.

Mais sobre métodos

Usa-se **void** para indicar que o método não retorna nenhum valor. Nesse caso apenas altera o valor do atributo saldo do objeto.

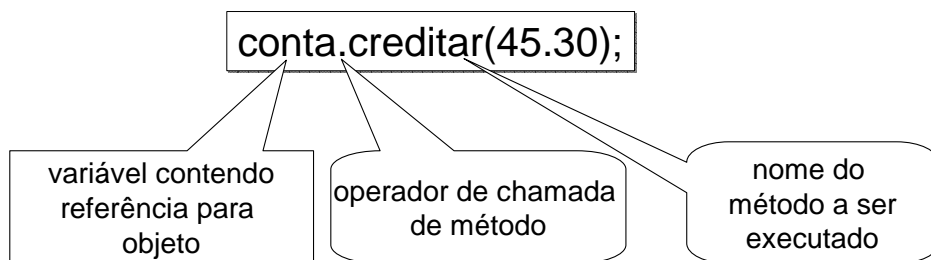
```
class Conta {  
    ...  
    void debitar(double valor) {  
        saldo = saldo - valor;  
    }  
}
```

Por que o debitar não tem como parâmetro o número da conta?

Copyright © 2002 Qualiti. Todos os direitos reservados.

Chamada de métodos

- ▶ Métodos são invocados em instâncias (objeto) de alguma classe.
 - Podem também ser invocados a partir da própria classe (métodos estáticos).
- ▶ Os objetos se comunicam para realizar tarefas
- ▶ Parâmetros são passados por “cópia”
- ▶ A comunicação é feita através da **chamada de métodos**



Modificadores

Copyright © 2002 Qualiti. Todos os direitos reservados.

Modificadores

► Modificadores de acesso

- public
- protected
- private
- default (friendly)

Valor padrão quando nenhum modificador de acesso é especificado

► Outros modificadores

- static
- final
- native
- transient
- synchronized

Copyright © 2002 QualiTi. Todos os direitos reservados.

Modificadores são palavras-chave utilizadas em Java, que fornecem ao compilador informações sobre a natureza do código, atributo ou classe.

Estes modificadores podem ser classificados em duas categorias: modificadores de acesso e outros modificadores. Os primeiros especificam quais classes têm acesso aos membros (classe, atributos, métodos e construtores) de uma determinada classe.

Os outros modificadores podem ser utilizados em conjunto com os modificadores de acesso para descrever os atributos de uma determinada classe.

Modificadores de Acesso

- ▶ Controlam o acesso aos membros de uma classe
- ▶ Membros de uma classe:
 - A própria classe
 - Atributos
 - Métodos e construtores (um tipo especial de métodos)
- ▶ Não são aplicados à variáveis

Copyright © 2002 QualiTi. Todos os direitos reservados.

Não é necessário o uso de modificadores de acesso em variáveis, pelo fato das mesmas serem visíveis SOMENTE dentro dos métodos nos quais são declaradas. Por este motivo é mandatório que variáveis não sejam declaradas com modificadores de acesso. Do contrário, uma exceção é gerada pelo compilador.

public

- ▶ Uma classe **public** pode ser instanciada por qualquer classe
- ▶ Atributos **public** podem ser acessados (lidos, alterados) por objetos de qualquer classe
- ▶ Métodos **public** podem ser chamados por métodos de qualquer classe

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

Copyright © 2002 QualiTi. Todos os direitos reservados.

protected

- ▷ Usado somente para atributos e métodos
- ▷ Atributos **protected** podem ser acessados (lidos, alterados) por objetos de classes dentro do mesmo pacote ou de qualquer subclasse da classe ao qual ele pertence
- ▷ Métodos **protected** podem ser chamados por objetos de classes dentro do mesmo pacote ou de qualquer subclasse da classe ao qual ele pertence

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

Qualiti Software Processes
Java Básico | 112



Copyright © 2002 Qualiti. Todos os direitos reservados.

Pacotes e subclasses são conceitos relacionados aos módulos posteriores. Por este motivo, não são detalhados neste momento. Por enquanto, é suficiente saber que pacote pode ser considerado como o agrupamento de classes em uma mesma localização e subclasse, como o nome sugere, herda características de uma outra classe.

default (friendly)

- ▶ A classe é visível somente por classes do mesmo pacote
- ▶ Se um atributo não tem nenhum modificador de acesso associado, ele é “implicitamente” definido como **friendly**, e só é visível para objetos de classes do mesmo pacote
- ▶ Se um método não tem nenhum modificador de acesso associado, ele é “implicitamente” definido como **friendly**, e só pode ser chamado a partir de objetos de classes do mesmo pacote

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

private e encapsulamento

atributos **private** podem ser acessados somente por **objetos da mesma classe**

```
class Cliente {  
    private String cpf;  
    private String nome;  
    ...  
}
```

- ▶ Java não obriga o uso de **private**, mas vários autores consideram isso essencial para a programação orientada a objetos
- ▶ Impacto em coesão e acoplamento
- ▶ Use **private** para atributos!

Encapsulamento (*data hiding* ou *information hiding*) nada mais do que combinar dados e comportamento em um mesmo “pacote” de modo a esconder os dados, do usuário que está utilizando o objeto.

Encapsulamento tem impacto em desacoplamento e coesão, uma vez que impede o acesso direto aos atributos de uma classe. Acesso direto aos atributos de uma classe, dificulta a manutenção.

private

- Métodos **private** só podem ser chamados por métodos da classe onde são declarados

```
class Cliente {  
    private String cpf;  
    ...  
    int getCpf(){  
        return formatarCpf();  
    }  
  
    private int formatarCpf(){  
        //código para adicionar os pontos e p hífen de Cpf  
    }  
    ...  
}
```

Qualiti Software Processes
Java Básico | 115



Copyright © 2002 Qualiti. Todos os direitos reservados.

Em geral métodos com visibilidade **private** devem ser definidos quando é necessário o uso de operações auxiliares aos métodos que, de fato, executam a funcionalidade correspondente ao objeto. Por exemplo, supondo que exista um método que dever retornar o ano de nascimento para um usuário qualquer, por exemplo `getAnoDeNascimento()`. Supondo ainda que esta informação deva ser manipulada antes de ser enviada como resposta, por exemplo, o ano deve ser convertido para um formato específico antes de ser enviado. Em vez deste processamento estar declarado no método, cuja função é retornar o ano, o mesmo pode ser definido em um método auxiliar, `formatarAno()`, por exemplo.

Este é um exemplo bem específico de uso. Um uso mais útil é quando operações auxiliares podem ser reusadas em diferentes métodos da classe. Isto evita duplicação de código. Por exemplo, uma aplicação que faz acesso ao banco de dados: imagine uma classe que possui diferentes métodos, `inserir`, `consultar`, `remover`, `atualizar`, etc., nos quais seria necessário obter e fechar conexões com o banco de dados.

Em vez do código correspondente a este processamento ser descrito em todos os métodos, resultando em código duplicado, o mesmo poderia ser definido em métodos **private** auxiliares, tais como `conectar()` e `desconectar()` e as chamadas aos mesmos serem inseridas nos locais onde são necessários nos métodos.

Outros modificadores

▷ final

- Pode ser utilizado para em qualquer membro de uma classe
- Torna o atributo **constante**
- O atributo não pode ser alterado depois de inicializado
- Métodos **final** NÃO podem ser redefinidos
- Classes **final** NÃO podem ser estendidas

Copyright © 2002 QualiTi. Todos os direitos reservados.

Além dos modificadores de acesso, existem outros modificadores que podem ser utilizados em conjunto com estes, objetivando imprimir uma característica a mais aos membros de uma classe.

Outros modificadores

► final

Atributos **final** geralmente são declarados como **static** também

```
class ConstantesBanco {  
    public static final int NUM_MAXIMO_CONTAS;  
    private static final double LIMITE_MIN_CHEQUES;  
    private static final int NUM_CHEQUES_TALAO;  
    ...  
}
```

Pelo padrão de codificação, constantes devem ter seus nomes em **maiúsculas**

Copyright © 2002 Qualiti. Todos os direitos reservados.

Outros modificadores

▸ static

- Pode ser usado somente em atributos e métodos
- Atributos static pertencem à classe e não aos objetos
- Só existe uma cópia de um atributo static de uma classe, mesmo que haja vários objetos da classe
- Atributos static são muito usados para constantes
- O acesso é feito usando o nome da classe:

```
int numCheques = numTaloes *  
    ConstantesBanco.NUM_CHEQUES_TALAO;
```

Outros modificadores

► static

- Métodos **static** pertencem a classes e não a objetos
- Podem ser usados mesmo sem criar os objetos
- Só podem acessar diretamente atributos estáticos
- O acesso é feito usando o nome da classe:

```
x = Math.random();
```

```
media = Funcoes.calcularMedia(valores);
```

Copyright © 2002 QualiTi. Todos os direitos reservados.

A classe **Math** é uma classe da API da Java, que contém métodos para manipulação de cálculos matemáticos. Estes métodos são declarados estáticos, o que possibilita seu uso, sem a necessidade da criação de uma instância da classe **Math**.

Outros modificadores

- ▶ **native**
 - Usado somente em métodos
 - Código nativo
- ▶ **transient**
 - Usado somente em atributos
 - Não é armazenado como parte persistente do objeto
- ▶ **synchronized**
 - Usado em métodos e em blocos de código
 - Acesso concorrente.
- ▶ **Volatile**
 - Usado somente em atributos
 - Acesso concorrente.

Copyright © 2002 QualiTi. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 120



native: utilizado em métodos que contenham processamento de código nativo, ou seja, código especificado em outra linguagem que não Java, por exemplo, C ou C++. Métodos especificados com esse modificador, têm declarado somente a assinatura dos mesmos seguido de um ; (ponto-e-vírgula). O modificador ***native*** indica que o corpo do método deve ser obtido a partir de uma biblioteca, por exemplo.

transient: este modificador é aplicado somente em variáveis, indicando que a mesma não deve ser armazenada como parte do estado persistente do objeto.

synchronized: é utilizado somente em métodos que participam de processamento concorrente.

Volatile: Avisa ao compilador para não fazer cache dessa variável em registradores pois ela pode ser atualizada por outras Threads.

Criação e remoção de objetos

Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes



Criação de objetos

- ▶ Objetos precisam ser criados antes de serem utilizados
- ▶ Construtores precisam ser definidos na classe
- ▶ A criação é feita com o operador **new**

```
Conta c = new Conta();
```

construtor

Construtores

- ▶ Além de métodos e atributos, uma classe pode conter **construtores**
- ▶ Construtores definem como os atributos de um objeto devem ser inicializados
- ▶ São semelhantes a métodos, mas não têm tipo de retorno
- ▶ O nome do construtor deve ser exatamente o nome da classe.
- ▶ Uma classe pode ter diversos construtores, diferenciados pelos parâmetros

modificador nome da classe (parâmetros) {...}

```
public Conta(String numero, Cliente cliente) {  
    this.numero = numero;  
    this.cliente = cliente;  
}
```

Construtor default

- ▶ Caso não seja definido um construtor, um construtor default é fornecido implicitamente
- ▶ O construtor default inicializa os atributos com seus valores default
- ▶ O construtor default não tem parâmetros:

```
public Conta() {  
    ...  
}
```

Um construtor é um tipo especial de método que deve existir em TODA classe java e é utilizado para inicializar os atributos declarados em uma classe.

Java fornece um construtor *default*, o qual não possui parâmetros. Este construtor é fornecido automaticamente pela linguagem, caso nenhum outro tenha sido fornecido explicitamente pelo programador. O construtor *default* inicializa TODOS os atributos declarados, com os valores *default* correspondentes a cada um.

Valores default para atributos

Tipo	Valor Default
byte, short, int, long	0
float	0.0f
double	0.0
char	`\u0000`
Tipos referência (Strings, arrays, objetos em geral)	null
boolean	false

Copyright © 2002 QualiTi. Todos os direitos reservados.

Todo tipo Java, seja tipo primitivo ou referência possui valores *default*. Isto significa que quando objetos são instanciados e os atributos definidos em uma classe não são inicializados explicitamente, os valores assumidos pelos atributos são os especificados neste tabela.

Nesta caso, quando o construtor *default* é invocado pela máquina virtual, os atributos são inicializados com os valores *default*.

O valor **null** indica uma **referência nula**. Indica que o atributo ou variável não faz referência a um objeto.

Outros construtores

- Podem ser criados novos construtores, com parâmetros

```
class Conta {  
    String numero;  
    double saldo;  
    public Conta(String numero, Cliente cliente){  
        this.numero = numero;  
        this.cliente = cliente;  
    }  
    ...  
}
```

Quando é definido um construtor com parâmetros, o construtor default não é mais gerado

Copyright © 2002 Quality. Todos os direitos reservados.

Java Básico | 126

Quality

O programador pode definir diferentes construtores com diferentes parâmetros em uma mesma classe, de modo a possibilitar a inicialização dos atributos de diferentes formas, dependendo dos requisitos do programa sendo construído. No entanto, uma vez que o programador tenha especificado, pelo menos, um construtor com parâmetros, o construtor *default* não é mais fornecido pela linguagem. Neste caso, se necessário, o mesmo deve também ser inserido, explicitamente, na classe.

Em geral, os construtores são declarados com o modificador de acesso **public**, o qual permite esta classe possa ser instanciada por outras classes.

O construtor pode também ser definido com o modificador de acesso **private**. Neste caso, objetos desta classe só podem ser instanciados por métodos da própria classe. Por uma simples razão: um construtor nada mais é que um método especial; como tal, se declarado *private*, só pode ser acessado por métodos da própria classe.

Mais sobre criação de objetos

```
Conta c;  
Cliente = new  
...  
c = new Conta("12345", cliente);
```

Atribui à variável
c a referência
criada para o
novo objeto

responsável por
criar um objeto do
tipo Conta em
memória

responsável por
inicializar os
atributos do
objeto criado

Copyright © 2002 Qualiiti. Todos os direitos reservados.

Remoção de objetos

- ▶ Não existe mecanismo de remoção explícita de objetos da memória em Java (como o `free()` de C++)
- ▶ O *Garbage Collector* (coletor de lixo) de Java elimina objetos da memória quando eles não são mais referenciados
- ▶ Você não pode obrigar que a coleta de lixo seja feita
- ▶ A máquina virtual Java decide a hora da coleta de lixo

Copyright © 2002 Qualiti. Todos os direitos reservados.



Implementação da classe Conta

Conta
-numero: String -saldo: double
+Conta(numero: String, saldo: double) +getNumero(): String +getSaldo(): double +creditar(valor: double) +debitar(valor: double)

```
class Conta{  
    private String numero;  
    private double saldo;  
  
    public Conta(String numero, double saldo) {  
        this.numero = numero;  
        this.saldo = saldo;  
    }  
  
    public String getNumero() {  
        return this.numero;  
    }  
  
    public double getSaldo() {  
        return this.saldo;  
    }  
  
    public void creditar(double valor) {  
        this.saldo += valor;  
    }  
  
    public void debitar(double valor) {  
        this.saldo -= valor;  
    }  
}
```

Referências do módulo

- ▶ Java Tutorial
 - <http://java.sun.com/docs/books/tutorial/>
- ▶ Documentação da Versão 6 da linguagem
 - <http://java.sun.com/javase/6/docs/>

Copyright © 2002 Qualiti. Todos os direitos reservados.



Módulo 5

Tipos referência, Strings, Enum e Arrays

Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes



Tipos referência

- ▷ Em Java há dois conjuntos de tipos
 - Tipos primitivos
 - Tipos referência
- ▷ **Tipos primitivos**
 - int, long, double, float, ...
- ▷ **Tipos referência**
 - classes, interfaces e arrays

Copyright © 2002 QualiTi. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 132



Além dos tipos primitivos (int, long, float, etc.) já estudados, Java também define outra categoria de tipos de dados denominada **tipos referência**. Tipos referência são compostos por classes, interfaces e array, ou seja, representam objetos em Java.

Ao contrário de tipos primitivos, os quais trabalham diretamente com o valor de uma variável, ou melhor, sua cópia, tipos referência não manipulam objetos diretamente, mas sim referências a objetos.

Por referências a objetos pode-se entender posições de memória onde o objeto está armazenado e não o valor real que o objeto agrega. Este mecanismo é o que permite, por exemplo, que duas variáveis diferentes possam referenciar o mesmo objeto:

```
Button b1, b2;  
b1 = new Button("OK");  
b2 = b1;
```

No exemplo, o objeto Button é criado e sua referência é atribuída à variável b1. Em seguida, a mesma referência é atribuída à variável b2, fazendo com que tanto a variável b1 quanto a variável b2 apontem para a mesma referência ao objeto. Ou seja, b1 e b2 apontam para a mesma posição de memória. Neste caso, qualquer modificação realizada sobre b1 é visível em b2, uma vez que ambas referenciam o mesmo objeto:

Tipos de referência

- ▶ Em Java não se trabalha diretamente com os objetos e sim com

referências a objetos

- ▶ Isso tem implicações na maneira em que objetos são comparados e copiados

Copyright © 2002 Quality. Todos os direitos reservados.

Quality Software Processes
Java Básico | 133



```
b1.setLabel("Cancel");  
String valor = b2.getLabel();
```

No exemplo acima, o objeto referenciado pela variável *b1* foi modificado para representar o valor "Cancel". Como *b2* também aponta para a mesma referência que *b1*, se o valor do *label* for obtido a partir de *b2* o valor retornado também será "Cancel".

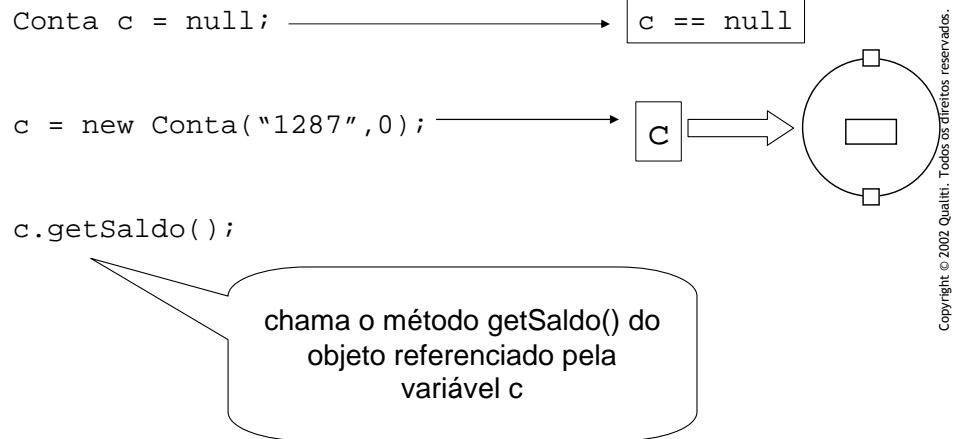
Isto só é possível porque objetos são manipulados por referência e não por valor (cópia) como o que acontece com os tipos primitivos:

```
int i =5;  
int j = i;
```

O valor da variável *i* é passado por cópia para a variável *j*. Se a variável *i* é modificada, por exemplo para 4, o valor de *j* continua sendo 5.

Referências

Objetos são manipulados através de referências



Referências

Mais de uma variável pode armazenar referências para um mesmo objeto (*aliasing*)

```
Conta a = new Conta("123-4", 340.0);  
Conta b;
```

```
b = a;
```

a e b passam a referenciar a mesma conta

```
b.creditar(100);  
System.out.println(a.getSaldo());
```

qualquer efeito via b é refletido via a

Strings

Copyright © 2002 Qualiti. Todos os direitos reservados.

Strings

- ▶ São seqüências de caracteres
- ▶ Não há um tipo primitivo para Strings em Java
- ▶ Em Java, Strings são **objetos**

```
String mensagem = "Operação concluída com sucesso";
```

Aqui Java cria um novo objeto do tipo String e o armazena na variável mensagem

Qualiti Software Processes
Java Básico | 137



Copyright © 2002 Qualiti. Todos os direitos reservados.

Uma String é definida como uma classe em Java. A mesma faz parte da API (conjunto de classes e interfaces disponibilizadas para uso) de Java e é largamente utilizada em aplicações Java quando se deseja definir um tipo que representa uma seqüência de caracteres.

Esta classe difere das classes definidas pelo usuário na forma como suas instâncias podem ser criadas:

- Usando seu construtor
`String s = new String("Este é um uso de String");`
- Usando atribuição (uso mais comum)
`String s = "Este é um uso de String";`

Concatenação de Strings

- Operador **+** é usado para concatenação

```
String nome = "George";  
String sobrenome = "Bush";  
String nomeCompleto = nome + " " + sobrenome;  
  
int anos = 10;  
double rendimento = 1270.49;  
  
String s = "Em " + anos + " anos o " +  
    "rendimento será de " + rendimento;  
  
System.out.println(s);
```

A conversão para
String é feita
automaticamente

Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 138



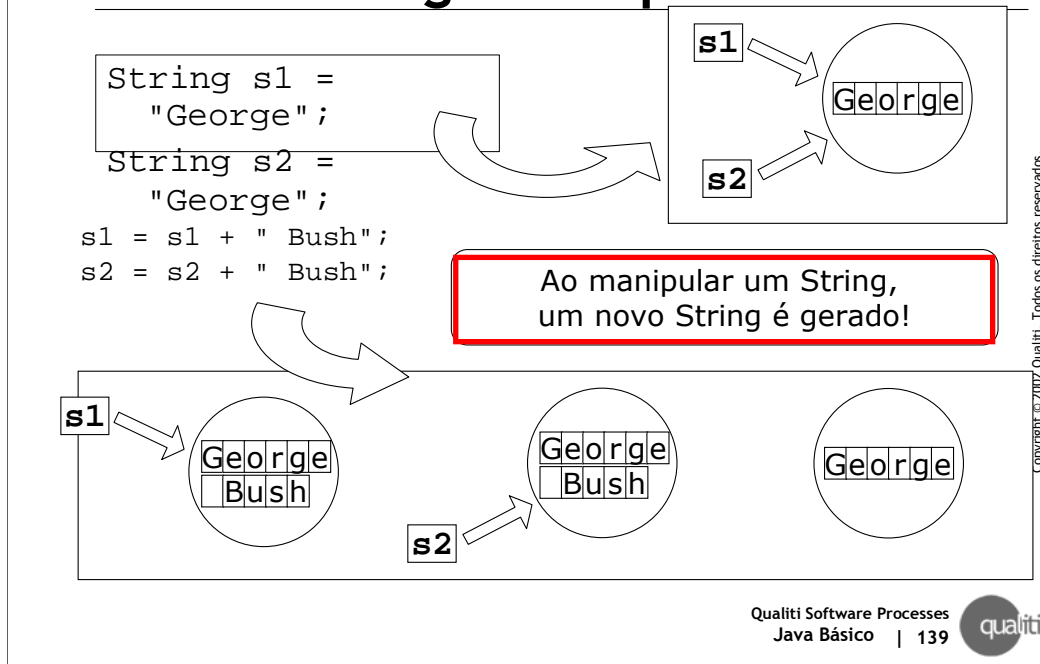
Strings podem utilizar o operador **+** para realizar concatenação. Na verdade, esta é uma prática bastante utilizada. Por exemplo:

```
int anos = 10;  
double rendimento = 1270.49;  
String s = "Em " + anos + "anos o rendimento será de " + rendimento;
```

Ao final das declarações acima, a variável **s** conterà a String "Em 10 anos o rendimento será de 127.49".

É possível concatenar qualquer tipo primitivo a uma String e mesmo qualquer tipo referência. No caso de concatenação de Strings com outros objetos, o valor substituído no lugar do objeto será um conjunto de caracteres com "@". Na prática, concatenação de Strings com objetos não tem muito sentido, mas é possível fazê-la .

Strings são tipos referência



Existem algumas características resultantes da manipulação de Strings que devem ser conhecidas:

Se duas variáveis do tipo *String* são declaradas (por atribuição) com exatamente o mesmo conteúdo:

```
String s1 = "Maria";
```

```
String s2 = "Maria";
```

As duas variáveis apontam para a mesma referência do objeto cujo conteúdo é "Maria".

Por outro lado, qualquer manipulação realizada sobre uma das Strings, irá gerar uma nova String, resultando em uma nova referência a objeto, ainda que o conteúdo das mesmas permaneçam inalterados:

```
s1 = s1 + "Souza";
```

```
s2 = s2 + "Souza";
```

No exemplo acima, houve manipulação das Strings *s1* e *s2*, resultando na criação de duas novas Strings, ou seja, duas novas referências, uma para cada objeto diferente. Embora o conteúdo das mesmas continue o mesmo (as strings foram manipuladas, mas o valor continua sendo o mesmo em cada uma) as referências de objetos não são as mesmas.

Outra característica importante da manipulação de Strings é que se as mesmas forem criadas utilizando o construtor:

```
String s1 = new String("Maria");
```

```
String s2 = new String("Maria");
```

Duas novas referências de objetos sempre são geradas, independente do conteúdo agregado às mesmas.

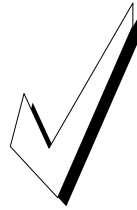
Igualdade de Strings

- ▶ Para testar se dois Strings são iguais, não deve ser usado `==`
- ▶ Deve-se usar o método `equals`:

`s1 == s2`



`s1.equals(s2)`



Qualiti Software Processes
Java Básico | 140



Copyright © 2002 Qualiti. Todos os direitos reservados.

String é um tipo referência. Neste caso, o operador de comparação `==` funciona comparando se duas variáveis diferentes apontam para o mesmo endereço de memória.

Caso se deseje comparar o conteúdo de duas Strings, o método `equals` deve ser utilizado.

Se o conteúdo de duas Strings for o mesmo, método `equals` retorna **true** como resposta. Do contrário, retorna **false**.

Igualdade de Strings

```
//Cria dois novos Strings
String s1 = "George";
String s2 = "George";

// Nesse momento, s1==s2 é verdadeiro!

s1 = s1 + " Bush"; // Cria um novo string e o atribui para s1
s2 = s2 + " Bush"; // Cria um novo string e o atribui para s2

if (s1 == s2)
    System.out.println("s1 e s2 sao os mesmos objetos.");
else
    System.out.println("s1 e s2 NAO sao os mesmos objetos.");

if (s1.equals(s2))
    System.out.println("s1 e s2 possuem o mesmo conteúdo.");
else
    System.out.println("s1 e s2 NAO possuem o mesmo
                        conteúdo.");
```

Para o seguinte exemplo:

```
String s1 = "George";
String s2 = "George";
```

Ao aplicarmos `s1 == s2` e `s1.equals(s2)`, a resposta é **true** para ambas declarações. Ou seja, tanto `s1` quanto `s2` apontam para o mesmo endereço de memória e possuem o mesmo conteúdo, respectivamente.

Por outro lado, qualquer modificação realizada sobre a referência da variável do tipo `String` resultará na criação de um novo objeto do tipo `String`, como ilustrado no exemplo a seguir:

```
String s1 = s1 + "Bush";
String s2 = s2 + "Bush";
```

A variável `s1` foi alterada utilizando uma referência da própria variável. O mesmo acontece com `s2`. Neste caso, apesar do conteúdo de ambas terem sido alterados elas permanecerem com o mesmo valor. No entanto, dois novos objetos são criados, a partir das modificações realizadas.

Neste caso, se aplicarmos `s1 == s2`, teremos **false** como resposta e se aplicarmos `s1.equals(s2)` teremos **true** como resposta. Ou seja, `s1` e `s2` são objetos diferentes e possuem o mesmo conteúdo, respectivamente.

Strings: Comparação e comprimento

- ▷ boolean **equals**(umString)
- ▷ boolean **equalsIgnoreCase**(umString)
- ▷ int **length**()

```
String a = "Sharon Stone";  
String b = "sharon stone";  
int comprimento = a.length();  
boolean resposta1 = a.equals(b);  
boolean resposta2 = a.equalsIgnoreCase(b);  
boolean resposta3 = b.equalsIgnoreCase(a);
```

Qual o valor das respostas?

Copyright © 2002 Qualiti. Todos os direitos reservados.

String: tratamento

- ▷ String `toLowerCase()`
- ▷ String `toUpperCase()`
- ▷ String `trim()`

```
String x = " Bom Dia! ";  
String y = x.toUpperCase();  
String z = x.toLowerCase();  
String w = x.trim();  
System.out.println(y);  
System.out.println(z);  
System.out.println(w);
```

```
BOM DIA!  
bom dia!  
Bom Dia!
```

Copyright © 2002 Qualiti. Todos os direitos reservados.

Strings: índices e substrings

Índices em Java
começam a partir de 0

Retorna um substring de
índiceInício até **índiceFinal-1**

- ▶ **int indexOf(umString)**
- ▶ **String substring(int índiceInício, int índiceFinal)**
- ▶ **char charAt(int índice)**

```
String x = "Pernambuco";  
String y = x.substring(0,5);  
String z = x.substring(6,10);  
int indice = x.indexOf("na");  
char letra = x.charAt(5);  
System.out.println(x); System.out.println(y);  
System.out.println(z); System.out.println(indice);  
System.out.println(letra);
```

Qual é a saída?

Copyright © 2002 Qualiti. Todos os direitos reservados.

Tipos Enumerados

Copyright © 2002 Qualiti. Todos os direitos reservados.

Tipo Enumerável

- ▶ Representa um conjunto de constantes que pertencem a uma única abstração.
- ▶ O conceito foi trazido do C++, porém em Java todo tipo enum é uma classe.
- ▶ Todo enum estende implicitamente de `java.lang.Enum`, por isso não pode estender mais nenhuma classe.
- ▶ Assim como Strings Java possui um suporte especial ao tipo Enum permitindo que na maior parte do tempo ele seja tratado como tipo primitivo.
- ▶ Foi adicionado na versão 5.0 (Tiger) da linguagem.

Copyright © 2002 Qualiti. Todos os direitos reservados.

Tipo Enumerável

- ▶ A declaração de um tipo enum é semelhante a declaração de uma classe
- ▶ Os elementos devem ser separados por virgula

```
public enum DiasDaSemana {  
    DOMINGO,  
    SEGUNDA,  
    TERCA,  
    QUARTA,  
    QUINTA,  
    SEXTA,  
    SABADO  
}
```

Tipo Enumerável

- Podemos usar o tipo enum como um primitivo fazendo uma inicialização implícita

```
public static void main (String args){  
  
    DiasDaSemana dia;  
  
    dia = DiasDaSemana.DOMINGO;  
  
    System.out.println( "Dia setado: " + dia );  
  
}
```

Dia setado: DOMINGO

Output

Tipo Enumerável

- ▶ Como todo enum é uma classe podemos adicionar métodos e atributos a ele.
- ▶ Os enums só aceitam construtores com visibilidade private ou package (default)

Copyright © 2002 Qualiti. Todos os direitos reservados.

Tipo Enumerável

```
public enum DiasDaSemana {
```

```
    DOMINGO (2),  
    SEGUNDA (1),  
    TERCA (1),  
    QUARTA (1),  
    QUINTA (1),  
    SEXTA (1),  
    SABADO (1.5F);
```

Passagem de parâmetro
ao construtor

Atributo do enum

```
    private float multiplicadorHora;
```

```
    DiasDaSemana( float multiplicador){  
        this.multiplicadorHora = multiplicador;  
    }
```

```
    public float getMultiplicadorHora(){  
        return .multiplicadorHora;  
    }
```

```
}
```

Tipo Enumerável

```
public static void main (String args){  
  
    float valorHoraTrabalho = 20.00F;  
    float QuantidadeHorasTrabalhadas = 8F;  
    float valorASerRecebido = valorHoraTrabalho * QuantidadeHorasTrabalhadas;  
  
    DiasDaSemana dia;  
  
    dia = DiasDaSemana.DOMINGO;  
    System.out.println( "Valor a receber pelo dia: " +  
                        valorASerRecebido * dia.getMultiplicadorHora() );  
  
    dia = DiasDaSemana.SEGUNDA;  
    System.out.println( "Valor a receber pelo dia: " +  
                        valorASerRecebido * dia.getMultiplicadorHora() );  
}
```

Output 1

1	Valor a receber pelo dia: 320
2	Valor a receber pelo dia: 160

Output 2

Arrays

Copyright © 2002 Qualiti. Todos os direitos reservados.

Arrays

- ▶ São tipos especiais de Java. Objetos especiais
- ▶ Arrays também são tipos referência
- ▶ Todos os elementos de um array são do mesmo tipo
- ▶ Arrays têm tamanho fixo depois de criados

Copyright © 2002 Qualiti. Todos os direitos reservados.



Declaração e criação de arrays

```
int[] a;  
double[] x;  
Cliente[] clientes;
```

```
int[] a = new int[100];
```

Primeiro item: **a[0]**
Último item: **a[99]**

```
String[] nomes = new String[200];
```

Inicialização de Arrays

```
int[] primosPequenos = {2, 3, 5, 7, 11,
    13};

String[] cores = {"Vermelho", "Azul", "Amarelo"};

double[] salarios = new double[5];

for (int i = 0; i<5; i++) {
    salarios[i] = i * 1000;
}
```

Copyright © 2002 Qualiti. Todos os direitos reservados.

Arrays multidimensionais

```
int[][] matriz;
```

Declaração não especifica dimensões

```
int[][] matriz = new int[10][5];
```

```
for (int i=0; i<10; i++)  
    for (int j=0; j<5; j++)  
        matriz[i][j] = 100;
```

Cria e inicializa um array bidimensional

```
long[][] x = { {0,1}, {2,3}, {4,5} };
```

Cria um array de 3 por 2

x[0][0]

x[0][1]

x[2][0]

Copyright © 2002 QualiTi. Todos os direitos reservados.

Acesso inválido

- ▶ Se é feito acesso a um elemento indefinido de um array, é gerada uma exceção:
 - `IndexOutOfBoundsException`

```
String nomes[] = {"José", "João", "Maria"};  
System.out.println(nomes[5]);
```

Gera um erro em tempo de execução

Enhanced For

- ▶ Nova utilização da estrutura de controle FOR
- ▶ Introduzida no Java 5.0 (Tiger)
- ▶ Simplifica a sintaxe e o entendimento
- ▶ Também chamado de “For each” (Para cada)

```
int numeros[] = {1,2,3,4,5,6,7,8,9,10};  
for( int item : numeros ){  
    System.out.println( item );  
}
```

Para cada elemento
do vetor “numeros”

Qualiti Software Processes
Java Básico | 158



Copyright © 2002 Qualiti. Todos os direitos reservados.

A nova sintaxe da estrutura de controle **for** tem por objetivo simplificar a escrita e leitura do código envolvido. Seu foco principal é o trabalho com Arrays e Coleções (A ser visto em Java Avançado).

Coleção de contas com array

- ▶ Uma classe que guarda contas num array de contas
 - Repositório ou conjunto de contas
- ▶ Métodos para inserção, procura, remoção e atualização dos objetos

Copyright © 2002 Qualiti. Todos os direitos reservados.

RepositorioContasArray

RepositorioContasArray
<u>+TAM CACHE CONTAS: int = 100</u> -contas: Conta[0..100] -indice: int
+RepositorioContasArray() +inserir(conta: Conta) +atualizar(conta: Conta) +remover(conta: Conta) -procurarIndice(numeroConta: String): int +existe(numeroConta: String): boolean +procurar(numeroConta: String): Conta

Copyright © 2002 QualiTi. Todos os direitos reservados.

RepositorioContasArray

```
public class RepositorioContasArray {  
    private Conta[] contas;  
    private int indice;  
    private final static int tamCache = 100;  
  
    public RepositorioContasArray() {  
        indice = 0;  
        contas = new Conta[tamCache];  
    }  
  
    public void inserir(Conta c){  
        contas[indice] = c;  
        indice = indice + 1;  
    }  
}
```

...

RepositorioContasArray

```
...  
  
private int procurarIndice(String num) {  
    int i = 0;  
    int ind = -1;  
  
    for( Conta c : contas ){  
        if ( (c.getNumero()).equals(num) ) {  
            ind = i;  
            break;  
        }  
        i++;  
    }  
  
    return ind;  
}  
  
...
```

RepositorioContasArray

```
...

    public boolean existe(String num) {
        boolean resp = false;

        int i = this.procurarIndice(num);

        if( i != -1){
            resp = true;
        }

        return resp;
    }

...
```

RepositorioContasArray

...

```
public void atualizar(Conta c){  
    int i = procurarIndice(c.getNumero());  
  
    if (i != -1) {  
        contas[i] = c;  
    } else {  
        System.out.println( "Conta nao encontrada" );  
    }  
  
}  
  
...
```

RepositorioContasArray

```
...

public Conta procurar(String num){
    Conta c = null;
    if (existe(num)) {
        int i = this.procurarIndice(num);
        c = contas[i];
    } else {
        System.out.println( "Conta nao encontrada" );
    }
    return c;
}

...
```

RepositorioContasArray

```
...

    public void remover(String num){
        if (existe(num)) {
            int i = this.procurarIndice(num);
            contas[i] = contas[indice - 1];
            contas[indice - 1] = null;
            indice = indice - 1;
        } else {
            System.out.println( "Conta nao encontrada" );
        }
    }

    ...
```

Referências do módulo

- ▶ Java Tutorial
 - <http://java.sun.com/docs/books/tutorial/>
- ▶ Documentação da Versão 6 da linguagem
 - <http://java.sun.com/javase/6/docs/>

Copyright © 2002 Qualiti. Todos os direitos reservados.



Módulo 6

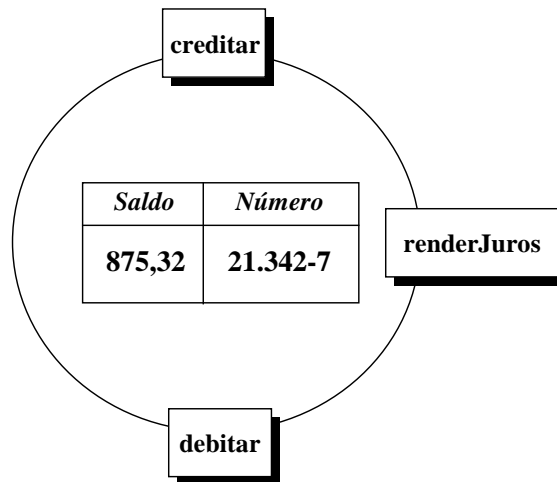
Herança, polimorfismo e ligação dinâmica

Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes

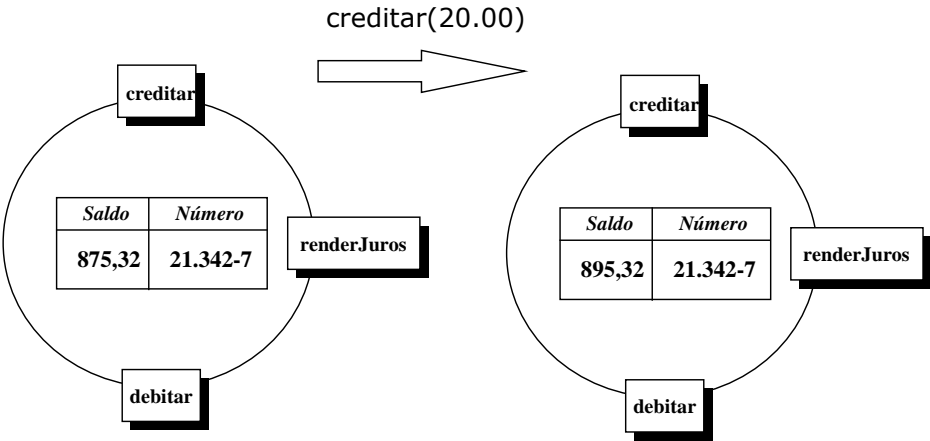


Objeto poupança



Copyright © 2002 Qualiti. Todos os direitos reservados.

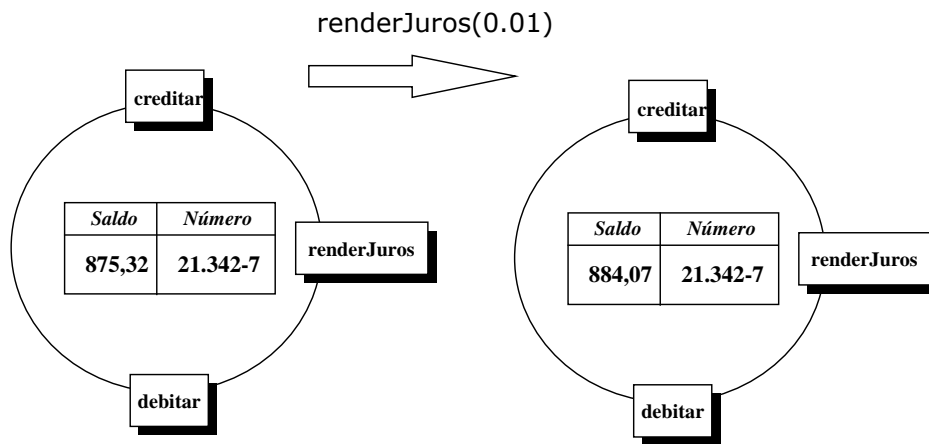
Estados do Objeto Poupança



Copyright © 2002 Qualiti. Todos os direitos reservados.



Estados do Objeto Poupança



Copyright © 2002 Qualiti. Todos os direitos reservados.

Implementação da classe Conta

Conta
-numero: String -saldo: double
+Conta(numero: String, saldo: double) +getNumero(): String +getSaldo(): double +creditar(valor: double) +debitar(valor: double)

```
class Conta{  
    private String numero;  
    private double saldo;  
  
    public Conta(String numero, double saldo) {  
        this.numero = numero;  
        this.saldo = saldo;  
    }  
  
    public String getNumero() {  
        return this.numero;  
    }  
  
    public double getSaldo() {  
        return this.saldo;  
    }  
  
    public void creditar(double valor) {  
        this.saldo += valor;  
    }  
  
    public void debitar(double valor) {  
        this.saldo -= valor;  
    }  
}
```

Classe de Poupanças: Assinatura

Conta	Poupanca
-numero: String -saldo: double	+numero: String +saldo: double
+Conta(numero: String, saldo: double) +getNumero(): String +getSaldo(): double +creditar(valor: double) +debitar(valor: double)	+Poupanca(numero: String, saldo: double) +getNumero(): String +getSaldo(): double +creditar(valor: double) +debitar(valor: double) +renderJuros(taxa: double)

Diferença entre Conta e poupança

Copyright © 2002 QualiTi. Todos os direitos reservados.

A classe Poupanca, neste exemplo, possui os mesmos atributos e métodos da classe Conta e ainda acrescenta um novo método, renderJuros().

Código das duas classes

<pre>class Conta{ private String numero; private double saldo; public Conta(String numero, double saldo) { this.numero = numero; this.saldo = saldo; } public String getNumero() { return this.numero; } public double getSaldo() { return this.saldo; } public void creditar(double valor) { this.saldo += valor; } public void debitar(double valor) { this.saldo -= valor; } }</pre>	<pre>class Poupanca{ private String numero; private double saldo; public Poupanca(String numero, double saldo) { this.numero = numero; this.saldo = saldo; } public String getNumero() { return this.numero; } public double getSaldo() { return this.saldo; } public void creditar(double valor) { this.saldo += valor; } public void debitar(double valor) { this.saldo -= valor; } public void renderJuros(double taxa) { this.saldo += (this.saldo * taxa); } }</pre>
--	---

Diferença entre Conta e poupança

Problemas

- ▶ Duplicação desnecessária de código:
 - A definição de `Poupanca` é uma simples extensão da definição de `Conta`
 - Clientes de `Conta` que precisam trabalhar também com `Poupanca` terão que ter código especial para manipular poupanças

- ▶ Falta refletir relação entre tipos do “mundo real”: uma poupança também é uma conta!

Copyright © 2002 Qualiti. Todos os direitos reservados.



Herança

- ▶ O mecanismo de herança permite reutilizar o código de classes existentes
- ▶ Apenas novos atributos ou métodos precisam ser definidos
- ▶ Herança introduz os conceitos de:
 - Superclasse e Subclasse
 - Redefinição de Métodos
 - Polimorfismo de subtipo

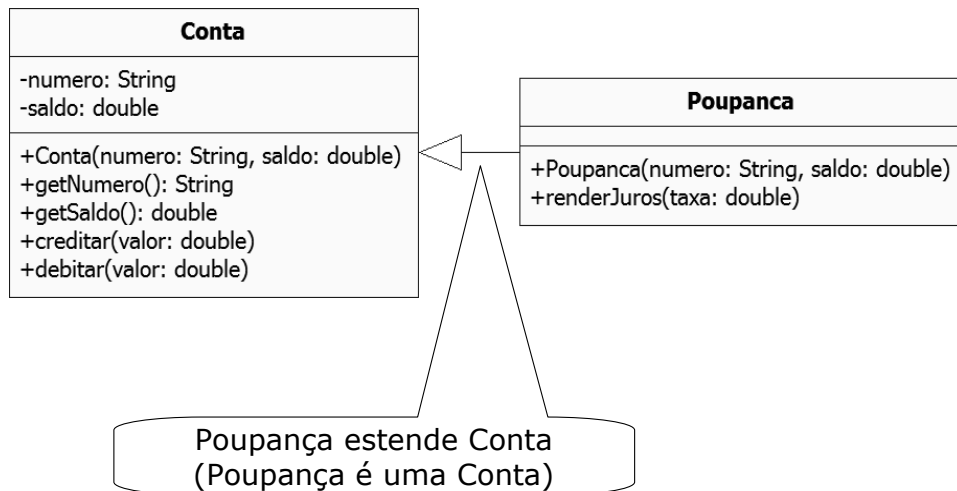
Copyright © 2002 QualiTi. Todos os direitos reservados.

Herança é um conceito do paradigma de Orientação a Objetos utilizado para reduzir a duplicação desnecessária de código.

Herança é uma forma de permitir a reutilização de software através da criação de novas classes a partir de classes existentes, absorvendo seus atributos e comportamentos e inserindo características que as novas classes exigem. Ou seja, através da utilização de herança, é possível estender comportamento e propriedades de uma classe, com o objetivo de criar uma outra classe de objetos que possua o mesmo comportamento e agreguem outros novos.

A classe que herda dados e comportamento de outra já existente é referida como **subclasse**. A classe da qual uma subclasse herda é referida como **superclasse**.

Nova classe Poupança (com herança)



Copyright © 2002 Quali. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 177



Ao criar uma nova classe, em vez de descrever atributos e métodos completamente novos, o programador pode determinar que a nova classe deve herdar os atributos e métodos de uma **superclasse** definida previamente. A nova classe, que herda de uma já existente, é referida como uma **subclasse**. Cada subclasse torna-se uma candidata a ser uma superclasse para alguma subclasse futura.

Uma subclasse pode ter somente uma superclasse direta. Em Java, este mecanismo é realizado utilizando a palavra-chave *extends*. Uma superclasse indireta é herdada de dois ou mais níveis acima na hierarquia da classe.

Java utiliza a abordagem de herança simples (uma classe é derivada somente de uma superclasse). Ao contrário de algumas linguagens, como C++, por exemplo, Java não suporta herança múltipla mas suporta a noção de interfaces. Interfaces serão discutidas no módulo 8 (Classes Abstratas e Interfaces).

Nova classe Poupança (com herança)

```
class Poupanca extends Conta{  
  
    public Poupanca(String numero, double saldo) {  
        super(numero,saldo);  
    }  
  
    public void renderJuros(double taxa) {  
        double saldoAtual = getSaldo();  
        creditar(saldoAtual * taxa);  
    }  
    //Somente isso  
}
```

```
subclasse extends superclasse
```

Copyright © 2002 Quali. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 178



Ao criar uma nova classe, em vez de descrever atributos e métodos completamente novos, o programador pode determinar que a nova classe deve herdar os atributos e métodos de uma **superclasse** definida previamente. A nova classe, que herda de uma já existente, é referida como uma **subclasse**. Cada subclasse torna-se uma candidata a ser uma superclasse para alguma subclasse futura.

Uma subclasse pode ter somente uma superclasse direta. Em Java, este mecanismo é realizado utilizando a palavra-chave *extends*. Uma superclasse indireta é herdada de dois ou mais níveis acima na hierarquia da classe.

Java utiliza a abordagem de herança simples (uma classe é derivada somente de uma superclasse). Ao contrário de algumas linguagens, como C++, por exemplo, Java não suporta herança múltipla mas suporta a noção de interfaces. Interfaces serão discutidas no módulo 8 (Classes Abstratas e Interfaces).

Herança

► Reuso de Código:

- tudo que a superclasse tem, a subclasse também tem
- o desenvolvimento pode se basear em o que já está pronto

► Extensibilidade:

- algumas operações da superclasse podem ser **redefinidas** na subclasse

Copyright © 2002 QualiTi. Todos os direitos reservados.

A capacidade de reutilização de software economiza tempo no desenvolvimento de aplicações.

Com a utilização de herança, é possível projetar e implementar programas que são mais facilmente extensíveis. Num primeiro momento, podem ser criadas classes genéricas para processar objetos de todas as classes existentes em uma hierarquia, por exemplo o objeto **Pessoa**. Em seguida, as classes que não existem durante o projeto (por não se saber previamente da necessidade destas ou por ainda não constarem nos requisitos do programa) podem ser adicionadas com pouca ou nenhuma modificação da classe genérica, contanto que estas classes façam parte da hierarquia que está sendo processada genericamente. Exemplos de classes que podem fazer parte da hierarquia para o exemplo citado podem ser **Funcionario**, **Gerente**, **Aluno**, **Professor**, entre outros.

Herança

► Comportamento:

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

► Princípio da Substituição:

- objetos da subclasse podem ser usados no lugar de objetos da superclasse
- Toda Poupança é uma Conta mas nem toda conta é uma Poupança

Copyright © 2002 QualiTi. Todos os direitos reservados.

Uma subclasse normalmente acrescenta seus próprios atributos e métodos; portanto, em geral, uma subclasse é maior que sua superclasse. Uma superclasse é mais genérica, uma vez que deve possuir atributos e métodos que são comuns às suas subclasses. Um subclasse é mais específica que sua superclasse e representa um grupo menor de objetos.

Quando o mecanismo de herança é utilizado, no início, a subclasse é essencialmente idêntica à superclasse. Depois, o programador pode definir nas subclasses adições, ou substituições, aos recursos herdados da superclasse.

Por este motivo, cada objeto de uma subclasse também é um objeto da superclasse dessa subclasse. Entretanto, o contrário não é verdadeiro - os objetos de superclasse não são objetos das subclasses dessa superclasse. Este relacionamento “um objeto da subclasse é um objeto da superclasse” pode ser utilizado para realizar algumas manipulações poderosas. Por exemplo, com base neste conceito é possível fazer uso do princípio da substituição, onde os objetos da subclasse podem ser utilizados no lugar de objetos da sua superclasse. Este é um dos princípios fundamentais da programação orientada a objetos.

Princípio da substituição

```
...  
Poupanca p;  
p = new Poupanca("21.342-7");  
p.creditar(500.87);  
p.debitar(45.00);  
System.out.println(p.getSaldo());  
...
```

Os métodos
creditar e
debitar são
herdados de
Conta

```
...  
Conta c;  
c = new Poupanca("21.342-7");  
c.creditar(500.87);  
c.debitar(45.00);  
System.out.println(c.getSaldo());  
...
```

Uma poupança
pode ser usada
no lugar de uma
conta

Substituição e Casts

- ▶ Nos contextos onde contas são usadas pode-se usar poupanças
 - Onde Conta é aceita, Poupança também será
- ▶ Nos contextos onde poupanças são usadas pode-se usar contas **com o uso explícito de casts**

Copyright © 2002 QualiTi. Todos os direitos reservados.



Da mesma forma que *casts* podem ser utilizados para conversão entre tipos primitivos, por exemplo, conversão de uma variável do tipo *float* para o tipo *int*, eles podem ser utilizados também para a conversão de referências de objetos de uma classe para outra.

O *cast* entre referências de objetos deve ser realizado quando se deseja utilizar um método existente somente em uma das classes (no contexto de uma hierarquia de herança). Por exemplo, objetos do tipo *Conta* (**superclasse**) podem ser utilizados no lugar de objetos do tipo *Poupança* (**subclasse**), uma vez que os mesmos participam da mesma hierarquia de herança. No entanto, se o método `renderJuros()`, declarado na classe *Poupança*, precisar ser utilizando, será necessário realizar um *casting* para esse objeto, uma vez que o mesmo não é visível a partir da classe *Conta*.

Casts

```
...
Conta c;
c = new Poupanca("21.342-7");
...
((Poupanca) c).renderJuros(0.01);
System.out.println(c.getSaldo());
...
```

cast

renderJuros só está disponível na classe Poupança. Por isso o cast para Poupança é essencial.

Copyright © 2002 QualiTi. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 183



Em Java toda referência de objeto tem um tipo. O tipo indica o tipo de objeto que a variável referencia e o que pode ser realizado a partir dela.

Se você atribui um objeto da subclasse à uma variável da superclasse, o compilador aceita, uma vez que todas as informações declaradas na superclasse, estão disponíveis também na subclasse. No entanto, na prática, somente os recursos (métodos e atributos) disponíveis na superclasse poderão ser acessados, havendo então uma “perda” em relação ao acesso ao recursos definidos somente na subclasse.

Nesta situação, para fazer uso do recursos da subclasse é necessário realizar um *casting* implícito, a partir do qual será possível ter acesso aos recursos definidos somente na subclasse.

A atribuição de um objeto da superclasse à uma variável da subclasse deve ser confirmada através de um *casting* explícito, do contrário um erro de compilação ocorrerá. Por exemplo:

```
Conta c = new Conta();
Poupanca p = c; //ERRO COMPILAÇÃO!
Poupanca p = (Poupanca)c; //COMPILAÇÃO OK
```

Isto acontece porque ao realizar uma atribuição de um objeto à uma referência de objeto, o objeto a ser atribuído deve possuir, no mínimo, os mesmos recursos disponíveis na referência. Uma superclasse, conceitualmente deve ter menos recursos que suas subclasses, por este motivo a atribuição implícita não é considerada pelo compilador.

instanceof

- O operador **instanceof** verifica a classe de um objeto (retorna true ou false)
- Recomenda-se o uso de instanceof antes de se realizar um cast para evitar erros

```
...  
Conta c = procura("123.45-8");  
  
if (c instanceof Poupanca)  
    ((Poupanca) c).renderJuros(0.01);  
else  
    System.out.print("Poupança inexistente!")  
...  

```

Copyright © 2002 QualiTi. Todos os direitos reservados.

O operador **instanceof** é bastante útil para verificar o tipo de objeto utilizado, quando trabalhando com herança. É uma boa prática descobrir se um objeto é ou não instância de uma determinada classe antes de realizar o *casting* para esse classe. Esta prática pode evitar erros de execução.

O **instanceof** retorna *true* se o objeto do lado direito for uma instância da classe ou implementa a interface do lado direito. Caso contrário, ele retorna *false*. Se o objeto do lado esquerdo é *null*, este operador também retorna *false*.

Herança e a classe Object

- ▶ Toda classe que você define tem uma superclasse
- ▶ Se não for usado "extends", a classe estende a classe "Object" de Java.
- ▶ A classe Object é a única classe de Java que não estende outra classe

```
class Cliente {}
```

São equivalentes!

```
class Cliente extends Object {}
```

Qualiti Software Processes
Java Básico | 185



Copyright © 2002 Qualiti. Todos os direitos reservados.

Toda classe Java estende a classe Object. Isto só não é verdadeiro quando a subclasse estende uma classe específica. Neste caso, a super classe estende, então, a classe Object.

Construtores e subclasses

```
class ContaBonificada extends Conta {  
    private double bonus;  
    ...  
    public contaBonificada(String num,double saldo)  
    {  
        super(num,saldo);  
    }  
}
```

super chama o construtor da superclasse

se **super** não for chamado,
o compilador acrescenta
uma chamada ao construtor
default: **super()**

se não existir um
construtor default na
superclasse, haverá
um erro de compilação

Quando uma classe é definida, Java garante que o seu construtor da classe é chamado sempre que uma instância dessa classe é criada. Esta regra também é válida quando uma instância de qualquer subclasse é criada.

Para que um objeto da subclasse seja criado, é necessário que ANTES, um objeto da sua superclasse possa ser criado. Isto é **mandatório** devido a hierarquia de herança entre as classes correspondentes aos dois objetos.

Para que este aspecto possa ser garantido, Java também exige que todo construtor da subclasse invoque o construtor da sua superclasse.

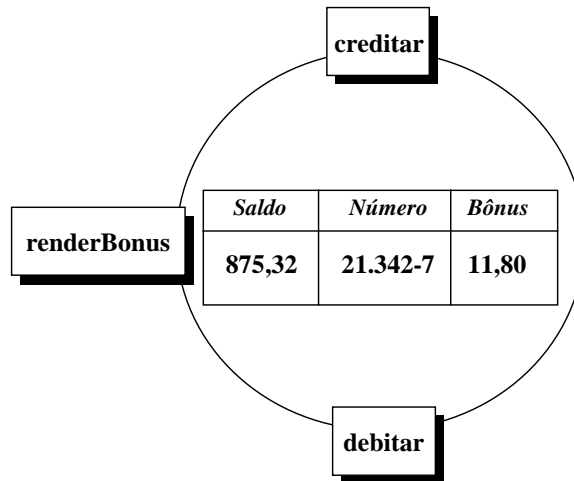
Para invocar o construtor da superclasse a palavra-chave *super* deve ser utilizada. Caso a primeira declaração no construtor da subclasse não seja uma chamada explícita ao construtor da superclasse, então Java inseri uma chamada implícita `super()`. Ou seja, a declaração `super ()` invoca o construtor sem argumentos da supeclasse.

Se a superclasse não tem construtor sem argumentos, a chamada EXPLÍCITA deve ser realizada a um construtor com argumento desta, usando `super(arg1, ar2, ar3,...)`. Caso contrário, um erro de compilação é lançado.

Overriding

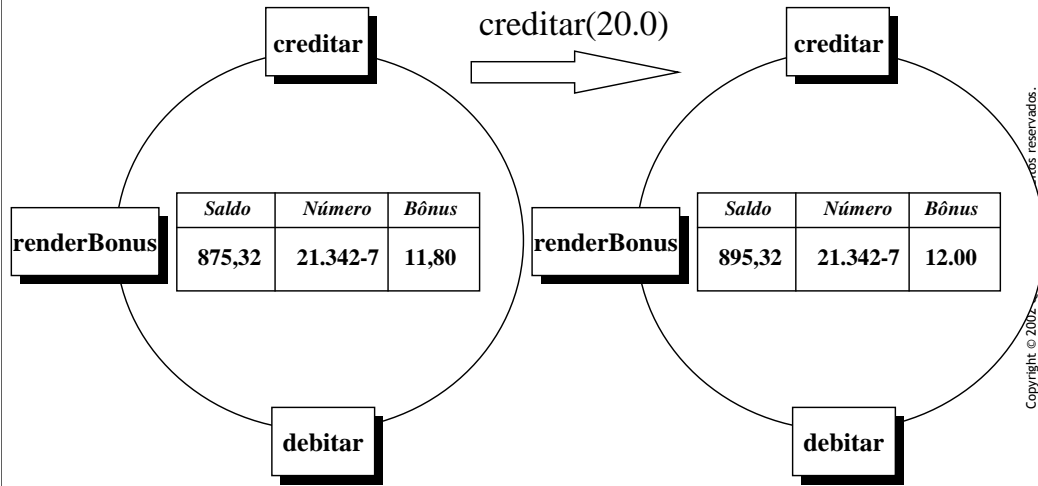
Copyright © 2002 Qualiti. Todos os direitos reservados.

Objeto Conta Bonificada

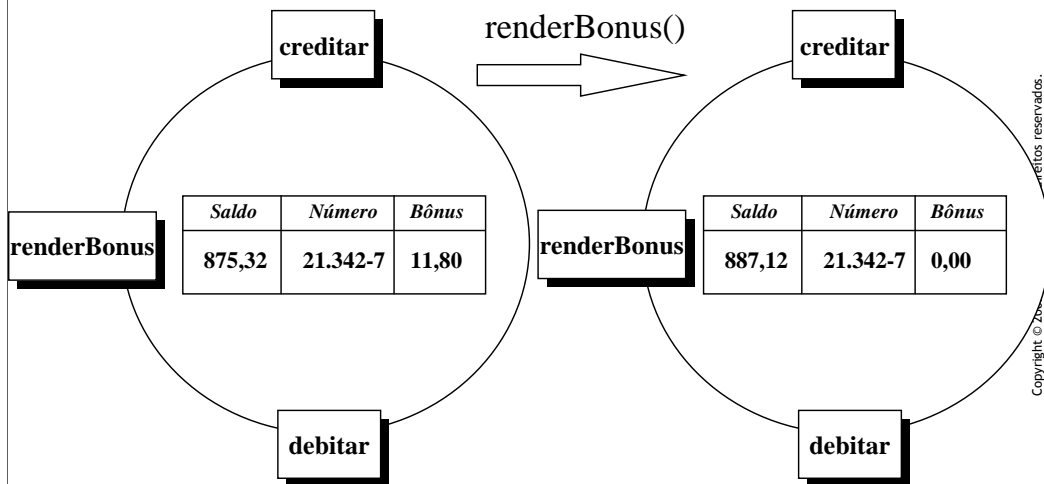


Copyright © 2002 QualiTi. Todos os direitos reservados.

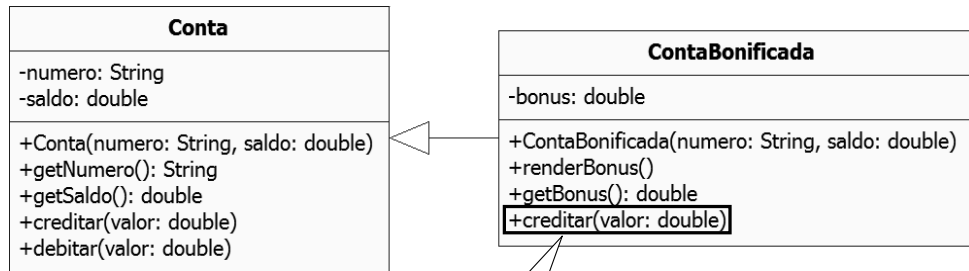
Estados de uma Conta Bonificada



Estados de uma Conta Bonificada



Contas Bonificadas: Assinatura



Redefinição do método

Copyright © 2002 Qualiiti. Todos os direitos reservados.

Contas Bonificadas: Assinatura

```
class ContaBonificada extends Conta{  
  
    private double bonus;  
  
    public ContaBonificada(String numero, double saldo) {  
        super(numero,saldo);  
    }  
  
    public String getBonus() { return this.bonus; }  
  
    public void renderBonus() {  
        super.creditar(this.bonus);  
        this.bonus = 0;  
    }  
  
    public void creditar(double valor) {  
        this.bonus = this.bonus + (valor * 0.01);  
        super.creditar(valor);  
    }  
}
```

Redefinição do
método creditar

Usando Contas Bonificadas

```
public static void main(String args[]){  
    ContaBonificada cb;  
  
    cb = new ContaBonificada( "21.342-7" ,100);  
  
    cb.creditar(200.00);  
    cb.debitar(100.00);  
    cb.renderBonus();  
  
    System.out.print(cb.getSaldo());  
  
}
```

Overriding

- ▷ Redefinição de métodos herdados da superclasse
- ▷ Para que haja a redefinição de métodos, o novo método deve ter a mesma assinatura (nome e parâmetros) que o método da super classe
- ▷ Se o nome for o mesmo, mas os parâmetros forem de tipos diferentes haverá overloading e não redefinição
- ▷ Redefinições de métodos devem preservar o comportamento (semântica) do método original
 - a semântica diz respeito ao estado inicial e estado final do objeto quando da execução do método

Copyright © 2002 QualiTi. Todos os direitos reservados.

A redefinição de métodos é uma técnica muito importante na programação orientada a objetos. Esta técnica é conhecida com o **Overriding**.

Quando uma classe define um método usando o mesmo nome, tipo de retorno e argumentos já definidos na superclasse desta, é dito que o método da subclasse redefine o método da superclasse.

Desta forma, quando este método é invocado sobre um objeto desta classe, o método redefinido é que é invocado e não, o método original definido na superclasse.

Com overriding, a semântica do método original não é modificada. Por exemplo, suponha que existe uma classe FormaGeometrica e suas subclasses Ellipse e Circulo. A superclasse define um método para calcular a área das figuras, `area()`. Este método `area()` é herdado pelas classes Ellipse e Circulo, mas o código utilizado para o cálculo da área de uma ellipse é diferente do mesmo utilizado para um círculo. Neste caso, este método deve ser redefinido em cada uma dessas classes para agregar este tipo de informação.

No exemplo citado, a forma como a área é calculada é que varia e não o SIGNIFICADO do método. Ou seja, ele continua calculando à área de uma determinada figura. Ou seja, a semântica original do mesmo não varia. Este é o conceito básico, vinculado ao uso de overrinding.

Polimorfismo e Ligações Dinâmicas

- ▶ Dois métodos com o mesmo nome e tipo:
 - qual versão do método usar?
- ▶ O método é escolhido dinamicamente (em tempo de execução), não estaticamente (em tempo de compilação)
- ▶ A escolha é baseada no tipo do objeto que recebe a chamada do método e não da variável

Copyright © 2002 Qualiti. Todos os direitos reservados.

O polimorfismo é uma das consequências do uso de herança. Com o polimorfismo é possível escrever programas de uma forma geral para tratar uma ampla variedade de classes relacionadas existentes e ainda a serem especificadas, tornando fácil adicionar novos recursos ao sistema.

Outro aspecto do uso de polimorfismo é que permite que a mesma mensagem (chamada de método) enviada a diferentes objetos resulte em um comportamento que depende da natureza do objeto que recebe a mensagem.

Por exemplo, quando uma mensagem é enviada para um objeto de uma subclasse em Java, a subclasse checa se ela tem um método com aquele nome e com exatamente os mesmos parâmetros. Se a resposta for positiva, ela usa esse método. Caso contrário, Java envia a mensagem à classe imediatamente superior.

Se toda a hierarquia for percorrida sem que um método apropriado seja encontrado um erro em tempo de compilação ocorre.

Este procedimento só é possível por causa do conceito de ligação dinâmica (*dynamic binding*), o qual é a chave para o polimorfismo. O compilador não gera código em tempo de compilação para uma chamada de método específica, ele gera código para determinar que método deve ser chamado em tempo de execução usando a informação de tipo disponível para o objeto.

Java oferece mecanismos para permitir a utilização de herança, bem como dos conceitos relacionados.

Ligações Dinâmicas

```
...  
Conta c1, c2;  
c1 = new ContaBonificada( "21.342-7" );  
c2 = new Conta( "12.562-8" );
```

```
c1.creditar(200.00);  
c2.creditar(100.00);  
c1.debitar(100.00);  
c2.debitar(60.00);
```

Qual é o **creditar**
chamado?

```
((ContaBonificada) c1).renderBonus();  
System.out.println(c1.getSaldo());  
System.out.println(c2.getSaldo());  
...
```

Overloading

Copyright © 2002 Qualiti. Todos os direitos reservados.

Overloading

- ▶ Quando se define um método com mesmo nome, mas com parâmetros de tipos diferentes, não há redefinição e sim

overloading ou sobrecarga

- ▶ Overloading permite a definição de vários métodos com o mesmo nome em uma classe. O mesmo vale para construtores
- ▶ A escolha do método a ser executado é baseada no tipo dos parâmetros passados

Overloading é uma técnica utilizada para definir um conjunto de métodos relacionados, com o mesmo nome, mas diferentes assinaturas. Métodos desta natureza, geralmente executam a mesma funcionalidade básica, mas permitem ao programador acrescentar novo comportamento ou novas restrições, especificando argumentos de diferentes formas.

Métodos *overloaded* são métodos que têm mesmo nome, diferentes tipos de parâmetros, diferentes números de parâmetros, ou o mesmo número de parâmetros em diferentes posições na lista de parâmetros. Um método não pode ser *overloaded* modificando somente o tipo de retorno (isto gera um erro de compilação). Método com o mesmo nome podem ter diferentes tipos de parâmetros somente se os parâmetros destes também são diferentes.

Overloading não é um conceito relacionado ao uso de herança, como o que ocorre com overriding. Ele pode ser utilizado em qualquer classe.

Overloading

```
class Formatacao {  
    static String formatar(double d, int  
        precisao){...}  
    static String formatar(double d) {...}  
    static String formatar(int d) {...}  
    ...  
}
```

```
//chama o primeiro método  
String s1 = formatar(10.0, 2);  
//chama o terceiro método  
String s2 = formatar(99);
```

Aqui o método **formatar**
tem três versões
"overloaded"

Copyright © 2002 Qualiti. Todos os direitos reservados.

Herança e Modificadores

Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes



Uso de `protected` e `private` em Herança

- ▶ Atributos e métodos com o modificador **`protected`** podem ser acessados na classe em que são declarados e nas suas subclasses
- ▶ Os membros **`private`** de uma superclasse são acessíveis apenas em métodos dessa superclasse

Copyright © 2002 Quali. Todos os direitos reservados.

Os atributos e métodos declarados em uma superclasse com o modificador de acesso **`protected`**, podem ser acessados somente por métodos da superclasse, métodos da subclasse e por métodos de classes localizadas no mesmo pacote.

O conceito de encapsulamento relacionado ao uso do modificador de acesso **`private`** é garantido, independente da hierarquia na qual a classe participa. Se fosse permitido uma subclasse ter acesso aos membros **`private`** da sua superclasse, a regra de encapsulamento seria quebrada. Por este motivo, ela é aplicada em qualquer circunstância.

Classes e métodos *final*

- ▶ Classes declaradas com o modificador *final* não podem ter subclasses

```
final class GeradorSenhas {  
    }  
}
```

- Usado por segurança
- String são exemplos de classes *final*

- ▶ Um método que é declarado *final* não pode ser redefinido em uma subclasse

Copyright © 2002 QualiTi. Todos os direitos reservados.

Uma classe declarada *final* não pode se estendida e cada método declarado em uma classe com esse modificador é implicitamente *final*.

O conceito de *overriding* não pode ser utilizado em um método herdado em uma subclasse, se este é declarado *final* na superclasse.

Referências do módulo

- ▶ Java Tutorial
 - <http://java.sun.com/docs/books/tutorial/>
- ▶ Documentação da Versão 6 da linguagem
 - <http://java.sun.com/javase/6/docs/>

Copyright © 2002 Qualiti. Todos os direitos reservados.



Referências do módulo

- ▶ Java Tutorial
 - <http://java.sun.com/docs/books/tutorial/>
- ▶ Documentação da Versão 6 da linguagem
 - <http://java.sun.com/javase/6/docs/>

Copyright © 2002 Qualiti. Todos os direitos reservados.



Módulo 7

Pacotes

Copyright © 2002 Qualiti. Todos os direitos reservados.

Tipos de Módulos em Java

► Classes e interfaces

- agrupam definições de métodos, atributos, construtores, etc.
- definem tipos

► Pacotes

- agrupam definições de classes e interfaces relacionadas
- estruturam sistemas extensos, facilitando a localização das classes e interfaces
- oferecem um nível mais alto de abstração: há mais classes do que pacotes

Copyright © 2002 QualiTi. Todos os direitos reservados.

Pacotes em Java devem ser utilizados como um mecanismo para estruturar (fisicamente) as classes e interfaces Java desenvolvidas em uma aplicação.

A estruturação de classes e interfaces em pacotes beneficia muito sistemas extensos com dezenas e mesmo centenas de arquivos Java, uma vez que facilita a localização de classes e interfaces.

Outro aspecto importante relacionado ao uso de pacotes é que os mesmos possibilitam a organização de classes e arquivos com conceitos relacionados em um mesmo pacote. Além disso, com pacotes também é possível garantir um nível de abstração maior ao sistema, uma vez que, na prática, existem mais classes e interfaces do que pacotes em uma aplicação.

Pacotes e Diretórios

- ▶ As classes de um pacote são definidas em arquivos com o mesmo cabeçalho:
`package nomeDoPacote;`
- ▶ Cada pacote é associado a um diretório do sistema operacional:
 - Os arquivos `.class` das classes compiladas do pacote são colocados neste diretório
 - É recomendável que o código fonte das classes do pacote também esteja neste diretório

Copyright © 2002 Qualiti. Todos os direitos reservados.

Cada pacote equivale na verdade a um diretório do sistema operacional. Neste diretório os arquivos fonte (`.java`) devem ser armazenados. Além disso, os arquivos `.class` também podem ser armazenados no mesmo diretório.

Todas as classes e interfaces organizadas em pacotes devem declarar onde estão localizadas. Isto implica que dentro de um arquivo Java declarado em um pacote, a primeira linha existente no mesmo deve ser a declaração do pacote do qual o mesmo faz parte. Esta declaração deve ser realizada utilizando a palavra chave **package** acrescida no nome do pacote e finalizada com o `;` (ponto-e-vírgula). Além disso, esta declaração deve, obrigatoriamente aparecer antes da declaração da definição da classe ou interface. Por exemplo:

```
package banco.contas;  
public classe Conta {  
    /* corpo da classe Conta */  
}
```

Nomes de Pacotes

- ▶ O nome de um pacote é parte do nome do seu diretório associado: o pacote

`qualiti.banco.conta`

deve estar no diretório

`c:\java\qualiti\banco\conta`

assumindo que o compilador Java foi informado para procurar pacotes em

`c:\java`

Copyright © 2002 Qualiti. Todos os direitos reservados.

Pacotes e Subdiretórios

- ▶ Subdiretórios não correspondem a "subpacotes". São subdiretórios como outros quaisquer
- ▶ Por exemplo, não existe nenhuma relação entre `exemplos` e `exemplos.banco`:

```
package exemplos;  
import exemplos.banco.*;  
/*...*/
```

```
package exemplos.banco;  
/*...*/
```

Qualiti Software Processes
Java Básico | 209



Copyright © 2002 Qualiti. Todos os direitos reservados.

Embora um pacote esteja diretamente relacionando a um diretório no sistema de arquivos, subdiretórios não correspondem a subpacotes. Este conceito não existe em Java. Independente de onde o arquivo Java está localizado, o diretório será considerado como sendo um pacote único.

Por exemplo, uma classe denominada `Exemplo1` pode estar armazenada no seguinte diretório do sistema operacional, `c:\exemplos` e outra denominada `Exemplo2` pode estar armazenada no diretório `c:\exemplos\banco`.

Neste exemplo, a classe `Exemplo1` deve ser definida no pacote de nome **`exemplos`** e a classe `Exemplo2` deve ser definida no pacote de nome **`exemplos.banco`**. Apesar de no sistema de arquivos um ser subdiretório do outro, em arquivos Java cada um representa um pacote diferente que não possuem relacionamentos entre si.

Pacotes e modificadores de acesso

- ▷ `public`
 - Elementos com este modificador podem ser utilizados (são visíveis) em qualquer lugar, mesmo em pacotes diferentes
- ▷ `protected`
 - Elementos com este modificador só podem ser utilizados no pacote onde são declarados, ou nas subclasses da classe onde são declarados
- ▷ “friendly” (sem modificador)
 - Elementos com nível de acesso default só podem ser utilizados no pacote onde estão declarados
- ▷ `private`
 - Elementos declarados com este modificador só podem ser utilizados na classe onde estão declarados

Copyright © 2002 Qualiti. Todos os direitos reservados.

O uso de pacotes também possuem relação direta com o uso de modificadores de acesso:

- Os elementos (atributos e métodos) de classes Java definidos com visibilidade *public* podem ser acessados por classes armazenadas em qualquer pacote.
- Os elementos (atributos e métodos) de classes Java definidos com visibilidade *protected* podem ser acessados somente por classes declaradas no mesmo pacote ou que sejam subclasses da classe onde estes são definidos.
- Os elementos (atributos e métodos) de classes Java definidos com visibilidade *friendly* podem ser acessados somente por classes armazenadas no mesmo pacote.
- Os elementos (atributos e métodos) de classes Java definidos com visibilidade *private* podem ser acessados por pelas próprias classes onde são definidos.

Reuso de Declarações

- ▶ As declarações feitas em um arquivo são visíveis em qualquer outro arquivo do mesmo pacote, a menos que elas sejam private
- ▶ Qualquer arquivo de um pacote pode usar as definições visíveis de outros pacotes, através do mecanismo de importação de pacotes

Copyright © 2002 Qualiti. Todos os direitos reservados.



Respeitando as regras do uso de modificadores em classes armazenadas em pacotes, para um arquivo Java (classes ou interfaces) ter acesso ao outro, os mesmos não precisam estar declarados no mesmo pacote.

Para que as definições de um arquivo possam ser visíveis por outro arquivo, este último deve utilizar o mecanismo de importação Java para ter acesso a essas informações.

Importação de Pacotes

► Importação de definição de tipo específica:

```
package segundo.pacote;  
import primeiro.pacote.NomeDaClasse;  
/*...*/
```

► Importação de todas as definições de tipo públicas:

```
package segundo.pacote;  
import primeiro.pacote.*;  
/*...*/
```

Copyright © 2002 QualiTi. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 212



Importação Java **não** representa que um arquivo será importado (em memória) para que outro possa ter acesso ao mesmo. O termo importar aqui simplesmente serve para indicar ao compilador onde o mesmo deve procurar as definições exigidas pela classe que está acessando uma outra.

Para tal, a classe que utiliza os elementos de outra deve declarar antes de definição da classe e logo depois de definição do pacote a palavra chave **import** seguida pelo nome no pacote, um ponto e o nome da classe, como no exemplo a seguir:

```
package segundo.pacote;  
Import primeiro.pacote.NomeDaClasse;  
...
```

Para cada classe a ser utilizada por uma outra, a declaração com o formato acima deve ser realizada.

Uma outra variação dessa declaração é informar ao compilador a localização de todas as classes de um pacote completo em vez de especificar uma de cada vez. Neste caso, no lugar do nome da classe o asterisco pode ser utilizado:

```
package segundo.pacote;  
Import primeiro.pacote.*;  
...
```

Importação de Pacotes: Detalhes

- ▶ Tanto `NomeDaClasse` quanto `primeiro.pacote.NomeDaClasse` podem ser usadas no corpo de classes pertencentes a `segundo.pacote`
- ▶ Em `segundo.pacote`, não pode ser definida uma classe com o nome `NomeDaClasse`, caso a importação tenha sido específica

Copyright © 2002 Qualiti. Todos os direitos reservados.

Importação de Pacotes: Mais Detalhes

```
package segundo.pacote;  
  
public class NomeDaClasse {  
    /*...*/  
}  
  
package primeiro.pacote;  
  
public class NomeDaClasse {  
    /*...*/  
}
```

Os exemplos 1 e 3 apresentam problemas!

1

```
import segundo.pacote.*;  
import primeiro.pacote.*;
```

2

```
import segundo.pacote.NomeDaClasse;  
import primeiro.pacote.*;
```

3

```
import segundo.pacote.NomeDaClasse;  
import primeiro.pacote.NomeDaClasse;
```

Copyright © 2002 QualiTi. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 214



Diferente pacote podem possuir classes com o mesmo nome. Por exemplo, os pacotes **primeiro.pacote** e **segundo.pacote** podem ter definida em cada um uma classe chamada **NomeDaClasse**.

Quando esta situação acontece um cuidado especial deve ser tomado quando uma classe qualquer de um outro pacote resolve utilizar definições declaradas nas duas classes **NomeDaClasse**.

Neste caso, esta classe deve realizar o import para as duas classes. Como ambas possuem o mesmo nome, a declaração de import para as mesmas, quando utilizadas em uma mesma classe, devem seguir algumas regras especiais:

- A declaração dos imports não pode ser feita da seguinte forma:

```
import primeiro.pacote.*;  
import segundo.pacote.*;
```

Se declarados dessa forma, quando uma das duas classes for instanciada, ou mesmo declarada, o compilador irá acusar um erro de ambigüidade. Como as classes possuem o mesmo nome ele não saberá em qual pacote deve procurar.

- Da mesma forma, as classes também não devem ser importadas como a seguir:

```
import primeiro.pacote.NomeDaClasse;  
import segundo.pacote.NomeDaClasse;
```

Pois o compilador irá gerar um erro acusando que existe declaração de import duplicada.

Estruturando Aplicações com Pacotes

- ▶ Agrupar classes relacionadas, com dependência (de implementação ou conceitual) entre as mesmas
- ▶ Evitar dependência mútua entre pacotes:

```
package a;  
import b.*;  
/*...*/
```

```
package b;  
import a.*;  
/*...*/
```

Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 215



Para que os imports possam ser realizados com sucesso, os mesmos devem ser realizados da seguinte forma:

```
import primeiro.pacote.NomeDaClasse;  
import segundo.pacote.*;
```

Ou

```
import primeiro.pacote.*;  
import segundo.pacote. NomeDaClasse;
```

Por *default*, quando uma das classes for instanciada, a instância criada será do tipo da classe declarada explicitamente no import. Por este motivo, é necessário que a classe que tiver sido declarada com o import usando asterisco acrescente o nome completo do pacote quando da criação de uma instância de seu tipo. Por exemplo, para o primeiro formato desse exemplo uma instância de NomeDaClasse deveria ser criada da seguinte forma:

```
NomeDaClasse c = new segundo.pacote.NomeDaClasse();
```

Outro aspecto importante a ser considerado com estruturação de aplicações em pacotes é que deve ser evitada a estruturação de classes em pacotes diferentes quando as mesmas possuem uma dependência cíclica, ou seja, uma classe A acessa uma classe B e a classe B também acessa a classe A. O compilador pode se confundir na ordem de compilação das classes.

Estruturando Aplicações com Pacotes

- ▶ Estruturação típica de um sistema de informação:
 - Vários pacotes para as classes das interfaces de GUI, um para cada conjunto de telas associadas
 - Um pacote para a classe fachada e exceções associadas
 - Um pacote para cada coleção de negócio, incluindo as classes básicas, coleções de dados, interfaces, e exceções associadas
 - Um pacote (sistema).util contendo classes auxiliares de propósito geral

Copyright © 2002 QualiTi. Todos os direitos reservados.

Quando estruturando aplicações empacotes é importante levar a organização agrupando em mesmos pacotes as classes conceitualmente relacionadas.

Por exemplo, um pacote `banco.clientes` poderia conter todas as classes conceitualmente relacionadas à classe cliente da arquitetura em camadas (`CadastroClientes`, `RepositorioClientesArray`, `RepositorioClientes`, `Cliente`, `ClienteInexistenteException` e `ClienteExistenteException`).

Para classes relacionadas à classe `Conta`, por exemplo, o pacote `banco.contas` poderia ser criado. Esta filosofia pode ser seguida para os demais tipos de classes de um sistema.

Além disso, para classes auxiliares de um sistema, ou seja, classes que podem ser utilizadas por outras classes em diferentes pacotes, poderia ser criado também um pacote `banco.util`. Este pacote ficaria responsável por armazenar classes auxiliares de propósito geral.

Referências do módulo

- ▶ Java Tutorial
 - <http://java.sun.com/docs/books/tutorial/>
- ▶ Documentação da Versão 6 da linguagem
 - <http://java.sun.com/javase/6/docs/>

Copyright © 2002 Qualiti. Todos os direitos reservados.



Módulo 8

Arquitetura em camadas

Uma abordagem para
estruturação de aplicações

Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes



Benefícios

► Modularidade:

- Dividir para conquistar
- Separação de conceitos
 - Coesão
 - Melhor estruturação do sistema
 - Melhor entendimento
- Reusabilidade
- Extensibilidade

► Facilidade de manutenção:

- Custos de manutenção representam em média 70% do custo total do software
- Mudanças em uma camada não afetam as outras, desde que as interfaces sejam preservadas (plug and play)

Copyright © 2002 QualiTi. Todos os direitos reservados.

A arquitetura em camadas é um padrão de arquitetura amplamente divulgado na literatura para a estruturação de aplicações vislumbrando alguns benefícios como modularidade e extensibilidade de sistemas.

Este padrão indica como um sistema deve ser dividido em camadas para obter tais aspectos. A forma como cada camada deve ser implementada não é algo definido pelo mesmo. Por este motivo, neste módulo é apresentada uma possível forma de implementação para as camadas presentes na arquitetura em camadas, de modo a alcançar os objetivos que a mesma propõe. Variações do padrão arquitetural em camadas também são amplamente utilizadas em aplicações comerciais que têm como premissa a busca pelo desenvolvimento de aplicações modulares e extensíveis.

Com o uso da arquitetura em camadas é possível adotar o jargão “**Dividir para conquistar**”. Com esta abordagem é possível construir sistemas modulares com separação de conceitos (interface com o usuário, regras de negócio e acesso ao mecanismo de armazenamento persistente) bem definida, facilitando o entendimento dos mesmos, bem como possibilitando a adição de fatores de qualidade como reusabilidade e extensibilidade. Além disso, a estruturação de aplicações em camadas tende a facilitar a manutenção das mesmas, uma vez que mudanças relativas ao código de uma camada não devem afetar as camadas adjacentes e subjacentes.

Benefícios

- ▶ Uma mesma versão de uma camada trabalhando com diferentes versões de outra camada
 - várias GUIs para a mesma aplicação
 - vários mecanismos de persistência suportados pela mesma aplicação
 - várias plataformas de distribuição para acesso a uma mesma aplicação

Copyright © 2002 Qualiti. Todos os direitos reservados.



Com a arquitetura em camadas independentes é possível promover a modularidade e reusabilidade de aplicações. Suas recomendações levam à construção de sistemas onde o código relativo às regras de negócio fica separado do código responsável pela comunicação, acesso a dados e interface com o usuário (GUI).

A separação do código em camadas independentes permite que se troque a interface gráfica ou o meio de armazenamento dos dados (arquivos por um SGBD, por exemplo) sem afetar as regras de negócio do sistema. Isso facilita a reusabilidade das classes básicas do negócio em outros projetos e permite maior flexibilidade na escolha de tecnologias para implementar a aplicação.

Características

- ▶ **Propósito geral de cada camada:**
 - Fornecer suporte para alguma camada superior
 - Abstrair as camadas superiores de detalhes específicos
- ▶ **Propósito específico de cada camada:**
 - Preocupar-se com os detalhes específicos que serão ‘escondidos’ das camadas superiores

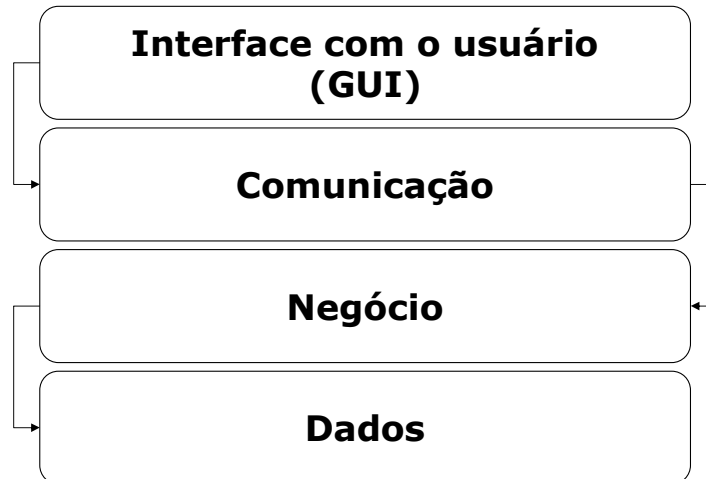
Copyright © 2002 QualiTi. Todos os direitos reservados.



Cada camada da arquitetura em camadas tem como propósito geral fornecer serviços para a camada superior, abstraindo a mesma dos detalhes específicos sobre os serviços que estão sendo fornecidos.

Cada camada específica deve, então se preocupar com os detalhes inerentes ao seu domínio e que devem ser escondidos das camadas superiores. Cada camada deve implementar mecanismos que permitam que a camada superior não precise saber dos detalhes de implementação da camada inferior.

Estrutura de Camadas



Copyright © 2002 Qualiiti. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 222



A figura ilustra quais são as camadas padrão de uma aplicação e que serviços cada uma delas deve apresentar. A tecnologia usada para implementar cada uma das camadas não é tratada aqui, pois deve ser escolhida de acordo com as características específicas de cada projeto.

A camada de **Interface com o usuário** deve conter classes responsáveis por gerar a interface pela qual a aplicação será utilizada pelo usuário desta.

A camada de **Comunicação**, é responsável por distribuir a aplicação, no caso de aplicações distribuídas em máquinas remotas. A mesma tem como objetivo esconder das camadas adjacentes e subjacentes aspectos de implementação específicos da tecnologia sendo utilizada para a distribuição de aplicações, bem como prover a comunicação entre a GUI e o restante da aplicação (negócio do sistema).

A camada de **Negócio**, tem como objetivo prover as regras de negócio do sistema. Ou seja, ela é responsável por definir a lógica de negócio do sistema.

A camada de **Dados**, é responsável pela manipulação da estrutura física de armazenamento dos dados.

Estas camadas se comunicam através de interfaces, o que possibilita que as mesmas possam esconder detalhes de implementação inerentes a cada uma.

Camada de negócios

- ▶ Responsável por implementar a lógica do negócio
- ▶ Classes do domínio da aplicação
 - classes básicas do negócio
 - coleções de negócio
 - fachada do sistema

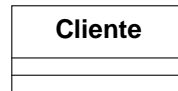
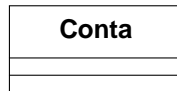
Copyright © 2002 QualiTi. Todos os direitos reservados.



Esta camada é o núcleo do sistema, responsável por implementar a lógica do negócio. Nela estão todas as classes inerentes ao domínio da aplicação, como as **classes básicas do negócio**, as **coleções de negócio**, os **controladores** e a classe **fachada** do sistema (tabela abaixo).

Classes básicas do negócio

- Representam conceitos básicos do domínio da aplicação
- Chamadas de Entidades (Entity) do sistema

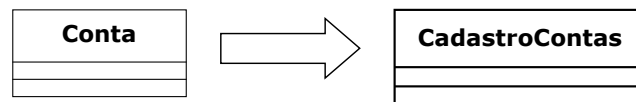


Copyright © 2002 QualiTi. Todos os direitos reservados.

As classes básicas representam objetos básicos manipulados pelo sistema e que normalmente são, direta ou indiretamente, persistidas pelo sistema. Uma aplicação bancária, por exemplo, possui as classes básicas Cliente e Conta, necessárias para representar dois conceitos centrais ao domínio da aplicação

Coleções de negócio

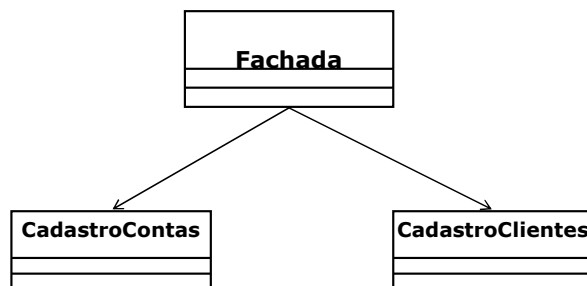
- ▶ Representam conjuntos de objetos
- ▶ Responsáveis pela inclusão, remoção, atualização e consultas a instâncias das classes básicas
- ▶ Encapsulam as verificações e validações relativas ao negócio
- ▶ Utilizam coleções de dados como suporte para o armazenamento de objetos



As coleções de negócio representam conjuntos (repositórios) de classes básicas e são responsáveis pelo armazenamento, temporário ou não, das instâncias destas classes. Nas coleções de negócio estão encapsuladas regras de *negócio* associadas ao armazenamento e manipulação das instâncias. Na aplicação bancária, por exemplo, a inclusão, remoção ou atualização de uma conta é responsabilidade da classe CadastroContas.

Fachada do sistema

- ▶ Segue o padrão de projeto *Facade*
- ▶ Representa os **serviços** oferecidos pelo sistema
- ▶ Centraliza as instâncias das coleções de negócio
 - Realiza críticas de restrição de integridade
- ▶ Gerencia as **transações** do sistema



Qualiti Software Processes
Java Básico | 226



Copyright © 2002 Qualiti. Todos os direitos reservados.

A fachada representa os serviços que são oferecidos pelo sistema, podendo implementar uma ou mais interfaces públicas para oferecer diferentes visões dos serviços disponíveis. Esta classe pode centralizar todas as instâncias das coleções de negócio da aplicação, mas deve ser o único ponto de acesso ao sistema (razão pela qual costuma ser construída segundo o padrão de projeto *Singleton*). A fachada pode encapsular as verificações e validações inerentes ao negócio (escolha do arquiteto) como, por exemplo, verificar se uma determinada instância já está cadastrada antes de inseri-la.

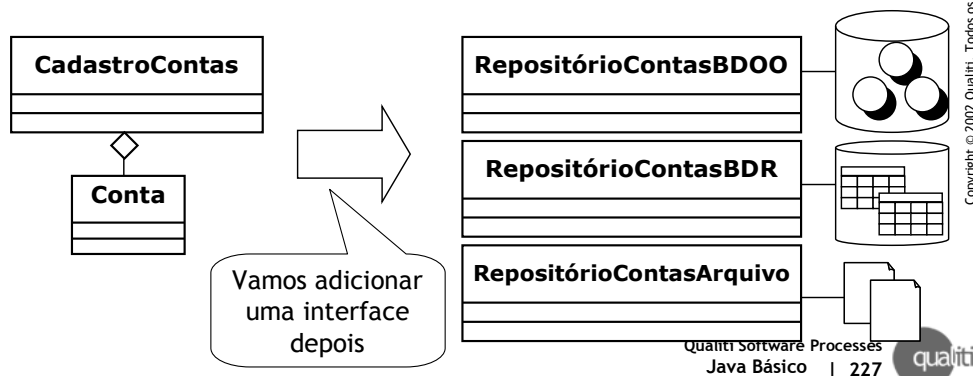
A classe Fachada, por exemplo, pode conter as instâncias das classes CadastroClientes e CadastroContas, assim como métodos para saque, extrato, transferência entre contas, enfim, para todos os serviços desejáveis em uma aplicação deste tipo.

Classes fachada seguem o padrão de projeto *Facade*, documentado no livro Design Patterns, de Gamma et. al, página 185.

O padrão Singleton está documentado no livro Design Patterns, de Gamma et. al, página 127.

Camada de dados

- ▶ Responsável pela manipulação da estrutura de armazenamento dos dados
- ▶ Isola o resto do sistema do mecanismo de persistência utilizado
- ▶ Classes de coleções de dados
- ▶ Executam inclusões, remoções, atualizações e consultas no meio de armazenamento usado

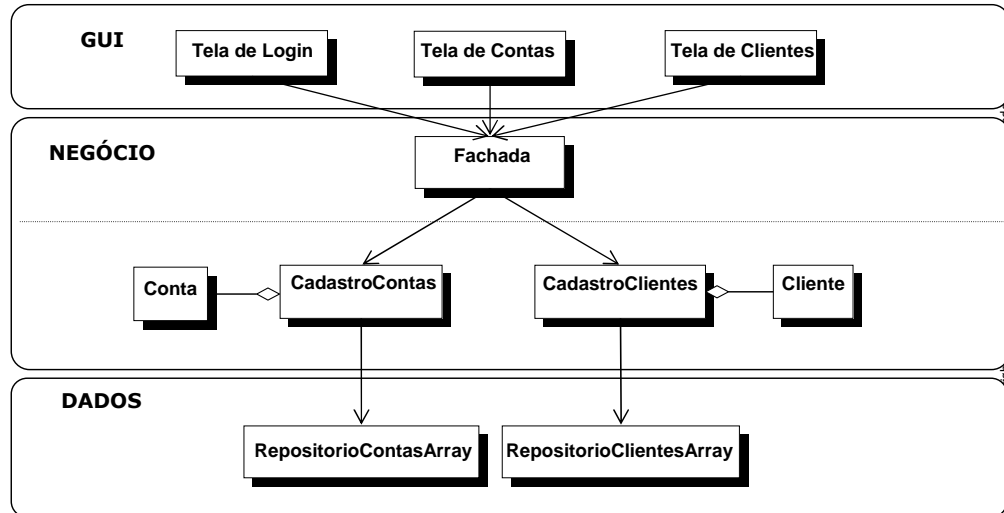


As classes desta camada são responsáveis pela manipulação da estrutura física de armazenamento dos dados e pela integração com sistemas externos. São elas que isolam o resto do sistema do meio de armazenamento usado (memória, arquivos simples, SGBD relacional, etc.), de maneira que se o meio de armazenamento for trocado, apenas as classes desta camada terão que ser modificadas ou substituídas.

A camada de negócio utiliza os serviços desta camada, para inserção, remoção, consulta e atualização das instâncias das classes básicas, através das **interfaces negócio-dados** (estas serão vistas no módulo de interfaces), que são implementadas pelas **coleções de dados**.

As coleções de dados dependem do meio de armazenamento utilizado. Porém, seus serviços são implementados de acordo com uma interface comum, chamada **interface negócio-dados**. As coleções de negócio na verdade referenciam objetos do tipo dessa interface, assim, as coleções de dados podem ser alteradas ou substituídas sem afetar as coleções de negócio que usam seus serviços.

Visão geral da arquitetura



A figura ilustra o exemplo da arquitetura em camadas e as classes participantes de cada uma. Neste exemplo a interface negócio-dados não é apresentada, pelo fato do assunto Interfaces ser abordado em módulos posteriores a este.

A GUI acessa os serviços fornecidos pelo sistema através da Fachada, a qual faz acesso às coleções de negócio. As coleções de negócio manipulam coleções de classes básicas, as quais são persistidas a partir das coleções de dados.

Exemplos de código (sem interface)

Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes



Coleção de Negócio

```
public class CadastroContas {  
    private RepositorioContasArray contas;  
    public CadastroContas(RepositorioContasArray r) {  
        this.contas = r;  
    }  
    public void atualizar(Conta c) {  
        contas.atualizar(c);  
    }  
    public void cadastrar(Conta c){  
        if (!contas.existe(c.getNumero())) {  
            contas.inserir(c);  
        } else {  
            System.out.println("Conta já cadastrada");  
        }  
    }  
}
```

Copyright © 2002 QualiTi. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 230



Coleção de Negócio

```
public void creditar(String n, double v) {
    Conta c = contas.procurar(n);
    c.creditar(v);
    contas.atualizar(c);
}
public void debitar(String n, double v) {
    Conta c = contas.procurar(n);
    c.debitar(v);
    contas.atualizar(c);
}
public void descadastrar(String n) {
    contas.remover(n);
}
```

Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 231



Coleção de Negócio

```
public Conta procurar(String n) {  
    return contas.procurar(n);  
}  
  
public void transferir(String origem,  
                        String destino,  
                        double val) {  
  
    Conta o = contas.procurar(origem);  
    Conta d = contas.procurar(destino);  
    o.transferir(d, val);  
    contas.atualizar(o);  
    contas.atualizar(d);  
  
}  
}
```

Qualiti Software Processes
Java Básico | 232



Copyright © 2002 Qualiti. Todos os direitos reservados.

Fachada

```
public class Fachada {
    private static Fachada instancia;
    private CadastroContas contas;
    private CadastroClientes clientes;

    public static Fachada obterInstancia() {
        if (instancia == null) {
            instancia = new Fachada();
        }
        return instancia;
    }
    private Fachada() {
        initCadastros();
    }
}
```

Copyright © 2002 QualiTi. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 233



Fachada

```
private void initCadastros() {  
    RepositorioContasArray rep =  
        new RepositorioContasArray();  
    contas = new CadastroContas(rep);  
  
    RepositorioClientesArray repClientes =  
        new RepositorioClientesArray();  
    clientes = new CadastroClientes(repClientes);  
}
```

Copyright © 2002 QualiTi. Todos os direitos reservados.



Fachada

```
//metodos para manipular clientes
public void atualizar (Cliente c) {
    clientes.atualizar(c);
}
public Cliente procurarCliente(String cpf) {
    return clientes.procurar(cpf);
}
public void cadastrar(Cliente c) {
    clientes.cadastrar(c);
}
public void descadastrarCliente(String cpf) {
    clientes.remover(cpf);
}
```

Copyright © 2002 Qualiti. Todos os direitos reservados.

Fachada

```
//metodos para manipular contas
public void atualizar (Conta c) {
    contas.atualizar(c);
}
public Conta procurarConta(String n) {
    return contas.procurar(n);
}
public void cadastrar(Conta c) {
    Cliente cli = c.getClient();
    if (cli != null) {
        clientes.procurar(cli.getCpf());
        contas.cadastrar(c);
    } else {
        System.out.println("cliente nulo");
    }
}
```

Qualiti Software Processes
Java Básico | 236



Copyright © 2002 Qualiti. Todos os direitos reservados.

Fachada

```
public void removerConta(String n) {
    contas.remover(n);
}
public void creditar(String n, double v) {
    contas.creditar(n, v);
}
public void debitar(String n, double v) {
    contas.debitar(n, v);
}
public void transferir(String origem,
    String destino, double val) {
    contas.transferir(origem, destino, val);
}
}
```

Referências do módulo

- ▶ Java Tutorial
 - <http://java.sun.com/docs/books/tutorial/>
- ▶ Documentação da Versão 6 da linguagem
 - <http://java.sun.com/javase/6/docs/>
- ▶ Livro Design Patterns, Gamma et. all. 2002

Copyright © 2002 Qualiti. Todos os direitos reservados.



Módulo 9

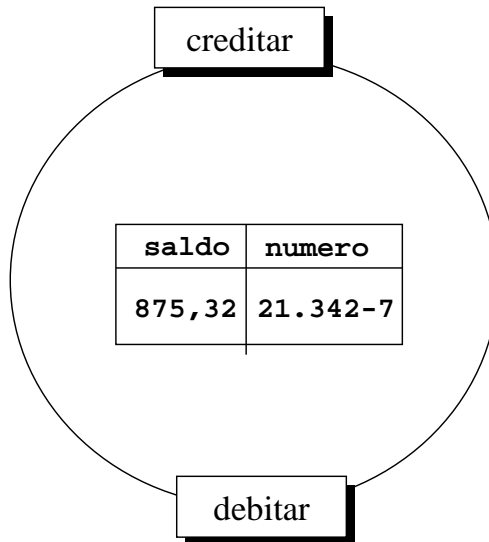
Classes Abstratas e Interfaces

Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes



Objeto Conta Imposto

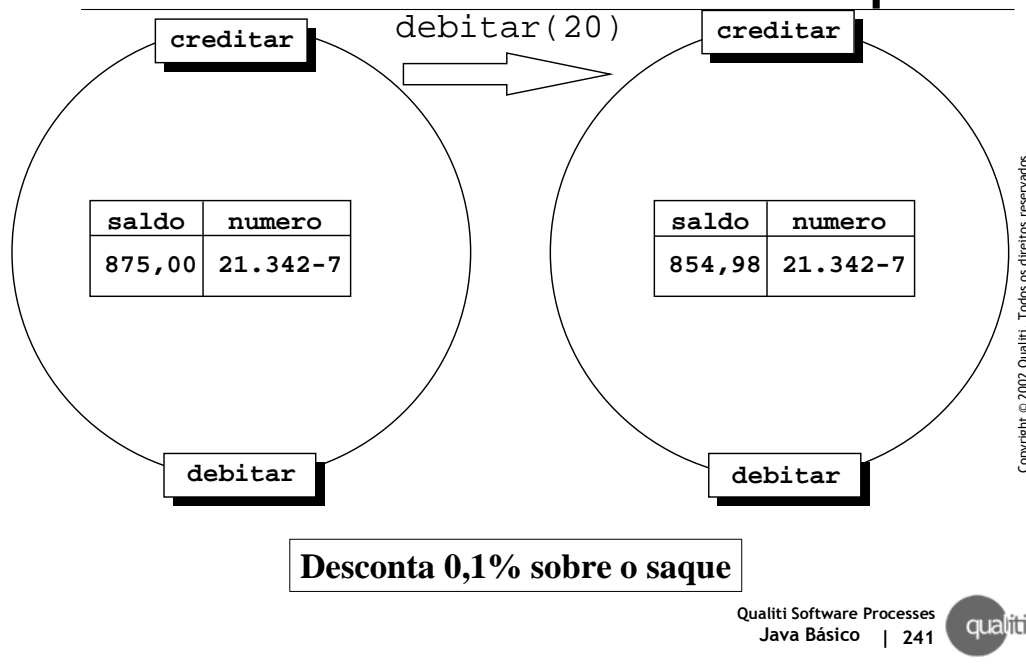


Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 240



Estados do Objeto Conta Imposto



Classe ContaImposto

```
public class ContaImposto extends Conta {  
    public static final double TAXA = 0.001; //0,1%  
  
    public ContaImposto (String n,Cliente c) {  
        super (n,c);  
    }  
  
    public void debitar(double valor){  
        double imposto = valor * TAXA;  
        super.debitar(valor + imposto)  
    }  
}
```

Copyright © 2002 QualiTi. Todos os direitos reservados.



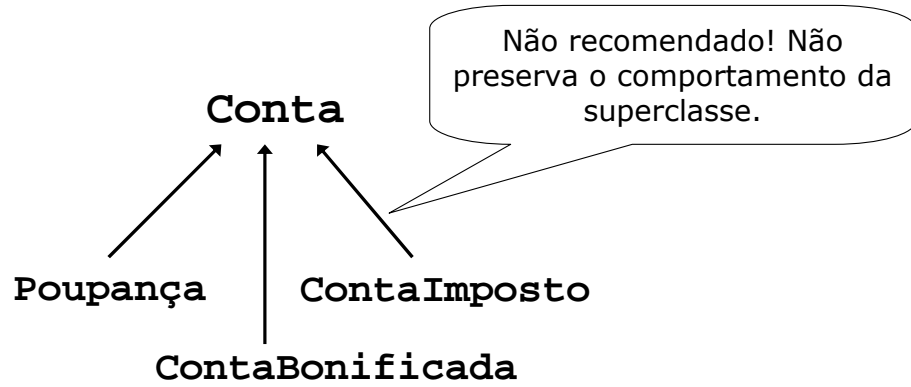
Subclasses e Comportamento

- ▶ Objetos da subclasse devem se comportar como os objetos da superclasse
- ▶ Redefinições de métodos devem preservar o comportamento (semântica) do método original
- ▶ Grande impacto sobre manutenção e evolução de software

Copyright © 2002 Qualiti. Todos os direitos reservados.

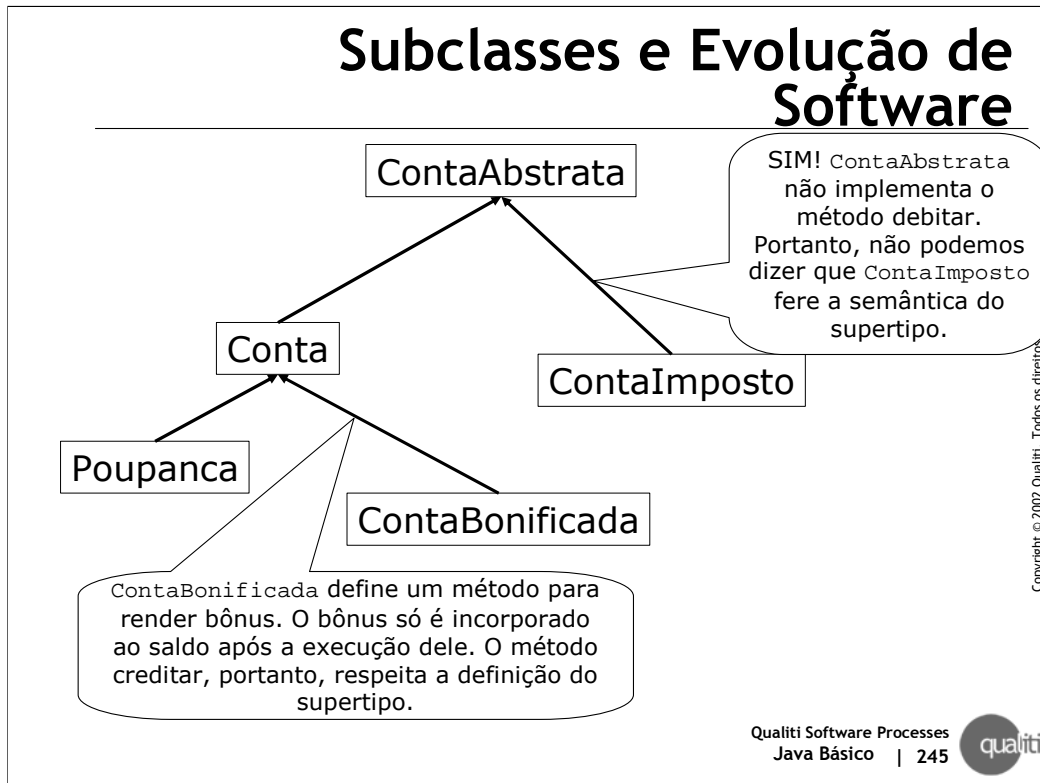


Subclasses e Evolução de Software



Copyright © 2002 Qualiti. Todos os direitos reservados.

Subclasses e Evolução de Software



Uma **classe abstrata** é dita ser uma **superclasse abstrata**, uma vez que é utilizada como um *Framework* para a construção de outras classes. Ou seja, uma classe abstrata possui membros que serão reutilizados para a criação de um novo tipo de classe, a qual deve fazer parte da mesma hierarquia de herança que a classe abstrata.

Classes abstratas não têm significado se isoladas, uma vez que têm o propósito somente de servir como base para criação de outras classes que possuem aspectos em comum. Por este motivo, não é possível criar instâncias de objetos de uma classe abstrata. Desta forma, classes abstratas:

- podem conter membros que são úteis em uma hierarquia de classes relacionadas por herança, mas são tão genéricas que acabam não tendo significado se isoladas destas classes. Por este motivo, uma classe abstrata não pode ter instâncias de sua classe.
- podem ser úteis para reuso de atributos, métodos (mesma implementação para as diferentes subclasses) e definição de métodos (os quais possuem diferentes implementações nas suas subclasses) comuns às suas subclasses.
- proporcionam maior poder ao programador para vetar a criação de objetos desnecessários.

Classes Abstratas

- ▷ Devem ser declaradas com a palavra-chave *abstract*
- ▷ Podem declarar métodos abstratos
 - Métodos sem implementação
 - Implementação fornecida na subclasse
- ▷ Podem declarar métodos concretos
 - Métodos com implementação

Copyright © 2002 QualiTi. Todos os direitos reservados.

Para exemplificar o uso de classes abstratas, suponha uma superclasse *FormaGeometrica* e suas subclasses *Retângulo* e *Losango*.

A superclasse *FormaGeometrica* pode conter atributos (lado e diagonal, por exemplo) e métodos comuns às suas subclasses (*getLado()*, *setLado()*, por exemplo). Além disso, todas as subclasses devem ter uma operação *calcularArea()*, cuja assinatura é a mesma utilizada em todas as subclasses, mas a implementação é diferente, dependendo da figura sendo calculada.

Neste cenário, a superclasse *FormaGeometrica* é uma forte candidata a ser transformada em uma classe abstrata, uma vez que:

- A superclasse possui métodos concretos (com implementação) e atributos comuns à todas as subclasses.
- A assinatura do método *calcularArea()* é a mesma, mas o modo como cada um deve ser implementado vai variar em cada subclasse. Desta forma, a definição do mesmo (método abstrato - sem implementação) pode ser descrita na superclasse, de modo que cada subclasse redefina este método de acordo com suas necessidades.
- A superclasse *FormaGeometrica*, do jeito que está definida, não tem sentido e nem é utilidade sozinha, portanto não faz sentido permitir a criação de instâncias desta.

Definição de Classes Abstratas

- ▶ Uma classe abstrata é declarada com o modificador **abstract**
- ▶ Um método é definido abstrato também usando o modificador **abstract**

```
public abstract class Nome_da_Classe {  
    atributo1;  
    atributo2;  
    ...  
  
    public void metodo1(){  
        // código do método 1  
    }  
    ...  
    public abstract void metodoAbstrato();  
}
```

Copyright © 2002 Qualiti. Todos os direitos reservados.

sses
247



Para realizar a declaração de uma classe abstrata em Java é necessário fazer uso do modificador **abstract** na definição da classe. Por exemplo:

```
public abstract class FiguraGeometrica{  
    ...  
}
```

Além de atributos, uma classe abstrata pode conter métodos concretos, ou seja, métodos contendo código; e pode também conter métodos abstratos, os quais são definidos somente com a assinatura, seguida de um ; (ponto-e-vírgula). Além disso, para que um método ser declarado abstrato, é necessário também fazer uso do modificador de acesso **abstract** em sua assinatura, como no exemplo a seguir :

```
...  
    public abstract calcularArea();  
...
```

Ou seja, método abstratos não possuem implementação. A implementação desses métodos é fornecida nas subclasses da mesma hierarquia de herança que a classe abstrata FiguraGeometrica, no exemplo citado.

Definindo uma Conta abstrata

```
public abstract class ContaAbstrata {  
    private String numero;  
    private double saldo;  
    private Cliente cliente;  
  
    public ContaAbstrata(String num, Cliente c) {  
        numero = num;  
        cliente = c;  
    }  
    public ContaAbstrata(String num, double s, Cliente c) {  
        numero = num;  
        saldo = s;  
        cliente = c;  
    }  
}
```

Copyright © 2002 QualiTi. Todos os direitos reservados.

No exemplo, a classe `ContaAbstrata` deve agregar os atributos e métodos que são comuns às classes `Conta` e `ContaImposto`, e definir os métodos que possuem a mesma assinatura, mas implementações diferentes em ambos (métodos abstratos).

Uma classe abstrata nunca é instanciada, mas pode definir construtores para inicialização dos atributos, permitindo também o reuso de construtores pelas subclasses.

Definindo uma Conta abstrata (cont...)

```
public Cliente getCliente(){
    return cliente;
}
public String getNumero() {
    return numero;
}
public double getSaldo() {
    return saldo;
}
public void setCliente(Cliente cliente) {
    this.cliente = cliente;
}
public void setNumero(String num) {
    this.numero = num;
}
public void setSaldo(double valor) {
    saldo = valor;
}
```

Copyright © 2002 QualiTi. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 249



Os métodos get e set para os atributos de ContaAbstrata, no exemplo citado, possuem a mesma implementação em qualquer subclasse. Por este motivo, eles continuam descritos como método concretos, para reuso pelas subclasses Conta e ContaImposto.

Definindo uma Conta abstrata (cont...)

```
public void creditar(double valor){
    saldo = saldo + valor;
}
public abstract void debitar(double valor) ;

public void transferir(ContaAbstrata c, double v){

    this.debitar(v);
    c.creditar(v);
}
}
```

Copyright © 2002 QualiTi. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 250



Os métodos `creditar()` e `transferir()` também possuem a mesma implementação nas subclasses `Conta` e `ContaImposto`, por este motivo, eles também podem ser declarados como métodos concretos na superclasse.

O método `debitar` de `Conta` possui um comportamento diferente do mesmo método relativo à `ContaImposto`. O processamento executado neste método para as duas classes é diferente. Por este motivo, o código do método `creditar` (tradicional) não poderá ser reutilizado. No entanto a definição (interface) do método pode ser reutilizada.

Neste caso, o método `debitar` é declarado como sendo abstrato.

Modificando a classe Conta

```
public class Conta extends ContaAbstrata {

    public Conta(String n, Cliente c) {
        super (n,c);
    }

    public void debitar(double valor){
        double saldo = getSaldo();
        if(valor <= saldo){
            setSaldo(saldo - valor);
        } else {
            System.out.println("Saldo insuficiente");
        }
    }
}
```

Copyright © 2002 QualiTi. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 251



A classe `Conta` pode herdar propriedades e métodos da Classe `ContaAbstrata`. Neste caso, a classe `Conta` deve estender a classe `ContaAbstrata` utilizando a palavra chave ***extends*** e redefinir o método `debitar`.

Além disso, o construtor de `Conta` pode fazer reuso também do construtor de `ContaAbstrata` através da palavra chave ***super***. Ou seja, as declarações para inicialização dos atributos não precisam ser descritas no construtor de `Conta` (a menos que processamentos ou inicializações adicionais precisem ser realizados), uma vez que o mesmo faz reuso da inicialização já realizada no construtor da superclasse `ContaAbstrata`.

Quando a palavra chave ***super*** é utilizada dentro de um construtor referenciando o construtor da superclasse, ela deve possuir os mesmos parâmetros que o construtor chamado. Por exemplo:

// Construtor da superclasse

```
public ContaAbstrata(String num, Cliente c) {
    numero = num;
    cliente = c;
}
```

// Construtor da subclasse Conta fazendo reuso do construtor da superclasse ContaAbstrata

```
public Conta(String numero, Cliente cliente){
    super(numero, cliente);
}
```

Modificando a classe ContaImposto

```
public class ContaImposto extends ContaAbstrata {
    public static final double TAXA = 0.001;

    public ContaImposto(String n, Cliente c) {
        super (n,c);
    }

    public void debitar(double valor) {
        double imposto = valor * TAXA;
        double saldo = this.getSaldo();
        if (valor + imposto <= saldo) {
            setSaldo(saldo - (valor + imposto));
        } else {
            System.out.println("Saldo Insuficiente");
        }
    }
}
```

Copyright © 2002 QualiTi. Todos os direitos reservados.



Os mesmos aspectos observados em Conta, também devem ser feitos aqui, ou seja, o método debitar também é redefinido na classe ContaImposto.

Classes abstratas: propriedades

- ▶ Uma classe abstrata não pode ser instanciada
- ▶ Mesmo sem poder ser instanciada, pode definir construtores para permitir reuso
- ▶ Qualquer classe com um ou mais métodos abstratos é automaticamente uma classe abstrata
- ▶ Se uma classe herdar de uma classe abstrata, deve redefinir todos os métodos abstratos. Caso contrário, deve ser declarada como abstrata.

Copyright © 2002 QualiTi. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 253



O uso de **classes abstratas** em Java inclui algumas restrições e controle que devem ser seguidos:

- Uma classe abstrata não pode ser instanciada, uma vez que a mesma deve ser definida como o intuito de servir como molde para a definição de outras classes. Ela deve ser genérica o suficiente para permitir reuso de código, propriedades e interface (método abstrato). O modificador ***abstract*** é utilizado exatamente como este intuito: impedir que classes declaradas abstratas sejam instanciadas.
- Apesar de não serem instanciadas, classes abstratas podem definir construtores para a inicialização dos atributos que são comuns à todas as suas subclasses, de modo que subclasses de uma classe abstrata possam reutilizar estas definições (inicializações).
- Quando uma classe define, pelo menos, um método abstrato, a mesma já é considerada uma classe abstrata. O compilador força que a mesma seja declarada com o modificador ***abstract***. Ou seja, se uma classe que contém um método abstrato não for declarada ***abstract***, o compilador gera um erro.
- Se uma classe herdar, através da palavra chave ***extends***, de uma classe abstrata, a mesma deve obrigatoriamente redefinir (fornecer código) os métodos abstratos declarados na superclasse. Do contrário, a subclasse deve também ser declarada abstrata.

Classes abstratas: propriedades

- ▷ Métodos **private**, **static** e **final** não podem ser abstratos.
 - Métodos declarados com estes modificadores não podem ser redefinidos
- ▷ Uma classe **final** não pode conter métodos abstratos
 - Métodos definidos em classes declaradas com este modificador não podem ser redefinidos
- ▷ Uma classe pode ser declarada como **abstract**, mesmo se não tiver métodos abstratos
 - Permite somente o reuso, mas não permite instâncias dessa classe

Copyright © 2002 QualiTi. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 254



- Uma classe que contém somente métodos concretos pode ser declarada abstrata. Neste caso, a mesma não poderá ser instanciada.

- Métodos abstratos devem ser redefinidos nas subclasses. Desta forma, um método abstrato não pode ser definido com o modificador **final**. Isto é óbvio, uma vez que se um método é declarado **final**, ele não pode ser redefinido em subclasses.

- Métodos declarados com o modificador **private** só são visíveis e manipulados por objetos da própria classe onde são declarados. Isto quer dizer que, em uma hierarquia de herança, os mesmos não são visíveis em uma subclasse. Dessa forma, um método abstrato não pode ser definido com o modificador **private**, uma vez que o mesmo deve ser redefinido nas subclasses que o implementam.

- Métodos abstratos também não podem ser declarados com o modificador **static**. Métodos declarados com este modificador podem ser chamados a partir da própria classe, sem a necessidade da criação de instâncias desta. Como um método abstrato é declarado sem corpo (sem código), se o mesmo pudesse ser declarado **static**, isto poderia gerar uma falha durante a execução, uma vez que não haveria nenhuma código para ser processado.

Classes abstratas: polimorfismo

```
public static void main(String args[]){
    ...
    ContaAbstrata ca1, ca2;
    ca1 = new ContaBonificada("21.342-7");
    ca2 = new ContaImposto("21.987-8");

    ca1.debitar(500);
    ca2.debitar(500);

    System.out.println(ca1.getSaldo());
    System.out.println(ca2.getSaldo());
    ...
}
```

Copyright © 2002 QualiTi. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 255



O uso de classes abstratas, na realidade, define uma interface comum para os vários membros em uma hierarquia de classes. A classe abstrata contém métodos que serão definidos nas subclasses. Dessa forma, todas as classes da mesma hierarquia podem fazer uso desta interface por meio de polimorfismo.

Polimorfismo é o nome dado ao processo de enviar uma mesma mensagem para uma variedade de objetos e a mesma poder assumir muitas diferentes formas. Isto possibilita, por exemplo, que objetos da subclasse possam ser criados e atribuídos à objetos da superclasse, de modo que a mesma será utilizada como uma interface. Neste caso, quando da chamada de métodos sobre esta interface, em tempo de execução será definido qual método será executado e, por conseguinte, quais diferentes ações podem ocorrer dependendo do tipo de objeto que recebe a chamada de método.

Classes abstratas: utilização

- ▶ Ajudam a estruturar sistemas definindo hierarquias consistentes de classes.
- ▶ Simplificam o reuso de código
- ▶ Definem "contratos" a serem realizados por subclasses
- ▶ Tornam o polimorfismo mais claro

Copyright © 2002 QualiTi. Todos os direitos reservados.

Com a utilização de classes abstratas é possível promover reuso de código, dados e interfaces entre classes envolvidas em uma mesma hierarquia de herança. Esta abordagem além de simplificar o reuso, ajuda a estruturar sistemas definindo hierarquias de classe mais consistentes, uma vez que define contratos bem estabelecidos que devem ser realizados pelas subclasses que herdam de classes abstratas.

Além disso, novas classes (novos tipos de objetos com o mesmo relacionamento de herança) podem ser adicionadas ao sistema sem modificação na estrutura já existente.

Outro aspecto importante relacionado ao uso de classes abstratas é que o uso de polimorfismo é mais evidente uma vez que cada subclasse implementa métodos abstratos de formas diferentes, o que significa que quando uma chamada de método é realizada ela é, de fato, feita de forma polimórfica, onde o comportamento é definido em tempo de execução.

Interfaces

Copyright © 2002 Qualiti. Todos os direitos reservados.

Auditor de Banco de Investimentos

```
class AuditorBI {
    final static double MINIMO = 500.00;
    private String nome;
    /* ... */

    public boolean investigaBanco(BancoInvest b) {
        double sm;
        sm = b.saldoTotal()/b.numContas();
        return (sm > MINIMO);
    }
}
```

Copyright © 2002 Qualiti. Todos os direitos reservados.

Para ilustrar a necessidade do uso de interfaces, é fornecido um exemplo de um programa que faz auditoria em diferentes tipos de bancos, por exemplo, Banco de Investimentos e Banco de Seguros. Este programa deve realizar uma série de procedimentos (operações) em cada banco para a realização da auditoria.

Cada Banco deve possuir, entre outros, métodos `saldoTotal()` e `numContas()`, os quais calculam o valor do saldo total, bem como o número total de contas existentes no banco. Estes cálculos são realizados de formas diferentes para cada tipo de banco. É importante ressaltar que, neste exemplo citado, embora Banco de Investimentos e Banco de Seguros possuam os mesmos métodos, eles não participam de uma mesma hierarquia de herança. Ou seja, são classes que não compartilham elementos entre si.

Entre as operações do programa Auditor existe uma chamada **investigaBanco**, a qual faz verificações com base no saldo total e numero de contas de cada tipo de banco. A operação `investigaBanco` recebe como parâmetro o banco sendo auditado.

No exemplo descrito, seria necessário a criação de dois programas diferentes para auditar dois tipos diferentes de banco: um `AuditoBI`, para auditar o banco de investimentos e outro `AuditorBS`, para o banco de seguros.

Auditor de Banco de Seguros

```
class AuditorBS {
    final static double MINIMO = 500.00;
    private String nome;
    /* ... */

    public boolean investigaBanco(BancoSeguros b)
    {
        double sm;
        sm = b.saldoTotal()/b.numContas();
        return (sm > MINIMO);
    }
}
```

Copyright © 2002 QualiTi. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 259



O método `investigaBanco` de cada programa Auditor deve calcular, a partir do tipo de banco passado como parâmetro, o valor resultante do saldo total pelo número de contas e em seguida realizar uma comparação a partir de uma constante. Este cálculo é realizado exatamente da mesma forma, para qualquer que seja o banco sendo passado como parâmetro. Isto implica que os diferentes programas para auditoria, teriam código duplicado.

Imagine se em vez de 2 tipos de bancos, fosse necessário realizar auditoria em N outros: seriam necessários N programas para auditoria, contendo o mesmo código, mudando somente o tipo de parâmetro sendo recebido.

Problema

- ▶ Duplicação desnecessária de código
- ▶ O mesmo auditor deveria ser capaz de investigar qualquer tipo de banco que possua operações para calcular
 - o número de contas no banco, e
 - o saldo total de todas as contas
- ▶ Casos em que as classes envolvidas não estão relacionadas através de uma hierarquia de herança

Copyright © 2002 Quality. Todos os direitos reservados.



O cenário apresentado ilustra uma situação de necessidade de reuso de código e definição de métodos entre classes que não estão relacionadas por uma hierarquia de herança.

Interfaces

- ▶ Caso especial de classes abstratas
- ▶ Definem tipo de forma abstrata, apenas indicando a **assinatura dos métodos** suportados pelos objetos do tipo
 - Os métodos declarados em uma interface não têm implementação
- ▶ Os métodos são implementados pelos subtipos (classes que implementam interfaces)
- ▶ Não têm construtores. Não se pode criar objetos já que métodos não estão implementados

Uma interface define um “contrato” a ser seguido

Para evitar o problema de duplicação de código do exemplo citado (Auditor de Bancos) é possível fazer uso do conceito de **Interfaces**.

Interfaces podem ser consideradas como sendo um tipo especial de classes abstratas. Ao contrário de classes abstratas, as quais são utilizadas para promover reuso de atributos, código e definições de métodos, bem como devem ser utilizadas como superclasses de subclasses que precisem de tais características, interfaces permitem reuso de definição de métodos. Ou seja, interfaces definem métodos abstratos que devem ser implementados por classes que não precisam estar relacionadas por uma hierarquia de herança.

Similarmente ao uso de classes abstratas, interfaces não podem ser instanciadas. Por este motivo e também pelo fato de em uma interface só poderem ser definidas assinaturas de métodos, as mesmas não podem declarar construtores em seu corpo.

Uma classe que implementa uma interface deve, obrigatoriamente, fornecer código para os métodos declarados na mesma. Do contrário a classe deve ser declarada abstrata. Isto demonstra que existe um contrato bem definido entre interfaces e as classes que as implementam.

Para o exemplo de Auditor de Bancos, em vez de criar um programa Auditor para cada tipo de Banco, poderia ser definida uma interface com as definições de métodos comuns a todos os bancos (no exemplo, `saldoTotal()` e `numcontas()`). Esta interface seria implementada por cada um dos tipos de banco, os quais forneceriam suas próprias implementações para os métodos declarados nesta. Desta forma, um único programa Auditor precisaria ser criado. Neste caso, em vez de receber um tipo de banco específico como parâmetro, o mesmo poderia receber a interface que o banco implementa. Assim, a chamada de método ao banco apropriado seria realizada polimorficamente, do mesmo jeito que acontece com métodos abstratos definidos em classes abstratas e, com isso, a duplicação de código poderia ser evitada.

Definição de Interfaces

```
interface Nome_Interface {  
  
    /*... assinaturas de novos métodos... */  
  
}
```

```
interface QualquerBanco {  
    double saldoTotal();  
    int numContas();  
}
```

Copyright © 2002 QualiTi. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 262



Uma interface é um tipo especial de classe Java e, como tal, deve ser declarada com uma sintaxe específica. A palavra chave **interface** permite a declaração de uma interface, por exemplo:

```
interface QualquerBanco {  
    /* Corpo da Interface*/  
}
```

A interface de nome QualquerBanco poderia ser declarada para conter as assinaturas dos métodos que são comuns aos bancos de investimentos e seguros: `saldoTotal()` e `numContas()`. Neste caso, a interface completa ficaria como no exemplo a seguir:

```
interface QualquerBanco {  
    double saldoTotal();  
    int numContas();  
}
```

Utilização de Interfaces

```
class Nome_classe implements Nome_Interface
{
    /*... Implementação dos métodos
        declarados na interface

        ...
    */
}
```

Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 263



Para que uma classe Java implemente uma interface, a mesma deve declarar a palavra chave **implements** e na sequência o nome da interface a ser implementada. Dentro desta classe, todos os métodos da interface devem ser implementados.

Se a classe que implementa uma interface também estender uma superclasse, a declaração de herança (**extends**) deve vir antes da declaração de implementação. Por exemplo:

```
class Nome_Classe extends Classe1 implements Nome_Interface {
    /* corpo do método */
}
```

Utilização de Interfaces

```
class BancoInvest implements QualquerBanco {  
    ...  
    double saldoTotal(){  
        /* código específico para BancoInvest */  
    }  
    int numContas(){  
        /* código específico para BancoInvest */  
    }  
    ...  
}
```

Copyright © 2002 QualiTi. Todos os direitos reservados.



Voltando ao exemplo do programa Auditor de Bancos, a classe que representa o banco de investimentos BancoInvest poderia implementar a interface QualquerBanco e, conseqüentemente, fornecer o código necessário para a implementação dos métodos descritos nela.

Utilização de Interfaces

```
class BancoSeguros implements QualquerBanco {  
    ...  
    double saldoTotal(){  
        /* código específico para BancoSeguros */  
    }  
    int numContas(){  
        /* código específico para BancoSeguros */  
    }  
    ...  
}
```

Copyright © 2002 QualiTi. Todos os direitos reservados.

O mesmo poderia ser feito pela classe que representa o banco de seguros BancoSeguros.

Desta forma, tanto BancolInvest quanto BancoSeguros poderiam, ser referenciadas a partir da interface QualquerBanco e, com isso, um único programa Auditor poderia fazer uso dos dois tipos de banco através desta interface.

Auditor Genérico

```
class Auditor {
    final static double MINIMO = 500.00;
    private String nome;
    /* ... */

    boolean investigaBanco(QualquerBanco b) {
        double sm;
        sm = b.saldoTotal()/b.numContas();
        return (sm > MINIMO);
    }
}
```

Copyright © 2002 QualiTi. Todos os direitos reservados.

A classe que representa o programa Auditor manipularia a interface implementada pelos diferentes tipos de bancos em vez dos tipos específicos. Ou seja, o método `investigaBanco` receberia como parâmetro a interface `QualquerBanco` e as chamadas para os métodos de `BancoInvest` ou `BancoSeguros` seria realizada polimorficamente. Com isto, é possível observar também que interfaces possibilitam a abstração da implementação dada a um método, uma vez que elas servem como um mecanismo para publicação de serviços (operações) fornecidos por uma dada classe, de modo que o cliente (quem invoca um método da interface, no exemplo a classe Auditor) de uma interface não precisa saber detalhes de como o método está sendo implementado.

Usando do Auditor Genérico

```
QualquerBanco bi = new BancoInvest();
QualquerBanco bs = new BancoSeguros();
Auditor a = new Auditor();

/* ... */
boolean res1 = a.investigaBanco(bi);
boolean res2 = a.investigaBanco(bs);
/* ... */
```

Copyright © 2002 QualiTi. Todos os direitos reservados.



Um cliente qualquer utilizando o programa Auditor, poderia criar uma instância do tipo BancoInvest e atribuí-la à uma variável do tipo da interface QualquerBanco, uma vez que BancoInvest implementa a mesma. Do mesmo jeito, uma instância do tipo BancoSeguros poderia ser atribuída à uma variável do tipo QualquerBanco. Desta forma, o método investigaBanco da classe Auditor, poderia receber qualquer uma das instâncias, uma vez que ambas podem ser representadas da mesma forma (através da interface que implementam).

Tipos e Subtipos

- ▶ Classes e Interfaces são tipos
- ▶ Os termos **subtipo** e **supertipo** também são usados
 - supertipo : interface
 - subtipo: classe que implementa a interface ou interface que estende outra interface
- ▶ Interfaces utilizam o conceito de herança múltipla
 - Herança múltipla de assinatura

Copyright © 2002 QualiTi. Todos os direitos reservados.

Os termos Supertipo e Subtipo também podem ser utilizados no contexto de interfaces e classes que implementam interfaces. Neste contexto, uma interface pode ser considerada um supertipo, uma vez que esta deve ser implementada por alguma (s) classe (s). Um subtipo pode ser considerado uma classe que implementa uma interface ou mesmo uma interface que herda de uma superinterface.

O mecanismo de herança também é válido entre interfaces. Porém, ao contrário de classes que possuem herança simples (uma classe só pode herdar de uma única superclasse), interfaces podem utilizar herança múltipla. Ou seja uma interface pode herdar de uma ou mais superinterfaces. Neste caso, um interface pode herdar as assinaturas de métodos de suas superinterfaces.

Herança múltipla de assinatura

```
interface I extends I1, I2, ..., In {  
    /*... assinaturas de novos métodos... */  
}
```

Interfaces podem estender
várias interfaces

Copyright © 2002 QualiTi. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 269



Para declarar uma interface que estende uma ou mais **supertipos**, basta utilizar a palavra chave **extends** e na sequência separar os nomes das interfaces (supertipos) por “,” (vírgula), como no exemplo abaixo:

```
interface Nome_Interface extends SuperTipo1, SuperTipo2 {  
  
    /* assinaturas dos métodos */  
}
```

Múltiplos supertipos

- ▶ Classe que implementa uma interface deve definir os métodos da interface
 - classes concretas têm que implementar todos os métodos da interface
 - caso algum método da interface não seja implementado, a classe deve ser declarada `abstract`
- ▶ Apesar de não suportar herança múltipla de classes, uma classe pode implementar mais de uma interface (pode ter vários **supertipos**)

```
class XXX implements interface1, interface2 {  
    ...  
}
```

Qualiti Software Processes
Java Básico | 270



Copyright © 2002 Qualiti. Todos os direitos reservados.

Se uma classe que declara uma cláusula **implements** para uma ou mais interfaces não implementar em seu corpo os métodos definidos nesta, um erro de compilação é gerado informado que a mesma deve ser declarada abstrata, uma vez que não implementa os métodos definidos na (s) interface (s) que a mesma implementa.

Uma classe pode declarar diretamente em sua cláusula a implementação de uma ou mais interfaces, ou seja uma classe pode implementar mais de um supertipo. A declaração da mesma deve ter o seguinte formato:

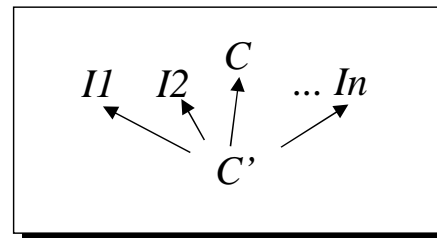
```
class Nome_classe implements SuperTipo1, Supertipo2,..., SupertipoN {  
  
    /* implementação do métodos declarados nas interfaces */  
  
}
```

Não existe um número fixo para interfaces que podem ser implementadas por uma classe, do mesmo modo não há um número fixo para o número de interfaces que um subtipo pode herdar.

Definição de classes: forma geral

```
class C'  
  extends C  
  implements I1, I2, ..., In {  
    /* ... */  
}
```

A classe C' é **subtipo** de $C, I1, I2, \dots, In$



Qualiti Software Processes
Java Básico | 271



Copyright © 2002 Qualiti. Todos os direitos reservados.

Para resumir a definição de interfaces, classes e seus conceitos relacionados à subtipos e supertipos, o seguinte cenário é exposto: Para classes C e C' , C' sendo uma subclasse de C ; interfaces $I1, I2, \dots, IN$, onde C' implementa estas interfaces. É possível afirmar que:

- C' é subclasse de C ;
- C é supertipo de C' ;
- $I1, I2, \dots, IN$ são supertipos de C' ; e
- C' é um subtipo de $I1, I2, \dots, IN$.

Interfaces e métodos

- ▷ Todos os métodos de uma interface são implicitamente **abstratos e públicos**
- ▷ métodos de interfaces não podem ser :
 - `static`
 - `final`
 - `private`
 - `protected`

Copyright © 2002 Qualiiti. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 272



Todas as assinaturas de métodos definidas em uma interface são públicas e abstratas por *default*. Isto implica que não é necessário acrescentar à assinatura de um método o modificador **abstract** e nem mesmo **public** a menos que você deseja enfatizar (visualmente) este aspecto.

Outras características da declaração de métodos em interfaces diz respeito ao uso dos demais modificadores Java:

- **static**: métodos declarados em uma interface não podem ser definidos com o modificador *static*, uma vez que os mesmos não têm código para execução.
- **final**: as classes que implementam as interfaces, devem redefinir os métodos da interface fornecendo código para os mesmos. Métodos declarados *final* não podem ser redefinidos.
- **private**: interfaces têm como objetivo tornar público determinados tipos de serviços (métodos), abstraindo a forma como os mesmos são implementados. Por este motivo todos os métodos declarados em uma interface são públicos por *default*. Além disso, métodos públicos não podem ser redefinidos com visibilidade mais restritiva.
- **protected**: idem

Interfaces e atributos

- ▶ Interfaces não podem conter atributos
- ▶ A única exceção são os atributos **static final** (constantes).
- ▶ Antes do suporte dado a tipos enumeráveis a Interfaces eram usadas como repositórios para constantes. Hoje essa prática é desaconselhada.

```
public interface CoresBasicas {  
    public static final int AZUL = 0;  
    public static final int VERMELHO = 1;  
    public static final int VERDE = 2;  
    public static final int PRETO = 3;  
    public static final int AMARELO = 4;  
}
```

```
janela.alterarCor(CoresBasicas.AZUL);
```

Qualiti Software Processes
Java Básico | 273



Copyright © 2002 Qualiti. Todos os direitos reservados.

Além de assinaturas de métodos, interfaces podem também declarar atributos, CONTANTO que estes sejam declarados como constantes, ou seja, sejam definidos com os modificadores *static* e *final*.

Constantes declaradas em interfaces podem ser reutilizadas nas classes que implementam tais interfaces e mesmo por interfaces que herdam de supertipos.

Interfaces e Reusabilidade

- ▶ Evita duplicação de código através da definição de um tipo genérico, tendo como subtipos várias classes não relacionadas
- ▶ Uma interface agrupa objetos de várias classes definidas independentemente, sem compartilhar código via herança, tendo implementações totalmente diferentes

Copyright © 2002 Qualiti. Todos os direitos reservados.



Subtipos e instanceof

```
class Circulo extends Forma implements
    Selecao, Rotacao, Movimentacao {
    ...
}
```

```
...
Circulo c = new Circulo();
res1 = c instanceof Circulo;
res2 = c instanceof Selecao;
res3 = c instanceof Rotacao;
res4 = c instanceof Movimentacao;
...
```

res1, res2, res3 e res4 são **true**!

Qualiti Software Processes
Java Básico | 275



Copyright © 2002 Qualiti. Todos os direitos reservados.

Quando utilizando interfaces também é possível utilizar o operador **instanceof** para identificação do tipo da instância de um dado objeto.

Neste caso, todo subtipo é uma instância do seu supertipo. Ou seja, uma classe que implementa uma interface é também uma instância de uma interface.

Classes abstratas X Interfaces

Classes (abstratas)

- ▶ Agrupa objetos com implementações compartilhadas
- ▶ Define novas classes através de herança (simples) de código
- ▶ Só uma classe pode ser supertipo de outra classe

Interfaces

- ▶ Agrupa objetos com implementações diferentes
- ▶ Define novas interfaces através de herança (múltipla) de assinaturas
- ▶ Várias interfaces podem ser supertipo do mesmo tipo

Copyright © 2002 QualiTi. Todos os direitos reservados.



Interfaces no padrão de camadas

- ▷ Publicam os serviços de uma camada específica
- ▷ Quebram dependência entre as camadas
 - Flexibilidade na implementação das camadas
- ▷ Fachada -regras do negócio
- ▷ Coleções de Negócio (Cadastros) - restrições específicas de um negócio
- ▷ Coleções de dados - restrições de estruturas de dados
 - Diversos tipos de estruturas de dados podem ser implementados nesta camada.

O objetivo principal é quebrar a dependência de implementação entre Coleções de Negócio e Coleções de Dados

Qualiti Software Processes
Java Básico | 277

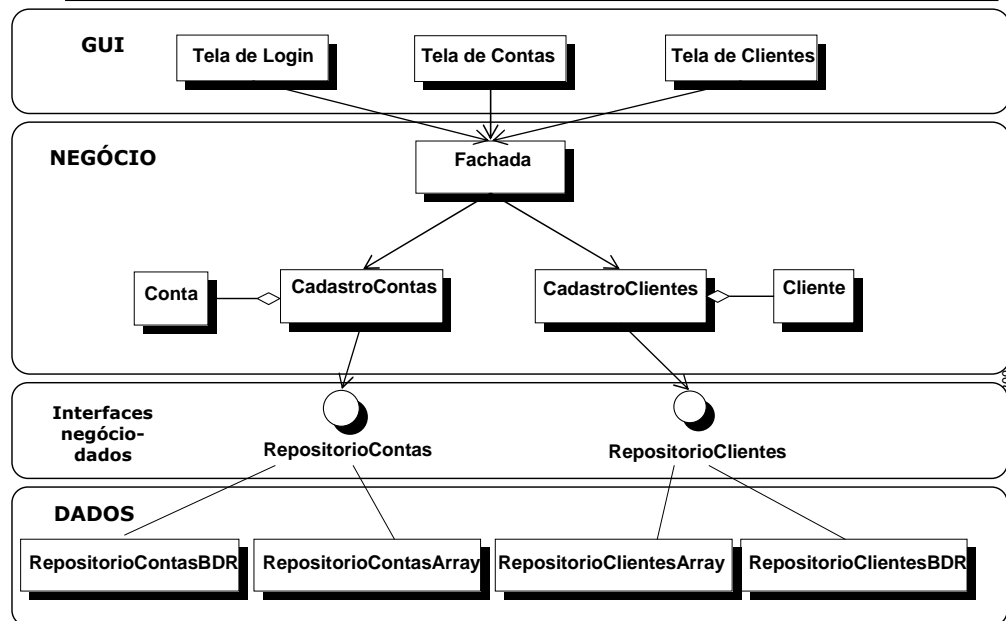


Copyright © 2002 Qualiti. Todos os direitos reservados.

Interfaces têm um papel muito importante na estruturação de aplicações utilizando o padrão de camadas. Como já descrito, além de permitirem reuso das assinaturas de métodos e atributos constantes, com interfaces é possível tornar transparente o código que está sendo implementado. Uma vez que um cliente de uma interface não tem acesso direto a este mas sim aos serviços que a mesma fornece.

No padrão de camadas a interface deve ser utilizada entre coleções de negócio e coleções de dados e, por este motivo, a mesma recebe a denominação de interface negócio-dados. Na realidade, uma coleção de negócio (Cadastro) só deve ter acesso a uma coleção de dados (Repositório) a partir desta interface. Esta interface deve ser implementada pela coleção de dados. Assim, é possível definir diferentes coleções de dados, por exemplo uma com código para manipulação em banco de dados (RepositorioBDR), outra para armazenamento em arquivos (RepositorioFile) ou a persistência sendo realizada em memória (RepositorioArray) e assim por diante, todas implementando a mesma interface.

Visão geral da arquitetura



O fato é que, independente do mecanismo de persistência sendo utilizado, é possível construir uma interface única, com as assinaturas dos métodos que devem ser implementados em cada mecanismo de persistência e, o que é melhor, a coleção de dados não possui qualquer informação sobre o mecanismo de armazenamento sendo utilizado, uma vez que a mesma faz acesso a este a partir de uma interface. Com isto, se o mecanismo de persistência for modificado, por exemplo de array para banco de dados relacional, basta criar uma outra coleção de dados que implementa a mesma interface negócio-dados já utilizada. A coleção de negócio se mantém inalterada. Ou seja, as classes de negócio do sistema não precisam ser alteradas se o mecanismo para armazenamento for modificado. Isto garante maior extensibilidade ao sistema, além de torná-lo mais modular.

Interfaces no padrão de camadas

```
public interface RepositorioContas {  
  
    public void inserir(Conta c);  
    public boolean existe(String num);  
    public void atualizar(Conta c);  
    public Conta procurar(String num);  
    public void remover(String num);  
}  
  
public interface RepositorioClientes {  
    public void atualizar(Cliente c);  
    public boolean existe(String cpf);  
    public void inserir(Cliente c);  
    public Cliente procurar(String cpf);  
    public void remover(String cpf);  
}
```

Copyright © 2002 QualiTi. Todos os direitos reservados.



Interfaces no padrão de camadas

```
public class CadastroContas {  
    private RepositorioContas contas;  
  
    public CadastroContas(RepositorioContas r) {  
        this.contas = r;  
    }  
    ...  
}  
  
public class CadastroClientes {  
    private RepositorioClientes clientes;  
  
    public CadastroClientes(RepositorioClientes l) {  
        this.clientes = l;  
    }  
    ...  
}
```

Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 280



Interfaces no padrão de camadas

```
public class RepositorioContasArray
    implements RepositorioContas {
    private Conta[] contas;
    private int indice;
    private final static int tamCache = 100;
    ...
}

public class RepositorioClientesArray
    implements RepositorioClientes {
    private Cliente[] clientes;
    private int indice;
    private final static int tamCache = 100;
    ...
}
```

Copyright © 2002 Qualiti. Todos os direitos reservados.



Interfaces no padrão de camadas

```
private Fachada {  
    RepositorioContas repContas  
        = new RepositorioContasArray ();  
  
    cadastroContas  
        = new CadastroContas(repContas);  
  
    RepositorioClientes repClientes  
        = new RepositorioClientesArray();  
  
    cadastroClientes  
        = new CadastroClientes(repClientes);  
}
```

Copyright © 2002 Qualiti. Todos os direitos reservados.



Referências do módulo

- ▶ Java Tutorial
 - <http://java.sun.com/docs/books/tutorial/>
- ▶ Documentação da Versão 6 da linguagem
 - <http://java.sun.com/javase/6/docs/>

Copyright © 2002 Qualiti. Todos os direitos reservados.



Módulo 10

Exceções

Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes



Robustez

► Sistemas robustos devem:

- Fornecer formas eficientes para recuperação de falhas
- Fornecer mensagens apropriadas de erros
- Fazer validação dos dados
- Evitar que situações indesejadas ocorram
- Garantir a consistência das operações

Copyright © 2002 QualiTi. Todos os direitos reservados.

Quando se fala em sistemas robustos deseja-se que os mesmos sejam capazes de se comportar de forma previsível e consistente diante de situações que fogem ao contexto de execução normal do mesmo.

Por exemplo, um programa robusto é capaz de fornecer mensagens de erros apropriadas quando necessário, de modo que o usuário do mesmo seja informado sobre o ocorrido. Da mesma forma, um programa robusto deve prover mecanismos para validações dos dados fornecidos e disponibilização de mecanismos para recuperação de falhas, por exemplo, em uma aplicação que acessa um banco de dados, o sistema poderia checar de tempos em tempos se a conexão ainda está ativa antes de executar um acesso ao banco. Neste caso, se uma conexão com o banco de dados não existe mais, o sistema pode tentar conectar automaticamente, em vez de lançar uma exceção e enviar uma mensagem de erro pro usuário.

Outro aspecto que torna um programa robusto diz respeito ao fato de que o mesmo deve garantir a consistência das operações. Por exemplo, em uma transação bancária de transferência entre contas é desejável que um valor seja debitado de uma conta fonte e creditado em uma conta destino. Por exemplo, suponha que o saldo da conta fonte seja insuficiente para o debito deste valor na mesma. Neste caso, a operação de crédito na conta destino teria que se cancelada. Uma das formas de garantir essa consistência seria utilizando exceções.

Classe Contas: definição

```
class Conta {  
    private String numero;  
    private double saldo;  
    /* ... */  
    public void debitar(double valor) {  
        saldo = saldo - valor;  
    }  
}
```

Como evitar débitos acima do limite permitido?

Copyright © 2002 QualiTi. Todos os direitos reservados.

No exemplo de uma conta bancária, a operação debitar deve decrementar do saldo o valor solicitado, como ilustrado abaixo.

```
...  
public void debitar(double valor){  
    saldo = saldo - valor;  
}  
...
```

Se o valor solicitado fosse maior do que o valor disponível na Conta e a mesma não pudesse ficar com o saldo negativo, a operação acima não seria realizada de forma consistente, uma vez que não define nenhum mecanismo para prover esse aspecto.

Possíveis soluções

- ▶ Desconsiderar operação
- ▶ Mostrar mensagem de erro
- ▶ Retornar código de erro

Copyright © 2002 QualiTi. Todos os direitos reservados.



Algumas possíveis abordagens para o problema de consistência da operação de debitar poderia ser:

- Desconsiderar a operação, caso o valor a ser debitado fosse maior que o valor disponível na conta.
- Mostrar uma mensagem de erro, caso o valor a ser debitado fosse menor que o saldo disponível.
- Retornar um código de erro para o usuário da aplicação, indicando que houve um problema na mesma.

Desconsiderar Operação

```
class Conta {  
    private String numero;  
    private double saldo;  
    /* ... */  
    public void debitar(double valor) {  
        if (valor <= saldo)  
            saldo = saldo - valor;  
    }  
}
```

Copyright © 2002 QualiTi. Todos os direitos reservados.

Para a abordagem de desconsiderar a operação em caso de saldo menor que o valor solicitado, embora a operação não seja realizada, o usuário da aplicação não possui nenhuma resposta acerca da operação solicitada. Isto acaba fazendo com que a mesma não tenha o comportamento desejável para o usuário final da mesma.

Desconsiderar Operação

► Problemas:

- quem solicita a operação não tem como saber se ela foi realizada ou não
- nenhuma informação é dada ao usuário do sistema

Copyright © 2002 Qualiti. Todos os direitos reservados.

Mostrar Mensagem de Erro

```
class Conta {  
    static final String msgErro = "Saldo Insuficiente!";  
    private String numero;  
    private double saldo;  
    /* ... */  
    void debitar(double valor) {  
        if (valor <= saldo)  
            saldo = saldo - valor;  
        else System.out.print(msgErro);  
    }  
}
```

Copyright © 2002 Quali. Todos os direitos reservados.

O tratamento de problemas desse natureza com mensagens de erro, acaba gerando uma mistura de conceitos na aplicação entre código necessário para a manipulação do negócio da aplicação (àquilo que a operação se propõe a fazer de fato) e código que serve como apresentação para o usuário (interface gráfica). Ou seja, acaba havendo uma certa dependência entre o processamento que deve ser realizado e como o resultado deste processamento deve ser apresentado ao usuário da aplicação.

Mostrar Mensagem de Erro

► Problemas:

- O usuário do sistema recebe uma mensagem, mas nenhuma sinalização é fornecida para métodos que invocam debitar
- Há uma forte dependência entre a classe `Conta` e sua interface com o usuário

Não há uma separação clara entre código da camada de negócio e código da camada de interface com o usuário

Copyright © 2002 QualiTi. Todos os direitos reservados.

Retornar Código de Erro

```
class Conta {  
    private String numero;  
    private double saldo;  
    /* ... */  
    boolean debitar(double valor) {  
        boolean r = false;  
        if (valor <= saldo) {  
            saldo = saldo - valor; r = true;  
        } else r = false;  
        return r;  
    }  
}
```

Copyright © 2002 QualiTi. Todos os direitos reservados.

Com a utilização de códigos de erro para tratamento de problemas como o citado, onde o método debitar de contas possui o saldo menor que o valor solicitado, os clientes (o usuário da aplicação ou, mesmo outros métodos) do método debitar devem antes avaliar o resultado retornado para decidir o que deve ser feito.

Esta situação fica mais complicada ainda quando o método utilizado já retorna algum tipo, inerente à lógica da aplicação. Nesta situação, o valor retornado pode representar tanto o resultado da operação quanto um código de erro.

Código de Erro: Problemas

```
class CadastroContas {  
    /* ... */  
    int debitar(String n, double v) {  
        int r = 0;  
        Conta c = contas.procurar(n);  
        if (c != null) {  
            boolean b = c.debitar(v);  
            if (b) r = 0; // Ok!  
            else r = 2;  // saldo insuficiente  
        } else r = 1;   // conta inexistente  
        return r;  
    }  
}
```

Copyright © 2002 QualiTi. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 293



A classe CadastroContas é um exemplo de cliente utilizando o método debitar da classe Conta. O método debitar de CadastroContas faz acesso ao método debitar de Conta e antes de retornar o valor final da operação precisa testar o resultado do método debitar de Conta e atribuir os códigos de resposta necessários para retorno.

A classe que representar o cliente de CadastroContas também vai ter que testar o valor resultante para decidir o que fazer com a resposta. Ou seja, esse processo é repetido até o último cliente na hierarquia de acesso à classe Conta do exemplo. Isto denota a complexidade e trabalho envolvidos quando da utilização desta abordagem para tratamento de falhas, além da facilidade de inclusão de erros durante a avaliação.

Retornar Código de Erro

► Problemas:

- dificulta a definição e uso do método:
 - métodos que invocam **debitar** têm que testar o resultado retornado para decidir o que deve ser feito
 - A situação é pior em métodos que retornam códigos de erro.
- a dificuldade é ainda maior quando o método já retorna algum tipo:
 - o retorno pode ser o resultado da operação ou um código de erro.

O que fazer quando o tipo retornado não é primitivo?

Não bastasse a complexidade inerente à abordagem de utilização de códigos de erro para tratamento de falhas, problema maior surge quando o tipo de retorno de um método não é um tipo primitivo. Neste caso, a solução para retornar códigos de erro é totalmente inviabilizada.

Exceções (Definição)

- ▶ Exceções são o mecanismo utilizado por Java para tratamento de erros ou situações indesejadas
 - Erros de programação: Acesso a uma posição inválida de um array, divisão por zero, invocação de um método em uma referência nula.
 - Situações indesejadas: Uma conexão de rede indisponível durante uma comunicação remota, a ausência de um arquivo procurado localmente

Copyright © 2002 Quality. Todos os direitos reservados.

Java fornece mecanismos para tornar programas escritos nesta linguagem mais robustos, sem precisar passar pelos problemas gerados com as abordagens já citadas.

Tais mecanismos são chamados de **Exceções**. Com exceções é possível realizar tratamentos de erros de programação, por exemplo divisão por 0, acesso a uma posição inválida no array; ou situações indesejadas, por exemplo conexão inválida com banco de dados, entre outros.

Exceções

- ▶ Exceções são declaradas como classes Java
- ▶ Os objetos de uma exceção encapsulam informações relevantes sobre o erro ocorrido
- ▶ Exceções podem ser:
 - declaradas
 - lançadas
 - tratadas

Copyright © 2002 QualiTi. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 296

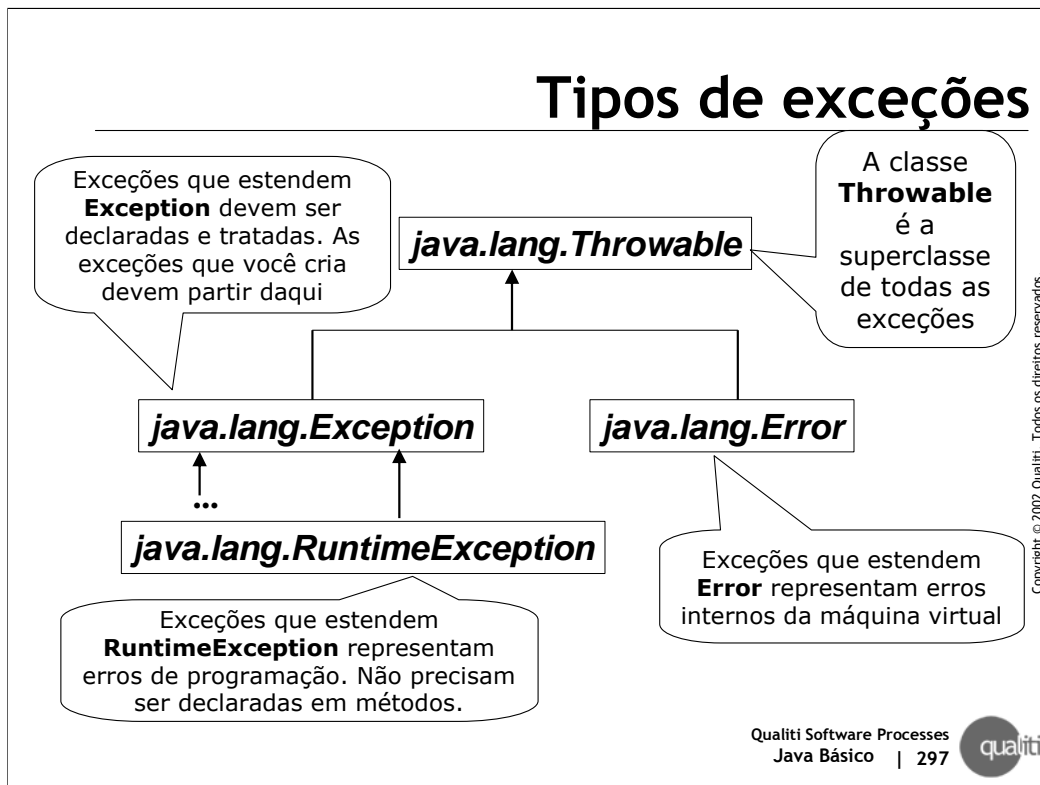


Uma exceção em Java, na realidade, corresponde a uma classe Java que utiliza uma declaração específica para descrever que a classe em questão se trata de uma exceção e não uma classe Java qualquer.

As informações sobre as falhas e erros ocorridos podem ser encapsuladas pelas exceções e manipuladas pela aplicação seguindo algumas regras específicas:

- Exceções podem ser declaradas, indicando que o processamento a ser realizado pode causar alguma falha ou problema;
- podem ser lançadas, dizendo explicitamente que tipo de exceção deve ser lançada quando um determinada situação de falha acontecer; e
- podem ser tratadas, denotando o tratamento dado quando o processamento de uma operação resulta em uma exceção.

Java fornece um conjunto de Exceções com propósitos específicos que podem ser utilizadas e reutilizadas para implementação de programas Java robustos.



Java fornece uma hierarquia de exceções para auxiliar na definição de exceções específicas para uma dada aplicação, bem como permitir o reuso de exceções já definidas para a manipulação de programas.

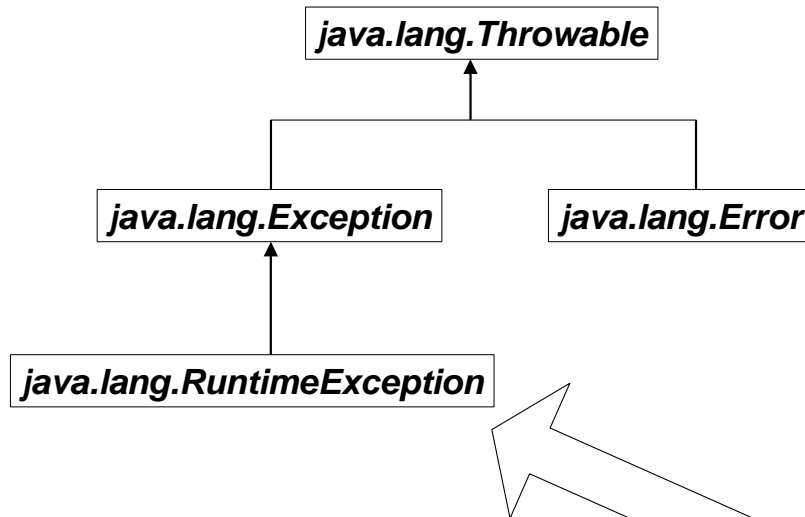
Toda exceção Java é uma subclasse de **Throwable**. Esta classe é a superclasse direta ou indireta de todas as exceções Java. Como subclasses diretas desta existem as classes **Exception** e **Error**. Além disso, existe ainda uma outra categoria de exceção que é **RuntimeException** a qual é uma subclasse direta de **Exception**, mas seu uso é destinado a um propósito diferente de sua superclasse **Exception**.

A classe **Exception** e suas subclasses (definidas pelo usuário), exceto **RuntimeException**, devem ser utilizadas para o tratamento de erros como o visto no exemplo de saldo insuficiente para débito na classe **Conta** ou problemas onde o programa está correto, mas a entrada do usuário pode gerar uma falha no sistema, ou referencia de objeto nula, conexão inativa com o banco de dados, objeto inexistente no banco de dados, entre outras.

A exceção **RuntimeException** e suas subclasses são lançadas automaticamente pela máquina virtual Java quando existe algum *bug* no programa que gera alguma falha, por exemplo acesso a uma posição inválida de um array.

A classe **Error** e suas subclasses descrevem problemas que não são comuns mas que quando de sua ocorrência podem tornar programas inconsistentes e difíceis de recuperar. Pode estar relacionado a um *bug* no programa, bem como problemas de ambiente de execução, por exemplo falta de memória.

Exceções não checadas (unchecked exceptions)



Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 298



As exceções do tipo *Error* são geradas pela máquina virtual Java e, por este motivo não precisam ser declaradas para tratamento em classes Java.

O mesmo acontece com exceções do tipo *RuntimeException* e suas subclasses. O programador não precisa especificar tratamento para esses tipos de exceções uma vez que a máquina virtual se encarrega e gerá-las e lançá-las em tempo de execução quando uma falha acontece. Por este motivo, exceções do tipo *RuntimeException* são denominadas *Unchecked Exceptions*.

Exceções não checadas (unchecked exceptions)

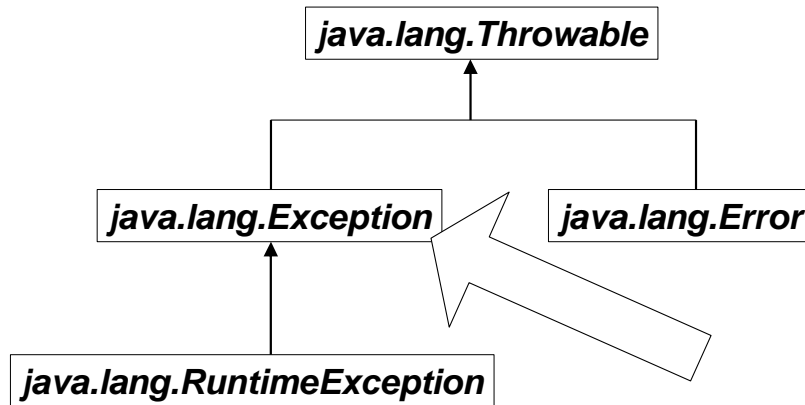
- ▶ Exceções que estendem `RuntimeException` indicam erros de programação e normalmente não são declaradas. Por exemplo, uma divisão por zero
- ▶ Qualquer método pode gerar essas exceções apesar de não explicitar isto na sua definição
- ▶ Tratar exceções deste tipo é como tentar corrigir um erro de programação durante a programação. Não faz muito sentido
- ▶ O que se faz é capturar a exceção e apresentar uma mensagem de erro agradável ao usuário indicando esta ocorrência

Copyright © 2002 Qualiti. Todos os direitos reservados.

Unchecked Exceptions não precisam ser declaradas em métodos uma vez que a máquina virtual se encarrega de gerá-las independente de terem ou não sido declaradas. Um bug de programa, por exemplo uma divisão por 0 em Java gera uma *ArithmeticException*, a qual é uma subclasse de *RuntimeException*.

Apesar de não precisar declarar exceções dessa natureza, é possível capturar essas exceções de modo a garantir que as mesmas possam ser reenviadas com uma mensagem de erro mais agradável para o usuário, uma vez que a mensagem de erro gerada pela máquina virtual é bem genérica, em inglês. Pode ser interessante capturar esse tipo de informação e personalizá-la para o usuário do programa.

Exceções checadas (checked exceptions)



Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 300



Um programador pode definir exceções específicas para a sua aplicação estendendo a classe *Exception*. Quando uma exceção do tipo *Exception* ou suas subclasses são definidas em um programa, o programador é responsável por indicar onde as mesmas devem ser declaradas (pontos de possíveis falhas no programa) e como devem ser tratadas.

Exceções dessa natureza não são geradas e lançadas automaticamente pela máquina virtual. O programador é responsável por indicar se o programa possui exceções e dizer como as mesmas devem ser manipuladas pela máquina virtual Java. Por este motivo, exceções do tipo *Exception* são chamadas de *Checked Exceptions*.

Como as *Checked Exceptions* são definidas pelo programador, para uma dada aplicação, as mesmas também são conhecidas como exceções de negócio.

Exceções checadas (checked exceptions)

- ▶ Exceções checadas são todas as exceções que são subclasses de **Exception**, exceto **RuntimeException** e suas subclasses
- ▶ Devem ser **declaradas e tratadas** no código

Copyright © 2002 Qualiti. Todos os direitos reservados.

Exceções

- ▶ Exceções podem ser definidas pelo programador e devem ser subclasses de **java.lang.Exception**
- ▶ Definem-se novas exceções para:
 - oferecer informações extra sobre o erro
 - distinguir os vários tipos de erro/situação indesejada
 - Específicas para uma dada aplicação (exceções de negócio)

Copyright © 2002 Quali. Todos os direitos reservados.



Formato da declaração de de exceções

```
class Nome_Excecao_Exception extends Exception {  
  
    public Nome_Excecao_Exception () {  
  
    }  
  
    /*...*/  
}
```

Por convenção, é aconselhável que o nome de qualquer exceção definida pelo programador tenha o sufixo **Exception**: **SaldoInsuficienteException**, **ObjetoInvalidoException**, etc

Copyright © 2002 QualiTi. Todos os direitos reservados.

Para que um programador defina uma exceção de negócio é necessário estender a classe *Exception* utilizando a palavra chave ***extends***.

Uma exceção de negócio é uma classe Java que estende a superclasse *Exception*. Por ser, em sua essência, uma classe Java normal, a mesma pode ser declarada com qualquer nome. Todavia, por convenção e para facilitar a identificação do tipo de classe representada, é aconselhável que uma exceção seja declarada com o sufixo **Exception**. Por exemplo, para o saldo insuficiente da operação debitar da classe Conta a exceção **SaldoInsuficienteException** poderia ser definida.

Exemplo de definição de Exceções

```
class SaldoInsuficienteException extends Exception {  
  
    public SaldoInsuficienteException() {  
        super("Saldo Insuficiente!");  
    }  
  
    /*...*/  
}
```

Copyright © 2002 QualiTi. Todos os direitos reservados.

O construtor de uma subclasse de *Exception* pode fazer uso do construtor de sua superclasse, do mesmo jeito que acontece com Superclasses e subclasses Java.

Exemplo de definição de Exceções

```
class SaldoInsuficienteException extends Exception {
    private double saldo;
    private String numero;

    public SaldoInsuficienteException(double saldo,
                                      String numero) {
        super("Saldo Insuficiente!");
        this.saldo = saldo;
        this.numero = numero;
    }

    public SaldoInsuficienteException() {
        super("Saldo Insuficiente!");
    }

    double getSaldo() {
        return saldo;
    }
    /*...*/
}
```

Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 305



Uma exceção pode ser definida de diferentes formas, com diferentes construtores e método auxiliares

Declaração e lançamento

- ▶ Exceções são declaradas na assinatura dos métodos, que devem tratar um dado processamento, usando o comando **throws**
- ▶ Exceções são lançadas no corpo dos métodos usando o comando **throw**

```
void metodo(...) throws Exc1, Exc2, ExcN {  
    ...  
    throw new Exc1(...);  
}
```

- ▶ Exceções lançadas no corpo de um método, e não tratadas, devem ser declaradas.

Quando o processamento de um método pode gerar falhas, uma ou mais exceções de negócio podem ser declaradas na assinatura do método, indicando que o mesmo pode gerar uma exceção. A declaração de exceções nas assinaturas de métodos devem ser realizadas utilizando a palavra chave **throws**. Por exemplo:

```
public void debitar (double valor) throws SaldInsuficienteException{  
    /* Corpo do método debitar */  
}
```

Quando uma exceção é declarada na assinatura de um método indica que o método em questão pode gerar a exceção ou o mesmo pode acessar um outro método que gera tal exceção. No primeiro caso, onde o método em questão é responsável por gerar inicialmente a exceção, a mesma deve ser declarada mas também lançada dentro do corpo do método para que os clientes deste possam tratar ou relançar a mesma para as camadas de cima (clientes em uma hierarquia de vários níveis). Este lançamento deve ser realizado utilizando a palavra chave **throw** dentro do corpo do método. Por exemplo:

```
public void debitar (double valor) throws SaldInsuficienteException{  
    if (valor <=saldo)  
        /* Corpo do método debitar */  
    else throw new SaldInsuficienteException ();  
}
```

Quando exceções são lançadas

- ▶ Exceções são lançadas quando:
 - um método que lança exceções é chamado
 - você detecta uma situação de erro e levanta uma exceção com **throw**
 - você comete um **erro de programação** como, por exemplo, tentar acessar uma posição inválida de um array: `a[-1]`. Neste caso, Java levanta a exceção indicando o erro.
 - um **erro interno** ocorre em Java

Copyright © 2002 Qualiti. Todos os direitos reservados.

Declaração e lançamento

- ▶ Quando uma exceção é lançada e não é tratada, o fluxo de controle **passa para o método invocador** e **sobe** pelas chamadas de métodos até que a exceção seja tratada
- ▶ Se a exceção não for tratada em lugar nenhum, Java assume o controle e pára o programa

Copyright © 2002 QualiTi. Todos os direitos reservados.

Quando um método que declara uma exceção não é o responsável por lançá-la originalmente, o mesmo pode somente declará-la em sua assinatura e deixar para que as classes em uma hierarquia acima capturem e tratem a exceção.

Se nenhuma das classes da hierarquia trata a exceção, a máquina virtual Java assume o controle e pára o programa.

Lançamento de Exceções

```
class Conta {
    /* ... */

    void debitar(double valor) throws
    SaldoInsuficienteException {
        if (valor <= saldo) {
            saldo = saldo - valor;
        }
        else {
            SaldoInsuficienteException sie;
            sie =
                new SaldoInsuficienteException(numero,saldo);
            throw sie;
        }
    }
}
```

Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 309



Antes de lançar uma exceção para as classes da hierarquia acima de onde a mesma é declarada, primeiramente um objeto do tipo da exceção deve ser criado, usando a palavra chave **new** e depois o mesmo deve ser lançado usando a palavra chave **throw**. Este processo pode ser declarado em uma única linha, como no exemplo a seguir:

```
void debitar(double valor) throws SaldoInsuficienteException {
    if (valor <= saldo) {
        saldo = saldo - valor;
    }
    else {
        throw new SaldoInsuficienteException(numero,saldo);
    }
}
```

Declarando e lançando exceções

```
class Conta {  
    /* ... */  
    void transferir(Conta c, double v)  
        throws  
        SaldoInsuficienteException {  
  
        this.debitar(v);  
        c.creditar(v);  
    }  
}
```

debitar levanta
uma exceção!

Exceções levantadas indiretamente
também devem ser declaradas!

Copyright © 2002 QualiTi. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 310



Uma vez que um método declara e lança uma exceção, outro método que venha utilizar este pode decidir em declarar a mesma exceção em sua assinatura ou realizar o tratamento das mesma.

Por exemplo o método **transferir()** da classe conta utiliza o método **debitar** da mesma classe (uso do **this**). O método **debitar** gera a exceção **SaldoInsuficienteException**. Neste caso o método **transferir** pode declarar a mesma exceção **SaldoInsuficienteException** e deixar com que o cliente da mesma capture esta exceção para tratamento e disponibilização da mensagem de erro para o usuário.

Tratamento de exceções

- ▶ Exceções são tratadas usando blocos **try-catch**

```
try {  
    // chamada aos métodos que podem lançar exceções  
}  
catch (Exc1 e1) {  
    // código para tratar um tipo de exceção  
}  
catch (ExcN eN) {  
    // código para tratar outro tipo de exceção  
}
```

Copyright © 2002 Quality. Todos os direitos reservados.

Para o tratamento de exceções é necessário utilizar um construtor específico em Java. Um bloco **try-catch**.

Dentro do bloco **try** devem ser especificadas as chamadas para os métodos que podem lançar exceções. Deve existir um bloco **catch** para cada exceção possível de ser gerada pelas chamadas realizadas dentro do bloco **try**.

Caso alguma falha aconteça, o fluxo do programa é direcionado para o bloco **catch** relacionado à exceção gerada. Cada bloco **catch** é responsável por definir código para o tratamento da exceção correspondente.

Tratamento de exceções

```
class CadastroContas {  
    /* ... */  
  
    void debitar(String n, double v)  
        throws SaldoInsuficienteException,  
            ContaInexistenteException {  
  
        Conta c = contas.procurar(n);  
        c.debitar(v);  
  
    }  
}
```

Copyright © 2002 Quali. Todos os direitos reservados.

Por exemplo, a classe `CadastroContas` deste exemplo, possui o método `debitar` que faz chamadas ao método `debitar` de `Conta`. Como o método `debitar` de `Conta` gera duas exceções, **`SaldoInsuficienteException`** e **`ContaInexistenteException`**, o método `debitar` de `CadastroContas` deve declarar as mesmas exceções em sua assinatura, de modo que o cliente da classe `CadastroContas` deve tratar tais exceções.

Tratamento de exceções

```
public static void main(String args[]){

    try {

        contas.debitar("123-4",90.00);
        System.out.println("Débito efetuado");

    }catch (SaldoInsuficienteException sie) {
        System.out.println(sie.getMessage());
        System.out.print("Conta/saldo: ");
        System.out.print(sie.getNumero() + "/" +
            e.getSaldo());
    }catch(ContaInexistenteException cie) {
        System.out.print(cie.getMessage());
    }
    ...
}
```

Copyright © 2002 Quali. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 313



Um cliente da classe CadastroContas deve inserir as chamadas para o método debitar desta classe dentro de um bloco **try-catch**. Para o exemplo citado devem existir dois blocos catch, um para a exceção **SaldoInsuficienteException** e outro para a exceção **ContaInexistenteException**. Cada bloco catch deve inserir o código necessário para o tratamento das exceções. No exemplo, este cliente declara código para a impressão das mensagens de erro (encapsuladas pelas exceções) na tela.

Tratamento de exceções

- ▶ A execução do **try** termina ao final do bloco ou assim que uma exceção é levantada
- ▶ O primeiro **catch** de um supertipo da exceção é executado e o fluxo de controle passa para o código seguinte ao último catch
- ▶ Exceções mais específicas devem ser capturadas primeiro. Caso contrário, um erro de compilação é gerado
- ▶ Se não houver nenhum catch compatível, a exceção e o fluxo de controle são passados para o método que invocou

Copyright © 2002 Qualiti. Todos os direitos reservados.

Usando finally

- um trecho de código com **finally** é sempre executado, havendo ou não exceções

```
try {  
    // chamada aos métodos que podem lançar exceções  
}  
catch (Excl e1) {  
    ...  
}  
catch (ExcN eN) {  
    ...  
}  
finally {  
    // Este código é sempre executado  
}
```

Copyright © 2002 Quality. Todos os direitos reservados.

Outro construtor que pode ser utilizado com um bloco **try-catch** é o construtor **finally**. O mesmo deve ser adicionado sempre ao final do último bloco **catch**. Um bloco **finally** deve ser declarado quando se deseja que um determinado código sempre seja executado independente se a operação foi realizada com sucesso ou se uma exceção foi gerada.

Por exemplo, pode ser necessário definir código para desconectar ao banco de dados assim que os comandos do bloco **try** forem executados. Este requisito pode ser obrigatório, para um dado cenário, independente se o código dentro do bloco **try** foi realizado com sucesso ou não.

Usando finally

```
public static void main(String args[]){
    ...
    try {

        contas.debitar("123-4",90.00);
        System.out.println("Débito efetuado");

    }catch (SaldoInsuficienteException e) {
        System.out.println(e.getMessage());
        System.out.print("Conta/saldo: ");
        System.out.print(e.getNumero() + "/" + e.getSaldo());

    }catch (ContaInexistenteException e) {
        System.out.print(e.getMessage());

    }finally {
        System.out.println("Obrigado, volte sempre!");
    }
    ...
}
```

Copyright © 2002 Qualiti. Todos os direitos reservados.

Exceções e redefinição de métodos

- ▶ Métodos redefinidos não devem declarar exceções que não sejam as **mesmas** ou **subclasses** das exceções declaradas no método original
- ▶ Métodos redefinidos podem declarar um número menor ou maior de exceções que o declarado no método original contanto que a condição acima se verifique

Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 317



Existem algumas regras que devem ser seguidas quando desenvolvendo programas que utilizam *Checked Exceptions*.

Quando as classes que declaram exceções estão envolvidas em uma hierarquia de herança, métodos de uma subclasse não podem ser redefinidos com exceções que não sejam as mesmas definidas nos métodos da superclasse ou exceções que sejam subclasses destas. Isto fica evidente quando utilizando os métodos polimorficamente. Por exemplo:

```
classe A {  
    void m() throws Excecao1  
}  
  
classe B extends A {  
    void m() throws Excecao1, Excecao2  
}
```

Onde, Exceção 2 **não** é subclasse de Excecao1.

Se um objeto do tipo B for atribuído a uma variável do tipo A, a chamada para o método **m()** será realizada polimorficamente. Desta forma, a classe A não tem conhecimento de que a classe B declara a exceção **Excecao2**. Se uma falha ocorrer durante a execução do método **m()** e esta estiver relacionada a **Excecao2**, a mesma não será manipulada, uma vez que não é visível ao objeto de tipo da superclasse. Por este motivo, métodos da subclasse só podem declarar exceções que sejam as mesmas declaradas nos métodos da Superclasse ou exceções que são subclasses destas.

Referências do módulo

- ▶ Java Tutorial
 - <http://java.sun.com/docs/books/tutorial/>
- ▶ Documentação da Versão 6 da linguagem
 - <http://java.sun.com/javase/6/docs/>

Copyright © 2002 Qualiti. Todos os direitos reservados.



Módulo 11

APIs de Java e Classes Wrappers

Copyright © 2002 Qualiti. Todos os direitos reservados.

Qualiti Software Processes



APIs da Plataforma Java 2 Standard Edition v1.4

Swing		AWT	
Sound	Input Methods	Java 2D	Accessibility
RMI	JDBC	JNDI	CORBA
XML	Logging	Beans	Locale Support
Preferences	Collections	JNI	Security
Lang	Util	New I/O	Networking

- ☐ User Interface Toolkits
- ☐ Integration APIs
- ☐ Core APIs

Para dar subsídios ao desenvolvimento de aplicações Java. A plataforma Java disponibiliza uma série de classes e interfaces, que são agrupadas em diferentes pacotes. Este grupamento total é chamado de API (Application Programming Interface).

As classes que compõem a API de Java podem ser categorizada em três tipos (Plataforma Java Standard Edition, V1.4):

- *User Interface Toolkits* - conjunto de classes que dão suporte ao desenvolvimento de GUIs, formado pelos pacotes *Swing*, *AWT*, *Sound*, *Input Methods*, *Java 2D* e *Accessibility*.
- *Integration APIs* - conjunto de classes que dão suporte ao desenvolvimento de aplicações distribuídas e acesso a banco de dados, formado pelos pacotes *XML*, *JDBC*, *JNDI* e *CORBA*.
- *Core APIs* - representa o conjunto de classes que representam o “núcleo” da linguagem Java, contendo alguns pacotes indispensáveis (pacote *Lang*, por exemplo) para o desenvolvimento de qualquer aplicação Java. É formado pelos pacotes *XML*, *Logging*, *Beans*, *Locale Support*, *Preferences*, *Collections*, *JNI*, *Security*, *Lang*, *Util*, *New I/O* e *Networking*.

Pacote java.lang

- ▶ É o principal pacote de Java
- ▶ Contém as classes fundamentais da plataforma
- ▶ É importado automaticamente em todas as classes criadas
- ▶ Classes essenciais:
 - Object, Throwable, Exception, String, Thread, Runnable, Math, System, Runtime

Copyright © 2002 Qualiti. Todos os direitos reservados.

O pacote *lang* representa o núcleo da linguagem Java, sem o qual aplicações não poderiam ser desenvolvidas. Por este motivo, ele é importado por *default* em qualquer classe Java. Ou seja, a declaração explícita de import para este não é necessária uma vez que o mesmo já é declarado por *default* implicitamente.

Classe Object

- ▷ É a superclasse de todas as classes de Java
- ▷ É a única classe sem superclasse
 - **boolean equals(Object obj)**: verifica se dois objetos são iguais
 - **String toString()**: retorna um String descrevendo o objeto
 - **Object clone()**: retorna uma cópia do objeto (mas só se a interface Cloneable for implementada)

A maioria dos métodos de Object é redefinida nas subclasses

A classe *Object* é a superclasse de todas as classes Java e, por este motivo, se encontra no topo da hierarquia de qualquer classe Java. Esta classe apresenta alguns métodos que podem ser reutilizados ou redefinidos pelas suas subclasses.

Throwable e Exception

- ▶ Throwable é a superclasse de todas as exceções
- ▶ Exception herda de Throwable. É a classe que deve ser usada como base para a definição de novas exceções

•**String getMessage():** retorna a mensagem encapsulada na exceção (retorno pode ser nulo)

•**void printStackTrace():** mostra a pilha de exceção (a seqüência de chamadas de métodos que resultaram na exceção)

Copyright © 2002 Qualiti. Todos os direitos reservados.

As classes *Throwable* e *Exception* também fazem parte do pacote *java.lang*. A classe *Throwable* é a classe que fica no topo da hierarquia de classes de exceções e erros. As classes *Exception* e *Error* são subclasses de *Throwable*.

A classe *Throwable* também define alguns métodos que podem ser resutilizados ou redefinidos em suas subclasses.

String

- ▶ Encapsula cadeias de caracteres e muitas operações de manipulação
- ▶ Vista detalhadamente na aula sobre Strings e Arrays

Copyright © 2002 QualiTi. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 324



Outra conhecida que faz parte do pacote `java.lang` é a classe *String*, a qual encapsula cadeias de caracteres e como a maioria das classes Java também fornece operações para manipulação sobre os caracteres definidos.

Strings são classes declaradas *final*, o que implica que a mesma não pode ser “subclasseada”. Portanto classes do tipo *String* não podem ter seus métodos redefinidos.

Thread e Runnable

- ▶ A classe *Thread* e a interface *Runnable* são essenciais para a implementação de aplicações com concorrência em Java

Copyright © 2002 QualiTi. Todos os direitos reservados.

Qualiti Software Processes
Java Básico | 325



A classe *Thread* e a interface *Runnable* possuem elementos que podem ser utilizados ou redefinidos em subclasses e subtipos destas, para o desenvolvimento de aplicações que têm como característica o acesso concorrente e compartilhado de recursos.

System

- ▷ Define uma interface padrão, independente de plataforma, para recursos do sistema operacional:
 - ▷ **out**: saída padrão
 - ▷ **in**: entrada padrão
 - ▷ **currentTimeMillis()**: hora atual em milisegundos desde 1 de Janeiro, 1970, GMT
 - ▷ **arrayCopy(Object origem, int origem_pos, Object destino, int destino_pos, int comprimento)**: copia um array ou parte de um array para outro
 - ▷ **exit()**: força a saída da aplicação

Copyright © 2002 Qualiti. Todos os direitos reservados.



Pacote java.util

- ▶ Define classes e interfaces de coleções e outras classes utilitárias
- ▶ Coleções:
 - Collection
 - List
 - Set, SortedSet
 - Vector, Stack
 - ArrayList, LinkedList
 - Map
 - Hashtable
 - Arrays
 - Iterator

Copyright © 2002 QualiTi. Todos os direitos reservados.



O pacote java.util define um conjunto de classes úteis para o desenvolvimento de aplicações. Por exemplo, programas que precisam manipular coleções de objetos ou mesmo coleções de tipos primitivos; o pacote java.util, fornece, entre outras coisas, mecanismos (classes e interfaces) que permitem a manipulação de tais aspectos de forma mais fácil.

Classes utilitárias

- ▷ Date, Calendar e GregorianCalendar
- ▷ Timer e TimerTask
- ▷ Random
- ▷ Properties
- ▷ StringTokenizer

Copyright © 2002 Qualiiti. Todos os direitos reservados.



Outras classe importantes do pacote java.util:

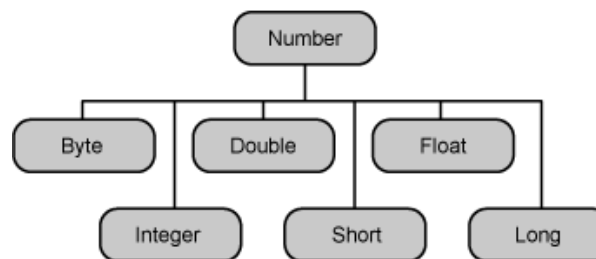
- Classes *Date*, *Calendar* e *GregorianCalendar*, as quais manipulam datas em diferentes formatos.
- *Time* e *TimerTask*, são classes que definem elementos que auxiliam no escalonamento de tarefas em programas concorrentes.
- *Random*, *Properties* e *StringTokenizer*, são classes que geram e retornam números randômicos de diferentes maneiras, podem representar um conjunto persistente de propriedades e permitem à uma aplicação dividir Strings em diferentes *tokens*, respectivamente.

Com as apresentadas neste módulo, existem muitas outras classes de diferentes pacotes que fazem parte da API de Java que definem elementos para auxiliar na construção de programas Java. Cada uma com um propósito específico.

Essas classes não precisam ser decoradas. Para evitar este tipo de inconveniente, o programador deve sempre fazer uso da API de Java, a qual é documentada no formato HTML e pode ser obtida do site da *Sun Microsystems*. Assim, sempre que precisar de alguma funcionalidade, o programador pode pesquisar se a API de Java a disponibiliza em alguma classe.

Classes Wrappers

- ▶ Classes que representam os tipos numéricos de Java
- ▶ Utilizadas quando precisamos manipular tipos primitivos como objetos
- ▶ Todas as classes wrappers tem como super classe a classe Number



Utilizando Wrappers

- ▶ A partir da versão 5.0 da linguagem as classes wrappers podem ser utilizadas como tipos primitivos.
- ▶ A operação de armazenar implicitamente um primitivo em um wrapper é chamada de Unbox e a de recuperar um primitivo implicitamente de um wrapper de Outbox

Copyright © 2002 Qualiti. Todos os direitos reservados.

Utilizando classes Wrappers

```
public static void main(String args[]){
```

```
    Integer i = new Integer(10);
```

Utilização normal

```
    Integer j = 30;
```

Unbox

```
    int x = i + j;
```

Outbox

```
    Integer resultado = x;
```

```
    System.out.println( i + "+" + j + "=" + resultado );
```

```
}
```

Referências do módulo

- ▶ Java Tutorial
 - <http://java.sun.com/docs/books/tutorial/>
- ▶ Documentação da Versão 6 da linguagem
 - <http://java.sun.com/javase/6/docs/>

Copyright © 2002 Qualiti. Todos os direitos reservados.

