Java Intro – Fundamentos de Orientação a Objetos

AVISO DE RESPONSABILIDADE

As informações contidas neste material de treinamento são distribuídas “NO ESTADO EM QUE SE ENCONTRAM”, sem qualquer garantia, expressa ou implícita. Embora todas as precauções tenham sido tomadas na preparação deste material, a Alfamídia Prow não tem qualquer responsabilidade sobre qualquer pessoa ou entidade com respeito à responsabilidade, perda ou danos causados, ou alegadamente causados, direta ou indiretamente, pelas instruções contidas neste material ou pelo software de computador e produtos de hardware aqui descritos.

Versão 1.0

Alfamídia Prow
http://www.alfamidia.com.br

JAVA INTRO: FUNDAMENTOS DE ORIENTAÇÃO A OBJETOS

Unidade 1 - Paradigma de Programação Orientada a Objeto

Paradigma é um conjunto de regras que estabelecem fronteiras e descrevem como resolver os problemas dentro dessas fronteiras. Os paradigmas influenciam nossa percepção; ajudam-nos a organizar e coordenar a maneira como olhamos para o mundo. (Definição literal)

1.1 Um pouco de história

O conceito de programação orientada a objeto não é algo novo. No final da década de 60, a linguagem Simula67, desenvolvida na Noruega, introduzia conceitos hoje encontrados nas linguagens orientadas a objetos. Em meados de 1970, o Centro de Pesquisa da Xerox (PARC) desenvolveu a linguagem Smalltalk, a primeira totalmente orientada a objetos. No início da década de 80, a AT&T lançou a Linguagem C++, uma evolução da linguagem de programação C em direção à orientação a objetos.

Atualmente, um grande número de linguagens incorpora características de orientação a objeto, tais como Java, Object Pascal, Python , etc.

1.2 Orientação a objeto – Conceitos básicos

O paradigma de programação orientada a objeto é baseado em alguns conceitos que definem uma forma de criar programas para computadores. A filosofia principal desse paradigma de programação é solucionar um problema (via programação) através da representação computacional dos objetos existentes nesse problema (objetos do mundo real), usando para isso os conceitos mencionados acima. Com isso, essa maneira de se desenvolver programas fica mais próxima das situações, dos problemas como eles acontecem no mundo real. Será um pouco difícil entender todas as vantagens de OO (Orientação a Objetos). Agora, portanto, vamos mencionar apenas duas:

- É mais fácil conversar com o cliente que pediu o software se falarmos com objetos que existem no mundo dele (o mundo do software fica mais perto do mundo real).

- O software feito com OO pode ser feito com maior qualidade e pode ser mais fácil de escrever e, principalmente, alterar no futuro.

O nosso mundo está repleto de objetos, sejam eles concretos ou abstratos. Qualquer tipo de objeto possui **atributos** (características) e **comportamentos** que são inerentes a esses objetos.

Para usar objetos na programação, primeiro precisamos distinguir entre um **objeto real** e a **representação de um objeto**.

No desenvolvimento de programas, sempre trabalhamos com representações de objetos. Essa representação precisa refletir os objetos reais, ou seja, objetos existentes no problema do mundo real que queremos solucionar.

Por exemplo, um sistema de conta corrente não manipula diretamente os clientes, contas e cheques (esses são os objetos reais do problema).

Em vez disso, o software deve criar representações desses objetos, com os mesmos atributos e comportamentos dos objetos do mundo real. Essa representação é chamada de

1.3 Abstração

Para entender isso, vamos voltar ao exemplo do sistema de conta de corrente.

No mundo real existem vários objetos Cliente, vários objetos Conta e vários objetos Agência (pode ser somente um), ou seja, muito provavelmente não existirá somente um cliente, uma conta e uma agência.

Todos os objetos cliente possuem o mesmo conjunto de atributos, por exemplo, todos os objetos cliente possuem nome e endereço, assim como todos os objetos conta possuem um número, um saldo, um histórico de transações e todos os objetos agência possui um número e um nome. Assim como os atributos, todos os objetos de um mesmo tipo compartilham do mesmo conjunto de **comportamentos**, por exemplo, todos os objetos Cliente fazem depósitos e saques. Apesar de cada objeto ter o mesmo conjunto de atributos, cada objeto possui valores próprios para cada atributo, ou seja, cada objeto é único.

Essa explicação para objetos do mundo real vale para quando trabalhamos com a representação desses objetos no contexto de desenvolvimento desse sistema (programação).

Portanto, podemos perceber que o sistema de conta corrente (programa de conta corrente) será composto de vários objetos, cada qual com um conjunto de atributos e comportamento em comum (se forem do mesmo tipo) e com valores próprios nos seus atributos. A figura abaixo ilustra os vários objetos cliente, conta e agência.

1.4 Para finalizar essa seção vamos fazer uma pequena revisão:

Um programa orientado a objeto é composto por vários objetos do mesmo tipo e de outros tipos que se relacionam. Chamamos esses objetos do mundo do software de representações de objetos.

Como num programa muito provavelmente existirão vários objetos do mesmo tipo, devemos criar uma estrutura ou *template* que represente os atributos e comportamentos desses objetos do mesmo tipo, essa estrutura é chamada de classe. Portanto classe é uma estrutura a partir da qual os objetos (do software) são criados.

1.5 Histórico da Linguagem Java

Em 1991, um pequeno grupo de funcionários da Sun mudou-se para a San Hill Road, uma empresa filial. O grupo estava iniciando um projeto denominado Projeto Green, que consistia na criação de tecnologias modernas de software para empresas eletrônicas de consumo.

Logo o grupo percebeu que não poderia ficar preso a plataformas pois os clientes não estavam interessados no tipo de processador que estavam utilizando e fazer uma versão do projeto para cada tipo de sistema seria inviável. Desenvolveram então o sistema operacional GreenOS, com a linguagem de programação também criada por eles chamada na época de Oak (carvalho, em inglês); nome dado pelo chefe do projeto, que enquanto pensava numa estrutura de diretórios para a linguagem observava pela janela um carvalho. Mas esse nome não pode ser utilizado, pois já havia sido registrado; sendo assim, não demorou muito e o nome Java surgiu em homenagem à terra de origem do café apreciado pelos programadores da equipe.

Em 1993, surgiu uma oportunidade para o grupo Green, agora incorporado como FirstPerson a Time-Warner, uma empresa que estava solicitando propostas de sistemas operacionais de decodificadores e tecnologias de vídeo sob demanda. Isso foi na mesma época em que o NCSA lançou o MOSAIC 1.0, o primeiro navegador gráfico para Web. A FirstPerson apostou nos testes de TV da Time-Warner, mas esta empresa preferiu optar pela tecnologia oferecida pela Silicon Graphics.

Depois de mais um fracasso a FirtsPeron dissolveu-se e metade do pessoal foi trabalhar para a Sun Interactive com servidores digitais de vídeo. Entretanto, a equipe restante continuou os trabalhos do projeto na Sun.

Finalmente em maio de 1995 a Sun anunciou um ambiente denominado Java que obteve sucesso graças a incorporação deste ambiente a browsers populares como o Netscape Navigator e padrões tridimensionais como o VRML (Virtual Reality Modeling Language – Linguagem de Modelagem para Realidade Virtual).

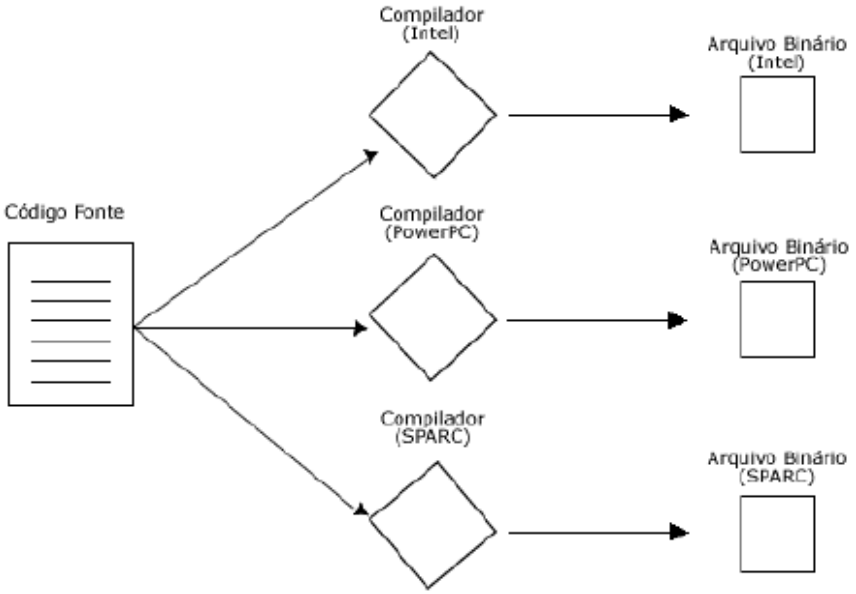
A Sun considera o sucesso do Java na Internet como sendo o primeiro passo para utilizá-lo em decodificadores da televisão interativa em dispositivos portáteis e outros produtos eletrônicos de consumo – exatamente como o Java tinha começado em 1991. Sua natureza portável (roda em qualquer ambiente) e o projeto robusto permitem o desenvolvimento para múltiplas plataformas, em ambientes tão exigentes como os da eletrônica de consumo.

1.6 Principal Característica

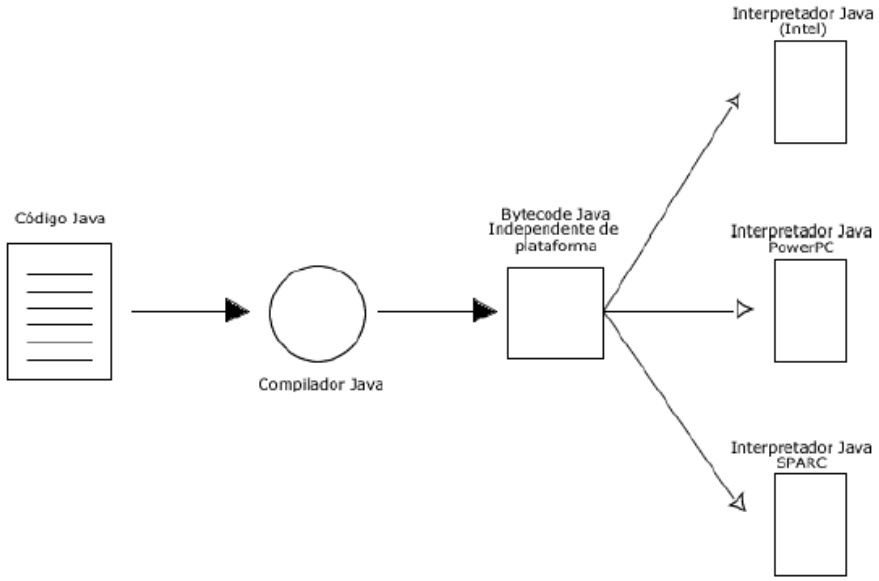
Independência de Plataforma

A independência de plataforma (portabilidade) é a capacidade de o mesmo programa ser executado em diferentes plataformas e sistemas operacionais.

Quando você compila um programa em C ou na maioria das outras linguagens, o compilador transforma seu arquivo-fonte em código de máquina – instruções específicas para o processador que seu computador está executando. Se você compilar seu código em um sistema baseado em plataforma Intel, o programa resultante será executado em outros sistemas baseados na plataforma Intel, mas não funcionará em computadores Macintosh, Commodore VIC-20 ou outras máquinas. Se você usar o mesmo programa em outras plataformas, deve transferir o seu código fonte para a nova plataforma e recompilá-lo para produzir o código de máquina específico para esse sistema. Em muitos casos , serão exigidas alterações no código fonte antes que ele seja compilado na nova máquina, devido a diferenças em seus processadores e outros fatores.



Os programas Java atingem essa independência através da utilização de uma máquina virtual (JVM – Java Virtual Machine), uma espécie de computador dentro de outro. A máquina virtual pega os programas Java compilados e converte suas instruções em comandos que um sistema operacional possa manipular. O mesmo programa compilado, que existe em um formato chamado de bytecode, pode ser executado em qualquer plataforma e sistema operacional que possua uma JVM.



1.7 Compilador, Código Fonte, JVM

Compilador Java javac

Um programa fonte em Java pode ser desenvolvido em qualquer editor que permita gravar textos sem caracteres de formatação. Uma vez que o programa tenha sido salvo em um arquivo, é preciso compilá-lo.

Para compilar um programa Java, a ferramenta oferecida pelo kit de desenvolvimento Java é o compilador Java, **javac**.

Na forma mais básica de execução, o javac é invocado da linha de comando tendo por argumento o nome do arquivo com o código Java a ser compilado. Arquivos com código Java sempre utilizam a extensão **.java**. Por exemplo, se o código a ser compilado está em um arquivo de nome Hello.java,

```
class Hello {
  public static void main(String[] args) {
    System.out.println("Oi!");
  }
}
```

sua compilação dá-se através do comando:

```
> javac Hello.java
```

O resultado dessa execução, se o programa fonte estiver sem erros, será a criação de um arquivo Hello.class contendo o *bytecode* que poderá ser executado em qualquer máquina.

Código Fonte

Um arquivo contendo código Java constitui uma unidade de compilação, podendo incluir comentários, declaração relacionadas a pacotes e pelo menos uma definição de classe ou interface.

Comentários em Java podem ser expressos em três formatos distintos:

//

O restante da linha corrente é um comentário.

/* ... */

Comentário no estilo C: todo o texto entre /* e */ é um comentário, podendo estender-se por diversas linhas.

/** ... */

Comentário no padrão javadoc: o texto entre /** e */ será utilizado para gerar a documentação do código em formato hipertexto.

Declarações relacionadas a pacotes podem ser:

package *nome.do.pacote*;

Opcional, mas se presente deve ser o primeiro comando do arquivo fonte. Indica que as definições que se seguem fazem parte do pacote especificado.

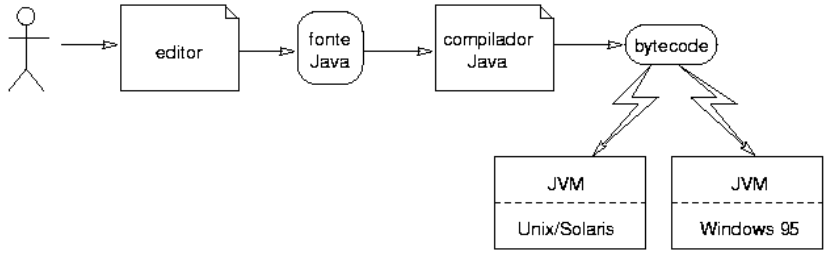
import *nome.do.pacote.Classe*;
import *nome.do.pacote.**;

Indica que a classe especificada ou, no segundo caso, quaisquer classes do pacote especificado serão utilizadas no código fonte.

Bytecodes

Um dos grandes atrativos da plataforma tecnológica Java é a portabilidade do código gerado. Esta portabilidade é atingida através da utilização de *bytecodes*. *Bytecode* é um formato de código intermediário entre o código fonte, o texto que o programador consegue manipular, e o código de máquina, que o computador consegue executar.

Na plataforma Java, o *bytecode* é interpretado por uma **máquina virtual Java** (JVM). A portabilidade do código Java é obtida à medida que máquinas virtuais Java estão disponíveis para diferentes plataformas. Assim, o código Java que foi compilado em uma máquina pode ser executado em qualquer máquina virtual Java, independentemente de qual seja o sistema operacional ou o processador que executa o código:



Java Virtual Machine

A Máquina Virtual Java (JVM) é uma máquina de computação abstrata e um ambiente de execução independente de plataforma. Programas escritos em Java e que utilizem as funcionalidades definidas pelas APIs dos pacotes da plataforma Java executam nessa máquina virtual.

Uma das preocupações associadas a execuções nessas máquinas virtuais é oferecer uma arquitetura de segurança para prevenir que applets e aplicações distribuídas executem fora de seu ambiente seguro (*sandbox*) a não ser quando assim habilitados. Um *framework* de segurança é estabelecido através de funcionalidades dos pacotes java.security, java.security.acl, java.security.cert, java.security.interfaces e java.security.spec.

Interpretador Java

Uma vez que um programa Java tenha sido compilado e esteja pronto para ser interpretado por uma JVM, sua forma básica de execução é através do **interpretador** Java. Por exemplo, se o arquivo Hello.class contém o *bytecode* correspondente ao código fonte do arquivo Hello.java, sua execução dá-se através da linha de comando

```
> java Hello
```

Observe que a extensão .class **não é** incluída na linha de comando. O interpretador java realiza os seguintes passos para executar o código indicado:

Dá início à execução de uma máquina virtual Java;

Carrega a classe indicada para a máquina virtual Java;

Executa o método main() presente nessa classe.

Para que o interpretador possa localizar a classe para a máquina virtual, é preciso que ela esteja localizada em um diretório conhecido da máquina virtual, indicado pela variável CLASSPATH. Esta deve ser definida com a lista de diretórios nos quais a JVM irá procurar pelos arquivos de classes. Por exemplo, para que os arquivos no diretório corrente (indicado por '.') sejam localizados, define-se o CLASSPATH como:

```
> export CLASSPATH=$CLASSPATH:.     (linux/bash)
> set CLASSPATH=%CLASSPATH%;.       (dos)
```

Unidade 2 - Programação Orientada a Objetos

2.1 Princípios da Linguagem Java

Os princípios da programação orientada a objetos são ilustrados através de exemplos na linguagem de programação Java. Aqueles interessados em outras linguagens de programação como C++, Smalltalk, Python, Eiffel podem encontrar aqui referências a estratégias de programação orientada a objetos, mas sem os exemplos específicos nessas linguagens.

2.2 Classes

A definição de um modelo conceitual para o domínio da aplicação, contemplando as classes relevantes e suas associações, é um dos principais resultados das etapas de análise e projeto orientado a objetos. A adoção de uma linguagem de modelagem, tal como o diagrama de classes UML, permite expressar esse resultado de maneira organizada e padronizada.

Uma classe é um gabarito para a definição de objetos. Através da definição de uma classe, descreve-se que propriedades -- ou **atributos** -- o objeto terá.

Além da especificação de atributos, a definição de uma classe descreve também qual o comportamento de objetos da classe, ou seja, que funcionalidades podem ser aplicadas a objetos da classe. Essas funcionalidades são descritas através de **métodos**. Um método nada mais é que o equivalente a um procedimento ou função, com a restrição que ele manipula apenas suas variáveis locais e os atributos que foram definidos para um objeto desse tipo.

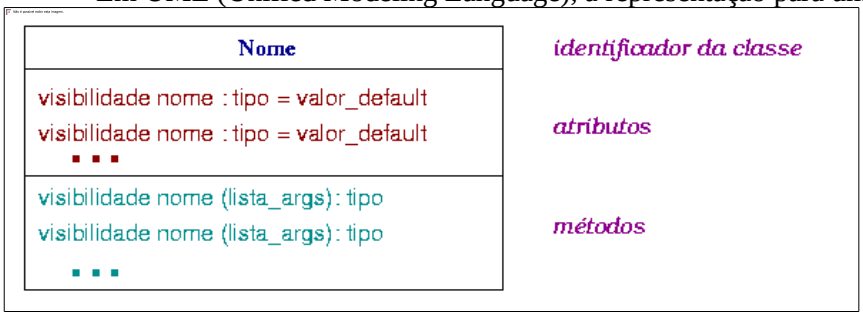
Uma vez que estejam definidas quais serão as classes que irão compor uma aplicação, assim como qual deve ser sua estrutura interna e comportamento, é possível criar essas classes em Java..

O que é uma Classe?

Uma classe é um gabarito para a definição de objetos. Através da definição de uma classe, descreve-se que propriedades -- ou atributos -- o objeto terá.

Além da especificação de atributos, a definição de uma classe descreve também qual o comportamento de objetos da classe, ou seja, que funcionalidades podem ser aplicadas a objetos da classe. Essas funcionalidades são descritas através de métodos. Um método nada mais é que o equivalente a um procedimento ou função, com a restrição que ele manipula apenas suas variáveis locais e os atributos que foram definidos para a classe.

Em UML (Unified Modeling Language), a representação para uma classe no diagrama de classes é:



A especificação de uma classe é composta por três regiões:

Nome da classe

Um identificador para a classe, que permite referenciá-la posteriormente -- por exemplo, no momento da criação de um objeto.

Atributos

O conjunto de propriedades da classe. Para cada propriedade, especifica-se:

nome: um identificador para o atributo.

tipo: o tipo do atributo (inteiro, real, caráter, etc.)

valor_default: opcionalmente, pode-se especificar um valor inicial para o atributo.

visibilidade: opcionalmente, pode-se especificar o quão acessível é um atributo de um objeto a partir de outros objetos. Valores possíveis são:

- (privativo), nenhuma visibilidade externa;
+ (público), visibilidade externa total; e
# (protegido), visibilidade externa limitada.

Métodos

O conjunto de funcionalidades da classe. Para cada método, especifica-se sua **assinatura**, composta por:

nome: um identificador para o método.

tipo: quando o método tem um valor de retorno, o tipo desse valor.

lista de argumentos: quando o método recebe parâmetros para sua execução, o tipo e um identificador para cada parâmetro.

visibilidade: como para atributos, define o quão visível é um método a partir de objetos de outros classes.

As técnicas de programação orientada a objetos recomendam que a estrutura de um objeto e a implementação de seus métodos devem ser tão privativos como possível. Normalmente, os atributos de um objeto não devem ser visíveis externamente. Da mesma forma, de um método deve ser suficiente conhecer apenas sua especificação, sem necessidade de saber detalhes de como a funcionalidade que ele executa é implementada.

2.3 Objetos

É através de objetos que (praticamente) todo o processamento ocorre em aplicações desenvolvidas com linguagens de programação orientadas a objetos. O uso racional de objetos, obedecendo aos bons princípios associados à sua definição conforme estabelecido no paradigma de desenvolvimento orientado a objetos, é chave para o desenvolvimento de bons sistemas de software.

Toda linguagem de programação orientada a objetos oferece mecanismos para definir os tipos de objetos para cada aplicação através do conceito de classes. Objetos são instâncias de classes; essas instâncias precisam ser criadas para que, através de sua manipulação, possam realizar seu trabalho. Após a conclusão de suas atividades, objetos podem ser removidos.

Arranjos de tipos primitivos ou de objetos são criados e manipulados de forma análoga a objetos.

O que é um objeto?

Um **objeto** é um elemento computacional que representa, no domínio da solução, alguma entidade (abstrata ou concreta) do domínio de interesse do problema sob análise. Objetos similares são agrupados em classes.

No paradigma de orientação a objetos, tudo pode ser potencialmente representado como um objeto. Sob o ponto de vista da programação, um objeto não é muito diferente de uma variável no paradigma de programação convencional. Por exemplo, quando define-se uma variável do tipo int em C ou em Java, essa variável tem:

- um espaço em memória para registrar o seu estado atual (um valor);

- um conjunto de operações associadas que podem ser aplicadas a ela, através dos operadores definidos na linguagem que podem ser aplicados a valores inteiros (soma, subtração, inversão de sinal, multiplicação, divisão inteira, resto da divisão inteira, incremento, decremento).

Da mesma forma, quando se cria um objeto, esse objeto adquire um espaço em memória para armazenar seu estado (os valores de seu conjunto de atributos, definidos pela classe) e um conjunto de operações que podem ser aplicadas ao objeto (o conjunto de métodos definidos pela classe).

Um **programa orientado a objetos** é composto por um conjunto de objetos que interagem através de "trocas de mensagens". Na prática, essa troca de mensagem traduz-se na invocação de métodos entre objetos.

Propriedades de objetos

Quando um objeto for ser utilizado em alguma aplicação (por outro objeto, normalmente), a única informação necessária que deve ser conhecida é "o que podemos fazer com ele?", ou seja, qual o conjunto de operações associadas àquele objeto. As técnicas de programação orientada a objetos recomendam que a estrutura interna de um objeto e a implementação de seus métodos devem ser tão privativos como possível.

Há dois conceitos básicos associados a essa visão de um objeto. O primeiro está associado ao fato de que cada objeto é uma unidade que contém, internamente, toda a informação necessária para sua manipulação. Este é o princípio da **encapsulação** -- cada componente de um programa deve agregar toda a informação relevante para sua manipulação como uma unidade (uma *cápsula*).

O segundo conceito fundamental associado a um bom objeto é o **ocultamento da informação**. Em outras palavras, quem usa o objeto não precisa nem deve conhecer como o objeto mantém sua informação ou como uma operação é implementada -- cada componente de um programa deve manter oculta, sob sua guarda, esses aspectos relativos a decisões de projeto.

Na orientação a objetos, cada objeto deve ser manipulado exclusivamente através dos métodos públicos do objeto, dos quais apenas a assinatura deve ser revelada. O conjunto de assinaturas desses métodos constitui a **interface operacional** da classe. O programador trabalha em um nível alto de abstração, sem preocupação com os detalhes internos da classe, simplificando a construção de programas que incorporam funcionalidades complexas, tais como interfaces gráficas e aplicações distribuídas.

Unidade 3 - Classes em Java

3.1 Definição de Classes em Java

Em Java, classes são definidas através do uso da palavra-chave class. Para definir uma classe, utiliza-se a construção:

```
[modif] class NomeDaClasse {
   // corpo da classe...
}
```

A primeira linha é um comando que inicia a declaração da classe. Após a palavra-chave class, segue-se o nome da classe, que deve ser um identificador válido para a linguagem. O modificador *modif* é opcional; se presente, pode ser uma combinação de public e abstract ou final.

A definição da classe propriamente dita está entre as chaves **{** e **}**, que delimitam blocos na linguagem Java. Este corpo da classe usualmente obedece à seguinte seqüência de definição:

As variáveis de classe, iniciando pelas public, seguidos pelas protected, pelas com visibilidade padrão (sem modificador) e finalmente pelas private.

Os atributos (ou variáveis de instância) dos objetos dessa classe, seguindo a mesma ordenação definida para as variáveis de classe.

Os construtores de objetos dessa classe.

Os métodos da classe, geralmente agrupados por funcionalidade.

Toda classe pode também ter um método main associado, que será utilizado pelo interpretador Java para dar início à execução de uma aplicação.

Java também oferece outra estrutura, denominada interface, com sintaxe similar à de classes mas contendo apenas a especificação da funcionalidade que uma classe deve conter, sem determinar como essa funcionalidade deve ser implementada.

Propriedades de uma classe (meta-informação) podem ser obtidas através das funcionalidades oferecidas na classe java.lang.Class.

3.2 Construtores

Um construtor é um (pseudo-)método especial, definido para cada classe. O corpo desse método determina as atividades associadas à inicialização de cada objeto criado. Assim, o construtor é apenas invocado no momento da criação do objeto através do operador new.

A assinatura de um construtor diferencia-se das assinaturas dos outros métodos por não ter nenhum tipo de retorno (nem mesmo void). Além disto, o nome do construtor deve ser o próprio nome da classe.

O construtor pode receber argumentos, como qualquer método. Usando o mecanismo de sobrecarga, mais de um construtor pode ser definido para uma classe.

Toda classe tem pelo menos um construtor sempre definido. Se nenhum construtor for explicitamente definido pelo programador da classe, um construtor *default*, que não recebe argumentos, é criado pelo compilador Java. No entanto, se o programador da classe criar pelo menos um método construtor, o construtor default **não será** criado automaticamente -- se ele o desejar, deverá criar um construtor sem argumentos explicitamente.

No momento em que um construtor é invocado, a seguinte seqüência de ações é executada para a criação de um objeto:

O espaço para o objeto é alocado e seu conteúdo é inicializado (bitwise) com zeros.

O construtor da classe base é invocado.

Os membros da classe são inicializados para o objeto, seguindo a ordem em que foram declarados na classe.

O restante do corpo do construtor é executado.

Seguir essa seqüência é uma necessidade de forma a garantir que, quando o corpo de um construtor esteja sendo executado, o objeto já terá à disposição as funcionalidades mínimas necessárias, quais sejam aquelas definidas por seus ancestrais. O primeiro passo garante que nenhum campo do objeto terá um valor arbitrário, que possa tornar erros de não inicialização difíceis de detectar.

Unidade 4 - Atributos

4.1 Atributos

A definição de atributos de uma classe Java reflete de forma quase direta a informação que estaria contida na representação da classe em um diagrama UML.

Para tanto, a sintaxe utilizada para definir um atributo de um objeto é:

[modificador] tipo nome [= default];

onde

[modificador] tipo nome [= default];

onde

modificador (opcional), uma combinação de

public, protected ou private;

final; e

static.

tipo deve ser um dos tipos primitivos da linguagem Java ou o nome de uma classe;

nome deve ser um identificador válido.

default (opcional) é a especificação de um valor inicial para a variável

4.2 Tipos Primitivos

Em Java, são oferecidos tipos literais primitivos (não objetos) para representar valores:

booleanos

caracteres

numéricos inteiros

numéricos em ponto flutuante

Booleanos

Variáveis do tipo boolean podem assumir os valores true ou false.

O valor *default* para um atributo booleano de uma classe, se não especificado, é false.

Uma variável do tipo boolean ocupa 1 bit de armazenamento.

Variáveis booleanas e variáveis inteiras, ao contrário do que ocorre em C e C++, não são compatíveis em Java. Assim, não faz sentido atribuir uma variável booleana a uma variável inteira ou usar uma valor inteiro como uma condição de um teste.

Exemplo de declaração e uso:

```
boolean deuCerto;
deuCerto = true;
```

Combinando definição e inicialização,

```
boolean deuCerto = true;
```

Caracteres

Uma variável do tipo char contém um caráter Unicode, ocupando 16 bits de armazenamento em memória. O valor default de um atributo de classe do tipo char, se não especificado, é o caráter NUL (código hexadecimal 0000).

Um valor literal do tipo caráter é representado entre aspas simples (apóstrofes), como em:

```
char umCaracter = 'A';
```

Nesse caso, a variável ou atributo umCaracter recebe o caráter A, código hexadecimal 0041 ou 65 decimal.

Valores literais de caracteres podem também ser representados por seqüências de escape, como em '\n' (o caráter *newline*).

Unicode

Unicode é um padrão internacional para a representação unificada de caracteres de diversas linguagens. Citando as palavras do consórcio Unicode:

Caracteres Unicode são codificados em dois bytes, ao invés de um único byte como no padrão ASCII adotado em outras linguagens de programação. Dessa forma, é possível representar um número muito maior de caracteres, permitindo abrigar diversas linguagens.

No entanto, para os usuários que utilizam caracteres do alfabeto latino, não há diferença perceptível. Os caracteres com valores de códigos hexadecimais entre 0000 (sendo 0020, ou 32 decimal, o primeiro caráter não de controle) e 007F (127 decimal) correspondem à codificação do alfabeto *Basic Latin* (equivalente a ASCII), que está contida em Unicode. Da mesma forma, caracteres com valores de códigos hexadecimais entre 0080 (sendo 00A0, ou 160 decimal, o primeiro caráter não de controle) e 00FF (255 decimal) correspondem à codificação do alfabeto *Latin-1 Supplement*. Juntos, esses dois alfabetos constituem a codificação ISO 8859-1 (Latin-1), contida em Unicode:

Seqüências de Escape

A lista de seqüências de escape reconhecidas em Java é:

Seqüência	Caráter
\b	*backspace*
\t	tabulação horizontal
\n	*newline*
\f	*form feed*
\r	*carriage return*
\"	aspas
\'	aspas simples
\\	contrabarra
xxx	o caráter com código de valor octal *xxx*, que pode assumir valores entre 000 e 377

\u*xxxx*	o caráter Unicode com código de valor hexadecimal *xxxx*, onde *xxxx* pode assumir valores entre 0000 e ffff

Seqüências de escape Unicode (\u) são processadas antes das anteriores, podendo aparecer não apenas em variáveis caracteres ou *strings* (como as outras seqüências) mas também em identificadores da linguagem Java.

Inteiros

Valores numéricos inteiros em Java podem ser representados por variáveis do tipo byte, short, int ou long. Todos os tipos contém valores inteiros com sinal, com representação interna em complemento de dois. O valor *default* para atributos desses tipos é 0.

Variáveis do tipo byte ocupam 8 bits de armazenamento interno. Com esse número de bits, é possível representar valores na faixa de -128 a +127.

Variáveis do tipo short ocupam 16 bits, podendo assumir valores na faixa de -32.768 a +32.767.

Variáveis do tipo int ocupam 32 bits, podendo assumir valores na faixa de -2.147.483.648 a +2.147.483.647.

Variáveis do tipo long ocupam 64 bits, podendo assumir valores na faixa de -9.223.372.036.854.775.808 a +9.223.372.036.854.775.807.

Constantes literais do tipo long podem ser identificadas em código Java através do sufixo l ou L, como em:

```
long valorQuePodeCrescer = 100L;
```

Ao contrário do que ocorre em C, não há valores inteiros sem sinal (unsigned) em Java.

Combinações da forma long int ou short int são inválidas em Java.

Reais

Valores reais, com representação em ponto flutuante, podem ser representados por variáveis de tipo float ou double. Em qualquer situação, a representação interna desses valores segue o padrão de representação IEEE 754, sendo 0.0 o valor *default* para tais atributos.

Variáveis do tipo float ocupam 32 bits, podendo assumir valores na faixa de ±1.40239846E-45 a ±3.40282347E+38 (com nove dígitos significativos de precisão).

Variáveis do tipo double ocupam 64 bits, podendo assumir valores na faixa de ±4.94065645841246544E-324 a ±1.79769313486231570E+308 (com 18 dígitos significativos de precisão).

Constantes literais do tipo float podem ser identificadas no código Java pelo sufixo f ou F; do tipo double, pelo sufixo d ou D.

4.3 Classes wrappers

Além dos tipos primitivos da linguagem Java, há classes pré-definidas em Java que oferecem uma série de funcionalidades e facilidades para a manipulação desses tipos:

classe Boolean;

classe Character;

classes para valores numéricos inteiros; e

classes para valores numéricos reais.

Strings não são tipos primitivos em Java, nem tampouco são seqüências de caracteres como em C. São objetos da classe String, também pré-definida em Java e que oferece diversas facilidades para a manipulação de strings.

classe Boolean

A classe java.lang.Boolean oferece facilidades para a manipulação de valores do tipo primitivo boolean.

Por exemplo, o método valueOf() permite converter uma *string* para um valor booleano. Se o argumento do método é a *string* "true" (independente de estar em letras maiúsculas ou minúsculas), esse método retorna o valor booleano verdadeiro (true):

Examplo: Boolean.valueOf("True") returns true.

Examplo: Boolean.valueOf("yes") returns false.

Classe Character

A classe java.lang.Character oferece diversas facilidades para a manipulação de valores do tipo caráter. Por exemplo, dado que uma variável foi definida como

```
char umCaracter = 'a';
```

as seguintes expressões retornariam o valor booleano true:

```
Character.isLetter(umCaracter);
Character.isLetterOrDigit(umCaracter);
Character.isLowerCase(umCaracter);
```

enquanto que as expressões a seguir retornariam false:

```
Character.isDigit(umCaracter);
Character.isISOControl(umCaracter);
Character.isSpaceChar(umCaracter);
Character.isWhitespace(umCaracter);
Character.isUpperCase(umCaracter);
```

Há também funcionalidades para conversão entre o valor de um dígito em uma dada base e sua representação como caráter:

```
int  Character.digit(char c, int radix);
char Character.forDigit(int digit, int radix);
```

e para a conversão entre letras minúsculas e maiúsculas:

```
char Character.toLowerCase(char c);
```

```
char Character.toUpperCase(char c);
```

Classes para manipular valores numéricos inteiros

Para cada tipo numérico inteiro, Java define uma classe correspondente com facilidades para manipulação de seus valores. Assim, são definidas as classes java.lang.Byte, java.lang.Short, java.lang.Integer e java.lang.Long.

Em cada uma dessas classes são definidas duas constantes, que especificam qual os limites de valores representáveis em uma variável do tipo correspondente:

```
Byte.MIN_VALUE
Byte.MAX_VALUE

Short.MIN_VALUE
Short.MAX_VALUE

Integer.MIN_VALUE
Integer.MAX_VALUE

Long.MIN_VALUE
Long.MAX_VALUE
```

Cada classe oferece também funcionalidades para converter uma representação de um valor da forma *string* para a forma numérica:

```
byte   Byte.parseByte(String s);
byte   Byte.parseByte(String s, int radix);
String Byte.toString(byte b);

short  Short.parseShort(String s);
short  Short.parseShort(String s, int radix);
String Short.toString(short s);

int    Integer.parseInt(String s);
int    Integer.parseInt(String s, int radix);
String Integer.toString(int i);
String Integer.toString(int i, int radix);

long   Long.parseLong(String s);
long   Long.parseLong(String s, int radix);
String Long.toString(long l);
String Long.toString(long l, int radix);
```

Classes para manipular valores numéricos reais

Java oferece duas classes com facilidades para manipular valores numéricos reais, java.lang.Float e java.lang.Double.

Para cada uma das classes, são definidas constantes que representam os mínimo e máximo valores representáveis, assim como as representações internas para os valores infinito negativo, infinito positivo e a representação para *Not A Number*:

```
Float.MIN_VALUE
Float.MAX_VALUE
Float.NEGATIVE_INFINITY
Float.POSITIVE_INFINITY
Float.NaN

Double.MIN_VALUE
Double.MAX_VALUE
Double.NEGATIVE_INFINITY
Double.POSITIVE_INFINITY
Double.NaN
```

Há funcionalidades para testar se o valor de uma variável corresponde a uma das situações especiais de representação:

```
boolean Float.isInfinite(float v);
boolean Float.isNaN(float v);

boolean Double.isInfinite(double v);
boolean Double.isNaN(double v);
```

Há funcionalidades para a conversão entre o valor de uma variável e sua representação como *string*:

```
String Float.toString(float f);
float  Float.parseFloat(String s);

String Double.toString(double d);
double Double.parseDouble(String s);
```

Há também funcionalidades para converter entre as representações binárias (internas) de um número real e seu valor:

```
int    Float.floatToIntBits(float f);
float  Float.intBitsToFloat(int bits);

long   Double.doubleToLongBits(double d);
double Double.longBitsToDouble(long bits);
```

Classe String

Ao contrário do que ocorre em C e C++, *strings* em Java não são tratadas como seqüências de caracteres terminadas por NUL. São objetos, instâncias da classe java.lang.String.

Uma *string* pode ser criada como em:

```
String s = "abc";
```

O operador '+' pode ser utilizado concatenar *strings*:

```
System.out.println("String s: " + s);
```

É possível concatenar valores de outros tipos a strings:

```
int umInteiro;
...
System.out.println("Valor: " + umInteiro);
```

onde implicitamente ocorre uma conversão da forma

```
System.out.println("Valor: " + Integer.toString(umInteiro));
```

A classe String define uma série de funcionalidades para manipular strings, tais como:

Obter o número de caracteres em uma *string*:

```
int length();
```

Concatenar *strings*:

```
String concat(String outro);
```

Comparar *strings*:

```
boolean equals(String outro);
boolean equalsIgnoreCase(String outro);
int compareTo(String outro);
```

Extrair caracteres e *substrings*:

```
int charAt(int posição);
String substring(int pos_inicial);
String substring(int pos_inicial, int pos_final);
```

Localizar *substrings*:

```
int indexOf(String substring);
int indexOf(String substring, int pos_inicial);
int lastIndexOf(String substring);
int lastindexOf(String substring, int pos_inicial);
boolean startsWith(String prefixo);
boolean endsWith(String sufixo);
```

Unidade 5 - Métodos

5.1 Métodos

A forma genérica para a definição de um método em uma classe é

```
[modificador] tipo nome(argumentos) {
    corpo do método
}
```

onde

modificador (opcional), uma combinação de:

public, protected ou private;

abstract ou final; e

static.

tipo é um indicador do valor de retorno, sendo void se o método não tiver um valor de retorno;

nome do método deve ser um identificador válido

argumentos são representados por uma lista de parâmetros separados por vírgulas, onde cada parâmetro obedece à forma tipo nome.

Métodos são essencialmente procedimentos que podem manipular atributos de objetos para os quais o método foi definido. Além dos atributos de objetos, métodos podem definir e manipular variáveis locais; também podem receber parâmetros por valor através da lista de argumentos.

Uma boa prática de programação é manter a funcionalidade de um método simples, desempenhando uma única tarefa. O nome do método deve refletir de modo adequado a tarefa realizada. Se a funcionalidade do método for simples, será fácil encontrar um nome adequado para o método.

Como ocorre para a definição de atributos, a definição de métodos reflete de forma quase direta a informação que estaria presente em um diagrama de classes UML, a não ser por uma diferença vital: o corpo do método.

Métodos de mesmo nome podem co-existir em uma mesma classe desde que a lista de argumentos seja distinta, usando o mecanismo de sobrecarga..

5.2 private

A palavra-chave private restringe a visibilidade do membro modificado, método ou atributo, exclusivamente a objetos da própria classe que contém sua definição.

5.3 public

Em Java, a visibilidade padrão de classes, atributos e métodos está restrita a todos os membros que fazem parte de um mesmo pacote. A palavra-chave public modifica essa visibilidade de forma a ampliá-la, deixando-a sem restrições.

Uma classe definida como pública pode ser utilizada por qualquer objeto de qualquer pacote. Em Java, uma unidade de compilação (um arquivo fonte com extensão .java) pode ter no máximo uma classe pública, cujo nome deve ser o mesmo do arquivo (sem a extensão). As demais classes na unidade de compilação, não públicas, são consideradas classes de suporte para a classe pública e têm a visibilidade padrão.

Um atributo público de uma classe pode ser diretamente acessado e manipulado por objetos de outras classes.

Um método público de uma classe pode ser aplicado a um objeto dessa classe a partir de qualquer outro objeto de outra classe. O conjunto de métodos públicos de uma classe determina o que pode ser feito com objetos da classe, ou seja, determina o seu comportamento.

5.4 protected

A palavra-chave protected restringe a visibilidade do membro modificado, atributo ou método, apenas à própria classe e àquelas derivada desta.

5.5 abstract

Uma **classe abstrata** não pode ser instanciada, ou seja, não há objetos que possam ser construídos diretamente de sua definição. Por exemplo, a compilação do seguinte trecho de código

```
abstract class AbsClass {
  public static void main(String[] args) {
    AbsClass obj = new AbsClass();
  }
}
```

geraria a seguinte mensagem de erro:

```
AbsClass.java:3: class AbsClass is an abstract class.
It can't be instantiated.
AbsClass obj = new AbsClass();
               ^
1 error
```

Classes abstratas correpondem a especificações genéricas, que deverão ser concretizadas em classes derivadas.

Método abstrato

Uma classe abstrata pode conter um ou mais **métodos abstratos**. Um método abstrato não cria uma definição, mas apenas uma declaração de um método que deverá ser implementado em uma classe derivada. Se esse método não for implementado na classe derivada, esta permanece como uma classe abstrata mesmo que não tenha sido assim declarada explicitamente. Por exemplo, a compilação de

```
abstract class AbsClass {
```

```
  abstract void meth();
}
class DerClass extends AbsClass {
  public static void main(String[] args) {
    DerClass obj = new DerClass();
  }
}
```

gera dois erros:

```
AbsClass.java:5: class DerClass must be declared abstract.
        It does not define void meth() from class AbsClass.
class DerClass extends AbsClass {
      ^
AbsClass.java:7: class DerClass is an abstract class.
        It can't be instantiated.
    DerClass obj = new DerClass();
                   ^
2 errors
```

No entanto, se for incluída uma implementação de meth() na classe derivada:

```
abstract class AbsClass {
  abstract void meth();
}
class DerClass extends AbsClass {
  void meth() { }
  public static void main(String[] args) {
    DerClass obj = new DerClass();
  }
}
```

então nenhum erro de compilação ocorre.

5.6 final

A palavra chave final pode ser utilizada como uma indicação de algo que não deve ser modificado ao longo do restante da hierarquia de descendentes de uma classe. Pode ser associada a atributos, a métodos e a classes.

Classe final

Uma classe definida como final não pode ser estendida. Assim, a compilação do arquivo Reeleicao.java

```
final class Mandato {
}

public class Reeleicao extends Mandato {
}
```

ocasionaria um erro de compilação:

```
caolho:Exemplos[39] javac Reeleicao.java
Reeleicao.java:4: Can't subclass final classes: class Mandato
public class Reeleicao extends Mandato {
                               ^
1 error
```

Atributo final

Um atributo final pode indicar um valor constante, que não deve ser alterado pela aplicação. Apenas valores de tipos primitivos podem ser utilizados para definir constantes. O valor do atributo deve ser definido no momento da declaração, pois não é permitida nenhuma atribuição em outro momento.

A utilização de final para uma referência a objetos é permitida. Como no caso de constantes, a definição do valor (ou seja, a criação do objeto) também deve ser especificada no momento da declaração. No entanto, é preciso ressaltar que o conteúdo do objeto em geral **pode** ser modificado -- apenas a referência é fixa. O mesmo é válido para arranjos.

A partir de Java 1.1, é possível ter atributos de uma classe que sejam final mas não recebem valor na declaração, mas sim nos construtores da classe. (A inicialização deve obrigatoriamente ocorrer em uma das duas formas.) São os chamados *blank finals*, que introduzem um maior grau de flexibilidade na definição de constantes para objetos de uma classe, uma vez que essas podem depender de parâmetros passados para o construtor.

Argumentos de um método que não devem ser modificados podem ser declarados como final, também, na própria lista de parâmetros.

Método final

Um método que é definido como final em uma classe não pode ser redefinido em classes derivadas. Considere o seguinte exemplo já visto anteriormente:

```
 1:class ComFinal {
 2:    final int f() {
 3:            return 1;
 4:    }
 5:}
 6:
 7:public class ExtComFinal extends ComFinal {
 8:    int f() {
 9:            return 0;
10:    }
11:}
```

A tentativa de compilação dessa unidade geraria a seguinte mensagem de erro

```
[ricarte@mucuripe work]$ javac ExtComFinal.java
ExtComFinal.java:8: The method int f() declared in class ExtComFinal
cannot override the final method of the same signature declared in
class ComFinal.  Final methods cannot be overridden.
   int f() {
      ^
1 error
```

5.7 static

Usualmente, métodos definidos em uma classe são aplicados a objetos daquela classe. Há no entanto situações nas quais um método pode fazer uso dos recursos de uma classe para realizar sua tarefa sem necessariamente ter de estar associado a um objeto individualmente.

Para lidar com tais situações, Java define os métodos da classe, cuja declaração deve conter o modificador static. Um método estático pode ser aplicado à classe e não necessariamente a um objeto.

Variáveis de classe static

Cada objeto definido a partir de uma classe terá sua cópia separada dos atributos definidos para a classe. No entanto, há situações em que é interessante que todos os objetos compartilhem a mesma variável, similarmente ao que ocorre com variáveis globais em linguagens de programação tradicional. O mecanismo para realizar esse compartilhamento é a definição de variáveis de classe.

Uma variável de classe tem sua declaração precedida pela palavra-chave static.

Várias constantes são definidas em Java como public static final. Por exemplo, a classe Math de Java define as constantes E (2.71828...) e PI (3.14159...). Para ter acesso a esses valores, basta precedê-los com o nome da classe e um ponto, como em

```
double pi2 = Math.PI/2;
```

Outro exemplo de variável public static final é a variável out da classe System. Essa variável, System.out, está associada a um objeto que representa a saída padrão (o monitor, tipicamente), sendo utilizada sempre que se deseja enviar um valor para essa saída.

Exemplo System.out

A variável out está associada à apresentação de caracteres na saída padrão, ou seja, na tela do monitor. Na documentação da API Java encontra-se:

out
public static final PrintStream out

Uma simples análise dessa declaração mostra que:

out é uma variável de classe e, portanto, pode ser acessada na forma System.out;

essa variável é pública;

a variável é final; e

a variável é do tipo PrintStream.

Exemplos de métodos estáticos em Java incluem os métodos para manipulação de tipos primitivos definidos nas classes java.lang.Character, java.lang.Integer e java.lang.Double, assim como todos os métodos definidos para a classe java.lang.Math. Por exemplo, para atribuir a raiz quadrada de 2 a uma variável sqr2, a expressão

```
double sqr2 = Math.sqrt(2.0);
```

poderia ser utilizada.

5.8 Definição do Corpo de Métodos

O corpo de um método é formado por declarações de variáveis locais e comandos da linguagem de programação.

A sintaxe de declaração de variáveis locais em Java é similar àquela de declaração de atributos de objetos, sem a opção dos modificadores de visibilidade -- variáveis locais têm visibilidade restrita ao método, exclusivamente. Assim, uma variável declarada em um método tem a forma

```
[modificador]  tipo  nome [ = default];
```

onde

o modificador opcional pode ser exclusivamente final;

tipo deve ser um dos tipos primitivos da linguagem Java ou o nome de uma classe;

nome deve ser um identificador válido.

default (opcional) é a especificação de um valor inicial para a variável.

Embora não seja obrigatório, é uma boa prática de programação manter todas as declarações de variáveis no início do método. Uma exceção aceita refere-se a blocos delimitados por iteração com for, onde a forma

```
for(int i=0; i < maximo; ++i) { ... }
```

é aceita. Neste caso, o escopo da variável i está restrito ao bloco da iteração.

Comandos podem representar uma expressão (uma operação a ser realizada) ou um comando de controle de fluxo de execução.

Unidade 6 - Manipulação de Objetos

6.1 Manipulação de Objetos

Quando declara-se uma variável cujo tipo é o nome de uma classe, como em

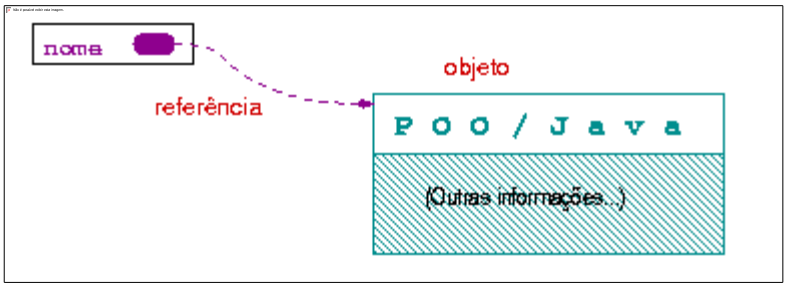
```
String nome;
```

não está se criando um objeto dessa classe, mas simplesmente uma **referência para um objeto** da classe String, a qual inicialmente não faz referência a nenhum objeto válido:

Quando um objeto dessa classe é criado, obtém-se uma referência válida, que é armazenada na variável cujo tipo é o nome da classe do objeto. Por exemplo, quando cria-se uma *string* como em

```
nome = new String("POO/Java");
```

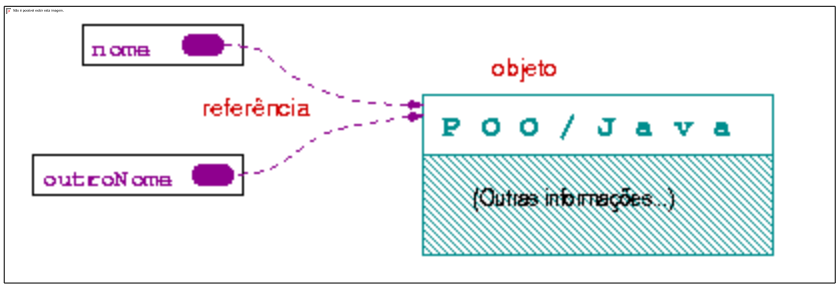
nome é uma variável que armazena uma referência para um objeto específico da classe String -- o objeto cujo conteúdo é "POO/Java":



É importante ressaltar que a variável nome mantém apenas a referência para o objeto e não o objeto em si. Assim, uma atribuição como

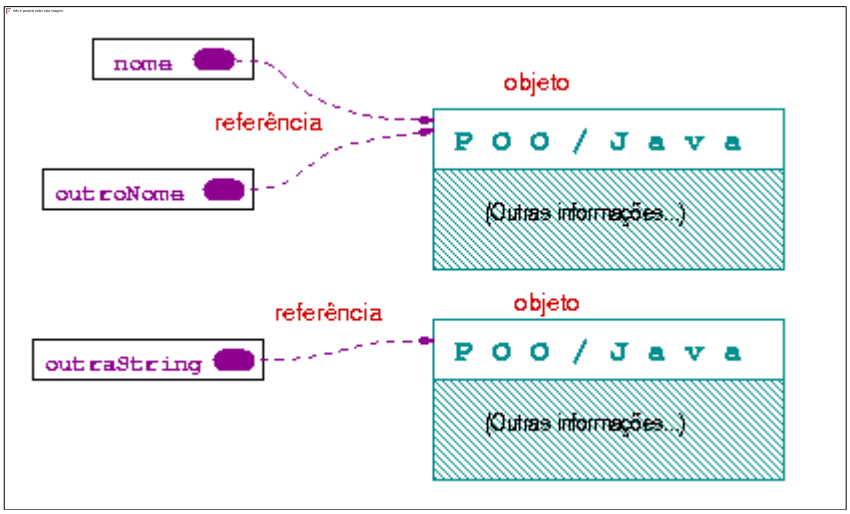
```
String outroNome = nome;
```

não cria outro objeto, mas simplesmente uma outra referência para o mesmo objeto:



O único modo de aplicar os métodos a um objeto é através de uma referência ao objeto. Seguindo com o mesmo exemplo, para criar um novo objeto como mesmo conteúdo do objeto existente, o método clone() pode ser aplicado a este:

String outraString = nome.clone();



Em Java, a palavra-chave this é utilizada por um objeto para fazer a referência a si próprio.

6.2 this

Quando um método é aplicado a um objeto, de algum modo deve ser sinalizado ao método a qual objeto a invocação está referindo-se. Por exemplo,

```
String s1 = "java";
String s2 = "linguagem de programação orientada a objetos";
int x1 = s1.length();  // 4
int x2 = s2.length();  // 44
```

Nesse exemplo, o método aplicado às duas *strings* é o mesmo -- length(). Como o método sabe qual o objeto que ele deve "olhar" para saber o que fazer?

Por trás da resposta a essa pergunta está a presença de um "parâmetro secreto", que é passado a todo método, que é uma referência ao objeto que está realizando a invocação. Em termos de uma linguagem procedimental, isso seria o equivalente a ter para as duas últimas linhas do exemplo acima

:

```
int x1 = length(s1);
int x2 = length(s2);
```

Se necessário, o próprio método pode ter acesso a essa referência em seu código. Essa referência é passada através da palavra-chave this, reservada para esse propósito.

6.3 Exemplos de Uso de this

A palavra-chave this é utilizada principalmente em dois contextos:

- Diferenciar atributos de objetos de parâmetros ou variáveis locais de mesmo nome;

- Acessar o método construtor a partir de outros construtores.

Esse exemplo ilustra esses dois usos

```
class ThisSample {
   int x;
   int y;

   // exemplo do primeiro caso:
   public ThisSample(int x, int y) {
      this.x = x;
      this.y = y;
   }

   // exemplo do segundo caso:
   public ThisSample( ) {
      this(1, 1);
   }
}
```

6.4 Remoção de objetos

Em linguagens de programação orientadas a objetos, a criação de objetos é um processo que determina a ocupação dinâmica de muitos pequenos segmentos de memória. Se esse espaço não fosse devolvido para reutilização pelo sistema, a dimensão de aplicações desenvolvidas com linguagens de programação orientadas a objetos estaria muito limitada.

As duas abordagens possíveis são dedicar essa tarefa ao programador ou deixar que o sistema seja o responsável por esta retomada de recursos. O problema da primeira abordagem é que o programador pode não considerar todas as possibilidades, ocasionando problemas como "vazamento de memória" *(memory leak)*. Na segunda abordagem, recursos adicionais do sistema são necessários para a manutenção da informação necessária para saber quando um recurso pode ser retomado e para a execução do processo que retoma os recursos.

C++ é uma linguagem que adota a primeira estratégia. A remoção de objetos é efetivada explicitamente pelo programador (através do operador delete), que pode também especificar o procedimento que deve ser executado para a liberação de outros recursos alocados pelo objeto através de um método **destructor**.

Java, assim como Smalltalk e lisp, adota a abordagem de ter um *garbage collector* verificando que objetos não têm nenhuma referência válida, retomando o espaço dispensado para cada um desses objetos. Dessa forma, o programador não precisa se preocupar com a remoção explícita de objetos.

Adicionalmente, pode ser necessário liberar outros recursos que tenham sido alocados para o objeto. O método finalize() especifica o que deve ser feito antes do espaço do objeto ser retomado pelo *garbage collector*.

Esse método é definido na classe Object como protected. Portanto, se for necessário redefini-lo o programador deve também declará-lo como protected.

É preciso observar que o uso de finalize() ocorre através da redefinição de métodos. Assim, ao contrário do que acontece com construtores (onde o método correspondente na superclasse é automaticamente invocado), o método finalizador da superclasse deve ser invocado explicitamente (super.finalize()) ao final do corpo do finalizador local.

Uma observação adicional com relação a *garbage collectors* é que programadores não têm como atuar explicitamente sobre esse processo.

No entanto, o programador pode:

- remover explicitamente a referência a um objeto para sinalizar ao GC que o objeto não é mais necessário e pode ser removido;

- sugerir ao (mas não forçar que o) sistema que execute o *garbage collector* através da invocação ao método gc() da classe java.lang.System:

public static void **gc**()

- sugerir que o sistema exexute os métodos finalize() de objetos descartados através do método runFinalization da classe java.lang.System:

6.5 A Classe Object

A classe java.lang.Object é a raiz a partir da qual todas as classes são definidas. Desse modo, as definições dessa classe estão disponíveis para objetos de todas as demais classes.

O método equals, que permite comparar objetos por seus conteúdos, é um dos métodos definido nessa classe:

public boolean **equals**(Object obj)

O método clone() permite criar duplicatas de um objeto

O método toString() permite converter uma representação interna do objeto em uma *string* que pode ser apresentada ao usuário:

A partir do objeto retornado, da classe java.lang.Class, é possível obter o nome da classe usando o método da classe Class getName(), que retorna uma *string* com o nome da classe.

Unidade 7 - Herança

7.1 Herança

Um dos grandes diferenciais da programação orientada a objetos em relação a outros paradigmas de programação está no conceito de herança, mecanismo através do qual definições existentes podem ser facilmente estendidas. Juntamente com a herança deve ser enfatizada a importância do polimorfismo, que permite selecionar funcionalidades que um programa irá utilizar de forma dinâmica, durante sua execução.

O conceito de encapsular estrutura e comportamento em um tipo não é exclusivo da orientação a objetos; particularmente, a programação por tipos abstratos de dados segue esse mesmo conceito. O que torna a orientação a objetos única é o conceito de herança.

Herança é um mecanismo que permite que características comuns a diversas classes sejam fatoradas em uma **classe base**, ou superclasse. A partir de uma classe base, outras classes podem ser especificadas. Cada **classe derivada** ou subclasse apresenta as características (estrutura e métodos) da classe base e acrescenta a elas o que for definido de particularidade para ela.

Sendo uma linguagem de programação orientada a objetos, Java oferece mecanismos para definir classes derivadas a partir de classes existentes. É fundamental que se tenha uma boa compreensão sobre como objetos de classes derivadas são criados e manipulados, assim como das restrições de acesso que podem se aplicar a membros de classes derivadas. Também importante para uma completa compreensão da utilização desse mecanismo em Java é a compreensão de como relacionam-se interfaces e herança.

Herança é sempre utilizada em Java, mesmo que não explicitamente. Quando uma classe é criada e não há nenhuma referência à sua superclasse, implicitamente a classe criada é derivada diretamente da classe Object. É por esse motivo que todos os objetos podem invocar os métodos da classe Object, tais como equals() e toString().

7.2 Formas de Herança

Há várias formas de relacionamentos em herança:

Extensão: subclasse estende a superclasse, acrescentando novos membros (atributos e/ou métodos). A superclasse permanece inalterada, motivo pelo qual este tipo de relacionamento é normalmente referenciado como **herança estrita**.

Especificação: a superclasse especifica o que uma subclasse deve oferecer, mas não implementa nenhuma funcionalidade. Diz-se que apenas a *interface* (conjunto de especificação dos métodos públicos) da superclasse é herdada pela subclasse.

Combinação de extensão e especificação: a subclasse herda a interface e uma implementação padrão de (pelo menos alguns de) métodos da superclasse. A subclasse pode então redefinir métodos para especializar o comportamento em relação ao que é oferecido pela superclasse, ou ter que oferecer alguma implementação para métodos que a superclasse tenha declarado mas não implementado. Normalmente, este tipo de relacionamento é denominado **herança polimórfica**.

A última forma é, sem dúvida, a que mais ocorre na programação orientada a objetos. Algumas modelagens introduzem uma forma de herança conhecida como contração, que deve ser evitada.

7.3 Contração

Contração é uma uma variante de herança onde a subclasse elimina métodos da superclasse com o objetivo de criar uma "classe mais simples". A eliminação pode ocorrer pela redefinição de métodos com corpo vazio. O problema com este mecanismo é que ele viola o princípio da substituição, pois a subclasse já não pode mais ser utilizada em todos os pontos onde a superclasse poderia ser utilizada.

Se a contração parece ser uma solução adequada em uma hierarquia de classes, provavelmente a hierarquia deve ser re-analisada para detecção de inconsistências (problema pássaros-pinguins). De modo geral, o mecanismo de contração deve ser evitado.

7.4 Sintaxe

A forma básica de herança em Java é a extensão simples entre uma superclasse e sua classe derivada. Para tanto, utiliza-se na definição da classe derivada a palavra-chave extends seguida pelo nome da superclasse.

Assim, definir uma classe Ponto2D como em

```
class Ponto2D {
  // ...
}
```

é equivalente a

```
class Ponto2D extends Object {
  // ...
}
```

O exemplo completo ilustra como uma classe Ponto2D pode ser utilizada como base para a definição de outra classe derivada Ponto3D através do mecanismo de extensão simples.

```
class Ponto2D {
   private int x;
   private int y;

   public Ponto2D(int x, int y) {
      this.x = x;
      this.y = y;
   }

   public Ponto2D( ) {
      this(0,0);
   }

   public double distancia(Ponto2D p) {
      double distX = p.x - x;
      double distY = p.y - y;

      return(Math.sqrt(distX*distX + distY*distY));
   }
}

class Ponto3D extends Ponto2D {
```

```
    private int z;

    public Ponto3D(int x, int y, int z) {
        super(x, y);
        this.z = z;
    }

    public Ponto3D( ) {
        z = 0;
    }

    public static void main(String[] args) {
        Ponto2D ref2 = new Ponto2D();
        Ponto2D p2 = new Ponto2D(1,1);
        System.out.println("Distancia2: " + p2.distancia(ref2));
        Ponto3D p3 = new Ponto3D(1,2,3);
        System.out.println("Distancia3: " + p3.distancia(ref2));
    }
}
```

7.5 Construção de Objetos Derivados

Durante a construção de um objeto de uma classe derivada, o construtor de sua superclasse é executado (implicitamente ou explicitamente) antes de executar o corpo de seu construtor.

Assim, ao se construir um objeto de uma classe derivada, o método construtor da superclasse será inicialmente invocado. Este, por sua vez, invocará o construtor de sua superclasse, até que o construtor da classe raiz de toda a hierarquia de objetos -- a classe Object -- seja invocado. Como Object não tem uma superclasse, seu construtor é executado e a execução retorna para o construtor de sua classe derivada. Então executa-se o restante do construtor de sua classe derivada e a execução retorna para o construtor de sua classe derivada e assim sucessivamente, até que finalmente o restante do construtor da classe para a qual foi solicitada a criação de um objeto seja executada.

Construtores da superclasse podem ser explicitamente invocados usando a palavra-chave super

7.6 A Palavra-Chave *super*

Java não oferece o mecanismo de **herança múltipla**, ou seja, não é possível criar uma classe derivada com mais de uma classe base. Por esse motivo, é simples fazer uma referência da classe derivada para sua superclasse; o mecanismo para tal é o uso da palavra-chave super.

Construtores da superclasse podem ser explicitamente invocados usando o método super(). No exemplo do Ponto3D, isso é feito para o construtor da classe derivada com argumentos:

```
public Ponto3D(int x, int y, int z) {
        super(x, y);
        this.z = z;
}
```

A expressão super(x, y); na primeira linha do construtor está invocando o construtor da classe base, Ponto2D, que recebe dois argumentos. O efeito dessa invocação é iniciar os valores dos atributos x e y do objeto com os valores dos parâmetros x e y recebidos pelo método construtor.

A invocação do método super(), se presente, deve estar na primeira linha. Implicitamente, o compilador faz a invocação do construtor super() *default* (sem argumentos) para cada construtor definido. Por esse motivo, o segundo construtor da classe Ponto3D não faz a invocação desse método explicitamente. Assim,

```
public Ponto3D( ) {
        z = 0;
}
```

equivale a

```
public Ponto3D( ) {
   super();
        z = 0;
}
```

Esse é um dos motivos pelo qual é sempre interessante ter o construtor *default* definido. Entretanto, a invocação direta pode ser interessante quando se deseja invocar algum construtor que não o *default*, como no exemplo anterior.

Outro uso dessa palavra-chave é como prefixo para referenciar métodos da superclasse.

```
class Ponto2D {
   private int x;
   private int y;

   public Ponto2D(int x, int y) {
      this.x = x;
      this.y = y;
   }

   public Ponto2D( ) {
      this(0,0);
   }

   public double distancia(Ponto2D p) {
      double distX = p.x - x;
      double distY = p.y - y;

      return(Math.sqrt(distX*distX + distY*distY));
   }
}

class Ponto3D extends Ponto2D {
   private int z;

   public Ponto3D(int x, int y, int z) {
      super(x, y);
      this.z = z;
   }

   public Ponto3D( ) {
      z = 0;
```

```
    }

    public static void main(String[] args) {
        Ponto2D ref2 = new Ponto2D();
        Ponto2D p2 = new Ponto2D(1,1);
        System.out.println("Distancia2: " + p2.distancia(ref2));
        Ponto3D p3 = new Ponto3D(1,2,3);
        System.out.println("Distancia3: " + p3.distancia(ref2));
    }
}
```

7.7 Restrições de acesso

O exemplo da classe Ponto3D ilustra a forma básica de herança -- uma vez que uma classe tenha sido definida como derivada de outra, ela preserva da superclasse os atributos (no caso, os valores das coordenadas x e y) e seus métodos (no caso, o método distancia(Ponto2D) aplicado a um Ponto3D).

No entanto, deve ser observado que nenhum método definido para a classe Ponto3D pode ter acesso aos atributos x e y, pois esses são declarados como private da classe Ponto2D. De fato, caso tente-se definir naquela classe um método para calcular a distancia entre dois pontos no espaço, tal como:

```
public double distancia(Ponto3D p) {
        double distX = p.x - x;
        double distY = p.y - y;
        double distZ = p.z - z;
        return(Math.sqrt(distX*distX + distY*distY + distZ*distZ));
}
```

os seguintes erros seriam detectados pelo compilador Java:

```
Ponto3Derr.java:35: No variable x defined in class Ponto3D.
        double distX = p.x - x;
                       ^
Ponto3Derr.java:35: Undefined variable: x
        double distX = p.x - x;
                         ^
Ponto3Derr.java:36: No variable y defined in class Ponto3D.
        double distY = p.y - y;
                       ^
Ponto3Derr.java:36: Undefined variable: y
        double distY = p.y - y;
                         ^
```

ou seja, mesmo embora a classe Ponto3D tenha os atributos x e y, métodos dessa classe não podem manipulá-los diretamente.

Uma alternativa para permitir que classes derivadas possam manipular atributos da superclasse é declarar tais atributos como protected ao invés de private. Assim, objetos de classes derivadas têm acesso tanto de leitura como de escrita para esses atributos.

Caso deseje-se dar acesso apenas de leitura a esses atributos, uma outra alternativa é usar um **método acessor** para cada atributo, ou seja, um método que simplesmente retorna o valor do atributo. Nesse exemplo, a classe Ponto2D definiria dois métodos,

```
public int getX() {
        return x;
}

public int getY() {
        return y;
}
```

que permitem que qualquer objeto de qualquer classe consiga ler os valores de x e y -- mas não alterá-los. Assim, a classe Ponto3D poderia definir o método para a distância no espaço como

```
public double distancia(Ponto3D p) {
        double distX = p.getX() - this.getX();
        double distY = p.getY() - this.getY();
        double distZ = p.z - z;
        return(Math.sqrt(distX*distX + distY*distY + distZ*distZ));
}
```

que compilaria e executaria sem erros.

Pode-se também permitir acesso de apenas leitura somente para as classes derivadas -- nesse caso, os métodos acessores seriam declarados como protected, como em

```
protected int getX() {
        return x;
}

protected int getY() {
        return y;
}
```

O método distancia(Ponto3D) não precisaria sofrer nenhuma modificação, mas outras classes não-derivadas de Ponto2D não teriam mais como invocar esses métodos acessores.

Unidade 8 - Polimorfismo

8.1 Polimorfismo:

Polimorfismo é o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação (assinatura) mas comportamentos distintos, especializados para cada classe derivada, usando para tanto uma referência a um objeto do tipo da superclasse. A decisão sobre qual o método que deve ser selecionado, de acordo com o tipo da classe derivada, é tomada em tempo de execução, através do mecanismo de ligação tardia.

8.2 Ligação Tardia:

Quando o método a ser invocado é definido durante a compilação do programa, o mecanismo de **ligação prematura** *(early binding)* é utilizado.

Para a utilização de polimorfismo, a linguagem de programação orientada a objetos deve suportar o conceito de **ligação tardia** *(late binding)*, onde a definição do método que será efetivamente invocado só ocorre durante a execução do programa. O mecanismo de ligação tardia também é conhecido pelos termos *dynamic binding* ou *run-time binding*.

Em Java, todas as determinações de métodos a executar ocorrem através de ligação tardia exceto em dois casos:

- métodos declarados como final não podem ser redefinidos e portanto não são passíveis de invocação polimórfica da parte de seus descendentes;

- e métodos declarados como private são implicitamente finais.

No caso de polimorfismo, é necessário que os métodos tenham exatamente a mesma identificação, sendo utilizado o mecanismo de **redefinição de métodos** *(overriding)*. Esse mecanismo de redefinição é muito diferente do mecanismo de sobrecarga de métodos *(overloading)*..

8.3 Sobrecarga (overloading):

Na programação orientada a objetos, um método aplicado a um objeto é selecionado para execução com base na classe a que ele pertence, no nome do método e nos tipos de seus parâmetros. O nome do método com a lista de tipos de parâmetros constituem a **assinatura** do método.

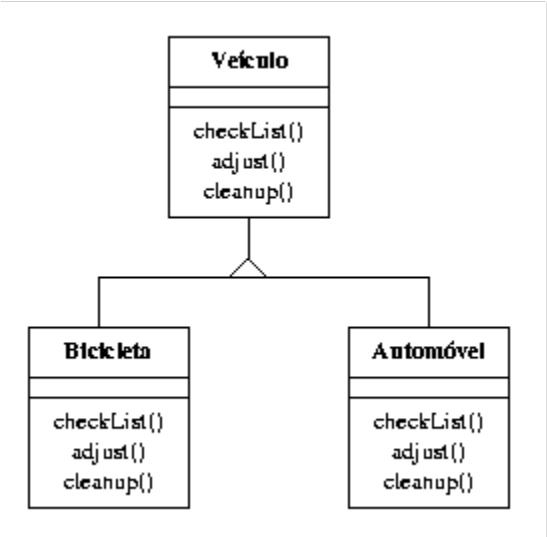
Através do mecanismo de sobrecarga, dois métodos de uma classe podem ter o mesmo nome, desde que suas assinaturas sejam diferentes. Tal situação não gera conflito pois o compilador é capaz de detectar qual método deve ser escolhido a partir da análise dos argumentos do método.

Um exemplo do uso de sobrecarga em Java é encontrado nos métodos abs(), max() e min() da classe java.lang.Math, que têm implementações alternativas para quatro tipos de argumentos distintos.

O uso de polimorfismo em Java é ilustrado através de um exemplo. Através desse exemplo introduzem-se os conceitos relacionados de *upcasting* e a motivação para a definição de métodos abstratos.

8.4 Polimorfismo em Java:

Considere uma classe Veículo com duas classes derivadas, Automóvel e Bicicleta:



Essas classes têm três métodos, definidos para veículos de forma geral e redefinidos mais especificamente para automóveis e bicicletas:

checkList(), para verificar o que precisa ser analisado no veículo;

adjust(), para realizar os reparos e a manutenção necessária; e

cleanup(), para realizar procedimentos de limpeza do veículo.

A aplicação Oficina define um objeto que recebe objetos da classe Veículo. Para cada veículo recebido, a oficina executa na seqüência os três métodos da classe Veículo. No entanto, não há como saber no momento da programação se a Oficina estará recebendo um automóvel ou uma bicicleta -- assim, o momento de decisão sobre qual método será aplicado só ocorrerá durante a execução do programa.

```
import java.util.*;
class Veiculo {
   public Veiculo() {
            System.out.print("Veiculo ");
   }
   public void checkList() {
            System.out.println("Veiculo.checkList");
   }
   public void adjust() {
            System.out.println("Veiculo.adjust");
   }
   public void cleanup() {
            System.out.println("Veiculo.cleanup");
   }
}

class Automovel extends Veiculo {
   public Automovel() {
            System.out.println("Automovel");
   }
   public void checkList() {
            System.out.println("Automovel.checkList");
   }
```

```
    public void adjust() {
            System.out.println("Automovel.adjust");
    }
    public void cleanup() {
            System.out.println("Automovel.cleanup");
    }
}

class Bicicleta extends Veiculo {
    public Bicicleta() {
            System.out.println("Bicicleta");
    }
    public void checkList() {
            System.out.println("Bicicleta.checkList");
    }
    public void adjust() {
            System.out.println("Bicicleta.adjust");
    }
    public void cleanup() {
            System.out.println("Bicicleta.cleanup");
    }
}
public class Oficina {
    Random r = new Random();

    public Veiculo proximo() {
            Veiculo v;
            int code = r.nextInt();
            if (code%2 == 0)
               v = new Automovel();
            else
               v = new Bicicleta();

            return v;
    }

    public void manter(Veiculo v) {
            v.checkList();
            v.adjust();
            v.cleanup();
    }

    public static void main(String[] args) {
            Oficina o = new Oficina();
            Veiculo v;

            for (int i=0; i<4; ++i) {
               v = o.proximo();
               o.manter(v);
            }
    }
}
```

Este trecho de código ilustra a utilização da classe Oficina, considerando que os métodos acima foram definidos para Veículo e para todas as suas classes derivadas. Um possível resultado da interpretação dessa aplicação é:

```
Veiculo Bicicleta
Bicicleta.checkList
Bicicleta.adjust
Bicicleta.cleanup
Veiculo Bicicleta
Bicicleta.checkList
Bicicleta.adjust
Bicicleta.cleanup
Veiculo Automovel
Automovel.checkList
Automovel.adjust
Automovel.cleanup
Veiculo Automovel
Automovel.checkList
Automovel.adjust
Automovel.cleanup
```

Alguns pontos a destacar com relação a esse exemplo:

8.5 Upcasting:

O método Oficina.proximo() realiza uma atribuição de um objeto Automóvel à variável (referência para objeto Veiculo) v quando o valor do número aleatório gerado é par (o resto da divisão inteira por 2 é igual a 0). Essa atribuição de um objeto de uma classe mais especializada para uma referência de uma classe ancestral é denominada **upcast**. Esse mesmo tipo de atribuição é realizado de Bicicleta para veículo quando o número aleatório gerado é ímpar.

Recuperando informação sobre um tipo

O uso de polimorfismo está intimamente relacionado ao mecanismo de *upcast*, onde parte da informação sobre um objeto torna-se inacessível - ou seja, informação é momentaneamente perdida. Esse processo é seguro do ponto de vista da orientação a objetos pois a interface da classe base nunca é maior que a interface da classe derivada.

Há situações onde é interessante recuperar a referência para o tipo original de um objeto, de modo a obter acesso à sua funcionalidade completa. Para tanto, o mecanismo de *downcast* precisa ser utilizado:

```
Ref_orig = (Tipo_orig) Ref_upcast;
```

O problema com *downcasting* é que é preciso verificar se o objeto que está tendo sua referência convertida é realmente do tipo especificado, ou caso contrário seria impossível garantir sua manipulação correta após a conversão.

Em Java, todas as operações de *downcasting* são verificadas através do mecanismo de *Run-Time Type Identification* (RTTI) suportado pela linguagem. Lembre-se que é possível, para qualquer objeto, obter a indicação de a qual classe ele pertence através do método getClass().

8.6 Uso de métodos abstratos:

Apesar de métodos da classe Veículo terem sido definidos, estes nunca são invocados nesse exemplo. (Se fossem, algo estaria errado.) Isso ilustra uma situação onde **métodos abstratos** poderiam ser utilizados, pois a definição do corpo desses métodos é de fato irrelevante. Ainda mais, se uma classe como essa só contém métodos abstratos, ela poderia ser implementada como uma **interface** Java.

É importante observar que, quando polimorfismo está sendo utilizado, o comportamento que será adotado por um método só será definido durante a execução. Embora em geral esse seja um mecanismo que facilite o desenvolvimento e a compreensão do código orientado a objetos, há algumas situações onde o resultado da execução pode ser não-intuitivo, como ilustra esse exemplo que usa polimorfismo em construtores.

8.7 Polimorfismo em construtores:

A invocação de métodos com ligação tardia abre uma possibilidade de invocar construtores cujo comportamento poderia ser diferenciado polimorficamente. A título de exemplo, considere o seguinte código:

```
abstract class Base {
   abstract void m();
   public Base() {
      System.out.println("Base: inicio construcao");
      m();
      System.out.println("Base: fim construcao");
   }
}

public class Derivada extends Base {
   int valor = 1;
   void m() {
      System.out.println("Derivada.m: " + valor);
   }
   public Derivada(int v) {
      System.out.println("Derivada: inicio construcao");
      valor = v;
      System.out.println("Derivada: fim construcao");
   }

   public static void main(String[] args) {
      new Derivada(10);
   }
}
```

envolvendo uma classe Derivada e uma classe Base cujo construtor invoca um método implementado na classe derivada e apresenta o seguinte resultado.

```
Base: inicio construcao
Derivada.m: 0
Base: fim construcao
Derivada: inicio construcao
Derivada: fim construcao
```

O resultado dessa execução pode ser explicado pela seqüência de ações que é obedecida para a construção de um objeto a partir do momento no qual seu construtor é invocado. O comportamento apresentado nesse exemplo pode ser diferente daquele intuitivamente esperado por um programador que esteja analisando um código onde esta situação. Em um programa de maior porte, pode levar a situações de erro de difícil detecção.

A recomendação que se faz com relação à utilização de métodos no corpo de construtores é

Não invoque métodos no corpo de construtores a menos que isto seja seguro.

Métodos seguros para invocação a partir de construtores são aqueles que não podem ser redefinidos.

Unidade 9 - Interfaces

9.1 Interface:

Uma interface Java é uma classe abstrata para a qual todos os métodos são implicitamente abstract e public, e todos os atributos são implicitamente static e final. Em outros termos, uma interface Java implementa uma "classe abstrata pura".

A sintaxe para a declaração de uma interface é similar àquela para a definição de classes, porém seu corpo define apenas assinaturas de métodos e constantes.

Uma interface estabelece uma espécie de contrato que é obedecido por uma classe. Quando uma classe implementa uma interface, garante-se que todas as funcionalidades especificadas pela interface serão oferecidas pela classe.

9.2 Declaração de interface:

A sintaxe para a definição de uma interface Java equivale àquela da definição de uma classe, apenas usando a palavra chave interface ao invés da palavra chave class.

Por exemplo, para definir uma Interface1 que declara um método void met1() a sintaxe é:

```
interface Interface1 {
  void met1();
}
```

A diferença entre uma classe abstrata e uma interface Java é que a interface obrigatoriamente não tem um "corpo" associado. Para que uma classe seja abstrata basta que ela seja assim declarada, mas a classe pode incluir atributos de objetos e definição de métodos, públicos ou não. Na interface, apenas métodos públicos podem ser declarados -- mas não definidos. Da mesma forma, não é possível definir atributos -- apenas constantes públicas.

Enquanto uma classe abstrata é "estendida" (palavra chave extends) por classes derivadas, uma interface Java é "implementada" (palavra chave implements) por outras classes.

9.3 Implementação de Interfaces:

Uma vez que interfaces estejam definidas, como são usadas?

Interfaces são especificações que são ou serão implementadas através de classes Java. Por exemplo, uma interface Abc definida como

```
interface Abc {
  void a(int i);
  int b(String s);
  String c();
}
```

Indica que qualquer classe implementando essa especificação oferece **pelo menos** os três métodos, exatamente com as assinaturas descritas. Não é necessário saber como os métodos estão sendo implementados.

Uma classe Xyz que implementa a especificação de uma interface Abc é declarada com a sintaxe:

```
class Xyz implements Abc {
  // declarações, outros métodos
  void a(int valor) {
    ...
  }

  int b(String nome) {
    ...
  }

  String c( ) {
    ...
  }
}
```

É na classe que o corpo de cada um dos métodos da interface é efetivamente especificado, determinando como ocorre a implementação.

9.4 Definição de Constantes:

Outro uso de interfaces Java é para a definição de constantes que devem ser compartilhadas por diversas classes. Neste caso, a recomendação é implementar interfaces sem métodos, pois as classes que implementarem tais interfaces não precisam tipicamente redefinir nenhum método:

```
interface Coins {
  int
    PENNY = 1,
    NICKEL = 5,
    DIME = 10,
    QUARTER = 25,
    DOLAR = 100;
}

class CokeMachine implements Coins {
  int price = 3*QUARTER;
  // ...
}
```

Unidade 10 - Identificadores

10.1 Identificadores

Nomes de classes, variáveis e métodos devem ser identificadores válidos da linguagem. Esses identificadores são seqüências de caracteres Unicode.

As regras para a definição de identificadores são:

- Um nome pode ser composto por letras (minúsculas e/ou maiúsculas), dígitos e os símbolos **_** e **$**.
- Um nome não pode ser iniciado por um dígito (0 a 9).
- Letras maiúsculas são diferenciadas de letras minúsculas.
- Uma palavra-chave da linguagem Java não pode ser um identificador.

Além dessas regras obrigatórias, o uso da convenção padrão para identificadores Java torna a codificação mais uniforme e pode facilitar o entendimento de um código.

10.2 Palavras Reservadas em Java:

As palavras a seguir são de uso reservado em Java e não podem ser utilizadas como nomes de identificadores:

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

10.3 Convenção para identificadores:

Embora não seja obrigatório, o conhecimento e uso da seguinte convenção padrão para atribuir nomes em Java pode facilitar bastante a manutenção de um programa:

- Nomes de classes são iniciados por letras maiúsculas.
- Nomes de métodos, atributos e variáveis são iniciados por letras minúsculas.

- Em nomes compostos, cada palavra do nome é iniciada por letra maiúscula -- as palavras não são separadas por nenhum símbolo.

- Detalhes sobre as convenções de codificação sugeridas pelos projetistas da linguagem Java podem ser encontrados no documento Code Conventions for the JavaTM Programming Language

Unidade 11 - Expressões

public static void gc()11.1 Expressões:

Expressões em Java são terminadas por ";", podendo incluir expressões do tipo

operação aritmética,

operação lógica inteira,

operação lógica booleana,

condições,

atribuição,

retorno, e

operações sobre objetos.

11.1 Expressões Aritméticas:

Expressões aritméticas envolvem atributos, variáveis e/ou constantes numéricas (inteiras ou reais). Os operadores aritméticos definidos em Java incluem

soma (+);

subtração (-);

multiplicação (*);

divisão (/);

resto da divisão (%), apenas para operandos inteiros;

incremento (++), operador unário definido apenas para operandos inteiros, podendo ocorrer antes da variável (pré-incremento) ou após a variável (pós-incremento); e

decremento (--), operador unário definido apenas para operandos inteiros, podendo ocorrer antes da variável (pré-decremento) ou após a variável (pós-decremento).

11.2 Expressões Lógicas Inteiras:

Operações lógicas sobre valores inteiros atuam sobre a representação binária do valor armazenado, operando internamente bit a bit. Operadores desse tipo são:

- complemento (~), operador unário que reverte os valores dos bits na representação interna;

- OR bit-a-bit (|), operador binário que resulta em um bit 1 se pelo menos um dos bits na posição era 1;

- AND bit-a-bit (&), operador binário que resulta em um bit 0 se pelo menos um dos bits na posição era 0;

- XOR bit-a-bit (^), operador binário que resulta em um bit 1 se os bits na posição eram diferentes;

- deslocamento à esquerda (<<), operador binário que recebe a variável cujo conteúdo será deslocado e um segundo operando que especifica o número de posições a deslocar à esquerda;

- deslocamento à direita com extensão de sinal (>>), operador binário que recebe a variável cujo conteúdo será deslocado e um segundo operando que especifica o número de posições a deslocar à direita. Os bits inseridos à esquerda terão o mesmo valor do bit mais significativo da representação interna;

- deslocamento à direita com extensão 0 (>>>), operador binário que recebe a variável cujo conteúdo será deslocado e um segundo operando que especifica o número de posições a deslocar à direita. Os bits inseridos à esquerda terão o valor 0..

Por exemplo, queremos resolver o seguinte problema: Calcular a média de todos os alunos que cursaram uma disciplina X, a partir da leitura das notas da 1ª e 2ª prova, passando por um cálculo de média aritmética. Após a média calculada, devemos anunciar se o aluno foi aprovado ou reprovado por nota. Somente estão aprovados os alunos com média maior ou igual à 5,0.

11.3 Expressões Lógicas Booleanas:

As operações lógicas booleanas operam sobre valores booleanos. Operadores booleanos incluem:

- complemento lógico (!), operador unário que retorna true se o argumento é false ou retorna false se o argumento é true;

- OR lógico booleano (|), operador binário que retorna true se pelo menos um dos dois argumentos é true;

- OR lógico condicional (||), operador binário que retorna true se pelo menos um dos dois argumentos é true. Se o primeiro argumento já é true, o segundo argumento não é nem avaliado;

- AND lógico booleano (&), operador binário que retorna false se pelo menos um dos dois argumentos é false;

- AND lógico condicional (&&), operador binário que retorna false se pelo menos um dos dois argumentos é false. Se o primeiro argumento já é false, o segundo argumento não é nem avaliado;

- XOR booleano (^), operador binário que retorna true quando os dois argumentos têm valores lógicos distintos.

11.4 Condições:

Condições permitem realizar testes baseados nas comparações entre valores numéricos. Operadores condicionais incluem:

- maior (>), retorna true se o primeiro valor for exclusivamente maior que o segundo;

- maior ou igual (>=), retorna true se o primeiro valor for maior que ou igual ao segundo;

- menor (<), retorna true se o primeiro valor for exclusivamente menor que o segundo;

- menor ou igual (<=), retorna true se o primeiro valor for menor que ou igual ao segundo;

- igual (==), retorna true se o primeiro valor for igual ao segundo;

- diferente (!=), retorna true se o primeiro valor não for igual ao segundo.

Assim como C e C++, Java oferece o operador condicional ternário, ?:. Na expressão b ? s1 : s2, b é booleano (variável ou expressão). O resultado da expressão será o resultado da expressão (ou variável) s1 se b for verdadeiro, ou o resultado da expressão (ou variável) s2 se b for falso.

11.6 Atribuição:

O operador binário = atribui o valor da expressão do lado direito à variável à esquerdado operador. Os tipos da expressão e da variável devem ser compatíveis.

O operador de atribuição pode ser combinado com operadores aritméticos e lógicos, como em C e C++. Assim, a expressão

a += b

equivale a

a = a + b

11.5 Retorno:

Métodos em Java têm sua execução encerrada de duas maneiras possíveis. A primeira é válida quando um método não tem um valor de retorno (o tipo de retorno é void): a execução é encerrada quando o bloco do corpo do método chega ao final. A segunda alternativa encerra a execução do método através do comando return:

return;

sem valor de retorno.

return expressão;

o valor de retorno é o resultado da expressão.

11.6 Operações Sobre Objetos:

Criação de objeto

Dada uma classe Cls, é possível (em princípio) criar um objeto dessa classe usando o operador **new**:

```
Cls obj = new Cls();
```

A "função" à direita do operador new é um construtor da classe Cls.

Aplicação de métodos

A classe pode definir métodos que podem ser aplicados aos seus objetos. A aplicação de um método meth(), definido em uma classe Cls, a um objeto obj, construído a partir da especificação de Cls, se dá através da construção

```
obj.meth()
```

assumindo que a assinatura de meth determinasse que nenhum argumento é passado para o método.

Tomando como exemplo a variável System.out, o envio de dados para a tela se dá através da aplicação do método print (ou println(), para impressão seguida de muança de linha) tendo como argumento a variável ou *string* que se deseja imprimir, como em

```
System.out.println("Hello");
```

Verificação de tipo

Dado um objeto obj e uma classe Cls, é possível verificar dinamicamente (durante a execução do método) se o objeto pertence ou não à classe.

O operador instanceof retorna true se o objeto à esquerda do operador é da classe especificada à direita do operador. Assim,

```
obj instanceof Cls
```

retornaria true.

Unidade 12 - Fluxo de Controle de Execução

12.1 Comandos de Fluxo de Controle de Execução:

A ordem de execução normal de comandos em um método Java é seqüencial. Comandos de fluxo de controle permitem modificar essa ordem natural de execução através dos mecanismos de escolha ou de iteração.

12.2 Escolha:

if

A estrutura if permite especificar um comando (ou bloco de comandos) que deve apenas ser executado quando uma determinada condição for satisfeita:

```
if (condição) {
  bloco_comandos
}
```

Quando o bloco de comandos é composto por uma única expressão, as chaves que delimitam o corpo do bloco podem ser omitidas:

```
if (condição)
  expressão;
```

Embora a indentação do código não seja mandatória, é uma recomendação de boa prática de programação.

if else

if ... else permite expressar duas alternativas de execução, uma para o caso da condição ser verdadeira e outra para o caso da condição ser falsa

:

```
if (condição) {
  bloco_comandos_caso_verdade
}
else {
  bloco_comandos_caso_falso
}
```

switch

switch ... case também é um comando que expressa alternativas de execução, mas onde as condições estão restritas à comparação de uma variável inteira com valores constantes:

```
switch (variável) {
case valor1:
    bloco_comandos
```

```
    break;
case valor2:
    bloco_comandos
    break;
...
case valorn:
    bloco_comandos
    break;
default:
    bloco_comandos
}
```

12.3 Iteração:

while

while permite expressar iterações que devem ser executadas se e enquanto uma condição for verdadeira:

```
while (condição) {
  bloco_comandos
}
```

do while

do ... while também permite expressar iterações, mas neste caso pelo menos uma execução do bloco de comandos é garantida:

```
do {
  bloco_comandos
} while (condição);
```

for

O comando for permite expressar iterações combinando uma expressão de inicialização, um teste de condição e uma expressão de incremento:

```
for (inicialização; condição; incremento) {
  bloco_comandos
}
```

Embora Java não suporte o operador "," de C/C++, no comando for múltiplas expressões de inicialização e de incremento podem ser separadas por vírgulas.

break

continue e break são comandos que permitem expressar a quebra de um fluxo de execução. break já foi utilizado juntamente com *switch* para delimitar bloco de comandos de cada case. No corpo de uma iteração, a ocorrência do comando break interrompe a iteração, passando o controle para o próximo comando após o comando de iteração.

continue também interrompe a execução da iteração, mas apenas da iteração corrente. Após um continue, a condição de iteração é reavaliada e, se for verdadeira, o comando de iteração continua executando.

Em situações onde há diversos comandos de iteração aninhados, os comandos break e continue transferem o comando de execução para o ponto imediatamente após o bloco onde ocorrem. Se for necessário especificar transferência para o fim de outro bloco de iteração, os comandos break e continue rotulados podem ser utilizados:

```
label: {
    for (...; ...; ...) {
        ...
        while (...) {
            ...
            if (...)
                break label;
            ...
        }
        ...
    }
    ...
}
```

Unidade 13 - Arranjos em Java

13.1 Arranjos em Java:

Arranjos podem ser definidos para literais ou para objetos

Assim como objetos, são criados com o comando new:

Criar referência para um arranjo de inteiros

```
int array1[];
```

Criar espaço para armazenar 100 inteiros em um arranjo array1:

```
array1 = new int[100];
```

Combinando declaração e criação de espaço:

```
int array1[] = new int[100];
```

Arranjos podem ser criados com a especificação de algum conteúdo:

```
int array2[] = {2, 4, 5, 7, 9, 11, 13};
```

O acesso a elementos individuais de um arranjo é especificado através de um índice inteiro. O elemento inicial, assim como em C e C++, tem índice 0. Assim, do exemplo acima,

```
int x = array2[3];
```

faz com que a variável x receba o valor 7, o conteúdo da quarta posição.

O acesso a elementos do arranjo além do último índice permitido -- por exemplo, a array2[7] -- gera um erro em tempo de execução (uma *exceção*).

A dimensão de um arranjo pode ser obtida através do campo length presente em todos os arranjos. Assim, a expressão

```
int y = array2.length;
```

Faz com que y receba o valor 7, o número de elementos no arranjo..

Unidade 14 - Strings

14.1 Strings:

A classe *java.lang.String* é uma seqüência de caracteres imutável.

Representa uma cadeia de caracteres Unicode

Otimizada para ser lida, mas não alterada

Nenhum método de String modifica o objeto armazenado

Há duas formas de criar *Strings*

Através de construtores, metodos, fontes externas, etc:

```
String s1 = new String("Texto");
String s2 = objeto.getText(); // método de API
String s3 = coisa.toString();
```

Através de atribuição de um literal

```
String s3 = "Texto";
```

Strings criados através de literais são automaticamente armazenadas em um pool para possível reuso;

Mesmo objeto é reutilizado: comparação de *Strings* iguais criados através de literais revelam que se tratam do mesmo objeto;

Pool de strings

Como *Strings* são objetos imutáveis, podem ser reusados

Strings iguais criados através de literais são o mesmo objeto

```
String um = "Um";
String dois = "Um";
if (um == dois)
System.out.println("um e dois são um!");
```

Todos os blocos de texto abaixo são impressos

Mas Strings criados de outras formas não são

```
String tres = new String("Um");
if (um != tres)
System.out.println("um nao é três!");
```

Literais são automaticamente guardados no pool. Outros Strings podem ser acrescentados no pool usando *intern*():

```
quatro = tres.intern();
```

```
if (um == quatro)
System.out.println("quatro é um!");
```

Principais Métodos de String

Métodos que criam novos Strings:

String ***concat****(String s):* retorna a concatenação do String atual com outro passado como parâmetro

String ***replace****(char old, char new):* troca todas as ocorrências de um caractere por outro

String ***substring****(int start, int end):* retorna parte do String incluindo a posição inicial e excluíndo a final

String ***toUpperCase()*** e String ***toLowerCase():*** retorna o String em caixa alta e caixa baixa

respectivamente

Métodos para pesquisa

boolean **endsWith**(String) e **startsWith**(String)

int **indexOf**(String), int **indexOf**(String, int offset): retorna posição

char ***charAt****(int posição):* retorna caractere em posição

Outros métodos

char[] ***toCharArray()***: retorna o vetor de char correspondente ao String

int ***length()***: retorna o comprimento do String

14.2 StringBuffer:

A classe StringBuffer é uma seqüência de caracteres mutável

Representa uma cadeia de caracteres Unicode

Otimizada para ser alterada, mas não lida

StringBuffers podem ser criados através de seus construtores

```
StringBuffer buffer1 = new StringBuffer();
StringBuffer buffer2 = new StringBuffer("Texto inicial");
StringBuffer buffer3 = new StringBuffer(40);
```

Métodos de StringBuffer operam sobre o próprio objeto

StringBuffer ***append****(String s):* adiciona texto no final

StringBuffer ***insert****(int posição, String s):* insere na posição

void **setCharAt**(int posição, char c): substitui na posição

String ***toString():*** transforma o buffer em String para que possa ser lido

Exemplo:

```
StringBuffer buffer = new StringBuffer("H");
```

```
buffer.append("e").append("l").append("l").append("o");
System.out.println(buffer.toString());
```

Quando usar String e StringBuffer

Use String para manipular com valores constantes

- Textos carregados de fora da aplicação
- Valores literais
- Textos em geral que não serão modificados intensivamente

Use StringBuffer para alterar textos

- Acrescentar, concatenar, inserir, etc.

Prefira usar StringBuffer para construir Strings

- Concatenação de strings usando "+" é extremamente cara: um novo objeto é criado em cada fase da compilação apenas para ser descartado em seguida

Use ***StringBuffer.append***() e, no final, transforme o resultado em String

14.3 java.util.StringTokenizer:

Classe utilitária que ajuda a dividir texto em tokens

- Recebe um String e uma lista de tokens
- Usa um Enumeration para iterar entre os elementos

Exemplo:

```
String regStr = "Primeiro,Segundo,Terceiro,Quarto";
StringTokenizer tokens =
new StringTokenizer(regStr, ",");
String[] registros = null;
List regList = new ArrayList();
while (tokens.hasMoreTokens()) {
String item = tokens.nextToken();
regList.add(item);
}
int size = regList.size();
registros = (String[])regList.toArray(new String[size]);
```

Unidade 15 - Pacote

15.1 Pacote

No desenvolvimento de pequenas atividades ou aplicações, é viável manter o código e suas classes no diretório corrente. No entanto, para grandes aplicações é preciso organizar as classes de maneira a evitar problemas com nomes duplicados de classes, e localizar o código da classe de forma eficiente.

Em Java, a solução para esse problema está na organização de classes e interfaces em pacotes. Essencialmente, uma classe Xyz que pertence a um pacote nome.do.pacote tem o nome completo nome.do.pacote.Xyz e o compilador Java espera encontrar o arquivo Xyz.class em um subdiretório nome/do/pacote. Este, por sua vez, deve estar localizado sob um dos diretórios especificados na variável de ambiente CLASSPATH.

Unidade 16 - Núcleo de Funcionalidades

16.1 Núcleo de Funcionalidades

Tudo em Java está organizado em classes. Algumas classes são desenvolvidas pelo programador da aplicação e outras já estão disponíveis através do núcleo de funcionalidades da plataforma Java.

As classes que compõem o núcleo de funcionalidades Java estão organizadas em pacotes. Um **package** Java é um mecanismo para agrupar classes de finalidades afins ou de uma mesma aplicação. Além de facilitar a organização conceitual das classes, o mecanismo de pacotes permite localizar cada classe necessária durante a execução da aplicação. No entanto, a principal funcionalidade de um pacote Java é evitar a explosão do espaço de nome, ou seja, classes com o mesmo nome em pacotes diferentes podem ser diferenciadas pelo nome completo, pacote.classe.

Entre os principais pacotes oferecidos como parte do núcleo Java estão:

java.lang

java.util

java.io

java.awt

java.applet

java.net

Observe que esses nomes seguem a convenção Java, pela qual nomes de pacotes (assim como nomes de métodos) são grafados em letras minúsculas, enquanto nomes de classes têm a primeira letra (de cada palavra, no caso de nomes compostos) grafada com letra maiúscula.

Além dessas funcionalidades básicas, há também APIs definidas para propósitos mais específicos compondo a extensão padronizada ao núcleo Java.

16.2 java.lang:

O pacote java.lang contém as classes que constituem recursos básicos da linguagem, necessários à execução de qualquer programa Java.

Entre as classes desse pacote destacam-se:

Object

expressa o conjunto de funcionalidades comuns a todos os objetos Java;

Class e ClassLoader

representa classes Java e o mecanismo para carregá-las dinamicamente;

String e StringBuffer

permite a representação e a manipulação de *strings*, fixos ou modificáveis;

Math

contém a definição de métodos para cálculo de funções matemáticas (trigonométricas, logarítimicas, exponenciais, etc) e de constantes, tais como Math.E e Math.PI;

Boolean, Character, Byte, Short, Integer, Long, Float, Double

são classes *wrappers*, permitindo a manipulação de valores dos tipos literais da linguagem como se fossem objetos;

System, Runtime e Process

são classes que permitem interação da aplicação com o ambiente de execução;

Thread, Runnable, ThreadGroup

classes que dão suporte à execução de múltiplas linhas de execução;

Throwable, Error e Exception

classes que permitem a definição e manipulação de situações de erros e condições inesperadas de execução, tais como OutOfMemoryError, ArithmeticException (por exemplo, divisão inteira por zero) e ArrayIndexOutOfBoundsException (acesso a elemento de arranjo além da última posição ou antes da primeira posição).

Sub-pacotes relacionados incluem java.lang.ref, de referências a objetos, e o pacote java.lang.reflect, que incorpora funcionalidades para permitir a manipulação do conteúdo de classes, ou seja, identificação de seus métodos e campos. Observe que funções matemáticas são definidas em java.lang.Math, não em classes do pacote java.math, que define funcionalidades para manipular números inteiros e reais de precisão arbitrária.

16.3 System

A classe System oferece algumas funcionalidades básicas de interface com o sistema no qual a máquina virtual Java está executando. A documentação desta classe descreve:

Entre as facilidades fornecidas pela classe System estão a entrada padrão (in), a saída padrão (out), e os erros de saída de streams (err); alcance às "propriedades externamente definidas"; meios de carregar arquivos e bibliotecas; e um método para copiar rapidamente uma parcela de um array.

Entre as facilidades oferecidas estão o conjunto de variáveis correspondentes aos dispositivos padrões de entrada e saída, um método de classe para obter uma leitura de relógio com precisão de milissegundos, long CurrentTimeMillis(), e um método de classe para encerrar a execução da máquina virtual Java, void exit(int status), onde por convenção um valor de status diferente de zero indica alguma situação anormal de término da execução.

Uma classe intimamente relacionada a System é a classe Runtime, para a qual existe um único objeto representando o ambiente no qual a máquina virtual Java está executando. Através desta classe é possível disparar novos processos através do método exec().

16.4 Utilitários de Propósito Genérico:

O pacote java.util oferece, além de estruturas de dados, algumas classes utilitárias de propósito genérico.

16.5 Date

Permite representar uma data e hora usando uma representação interna em milissegundos decorridos desde a meia-noite (GMT) de 01 de janeiro de 1970. O construtor default cria um objeto com a data e hora corrente. Para obter o valor associado a esse objeto, o método long getTime() é utilizado.

16.6 Random

Implementa um gerador de números pseudo-aleatórios, com métodos double nextDouble() e float nextFloat() gerando valores uniformemente distribuídos entre 0,0 e 1,0; long nextLong() e int nextInt() geram valores inteiros uniformemente distribuídos ao longo da faixa de valores possíveis. O método double nextGaussian() retorna um valor com distribuição normal (média 0, desvio padrão 1).

A semente do gerador de números pseudo-aleatórios é obtida a partir do tempo corrente. Outro valor pode ser especificado no construtor ou com o método setSeed() -- para um mesmo valor inicial, uma mesma seqüência de números será gerada.

Unidade 17 - Entrada e Saída

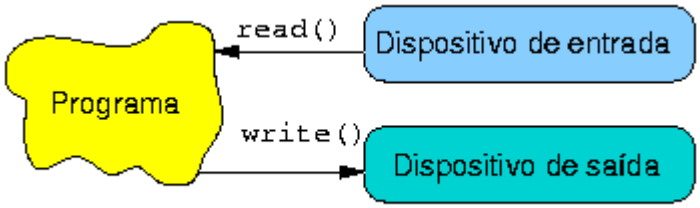
Tópicos da Unidade

ENTRADA E SAÍDA
ARQUIVOS BINÁRIOS
ARQUIVOS TEXTO
BUFFERS
FILTROS
DISPOSITIVOS PADRÕES DE ENTRADA E SAÍDA
ARQUIVOS
SERIALIZAÇÃO
ATIVIDADE JAVA.IO

16.7 Crítica ou Consistência de Dados 12.1:

Por entrada e saída subentende-se o conjunto de mecanismos oferecidos para que um programa executando em um computador consiga respectivamente obter e fornecer informação de dispositivos externos ao ambiente de execução, composto pelo processador e memória principal.

De forma genérica, havendo um dispositivo de entrada de dados habilitado, o programa obtém dados deste dispositivo através de uma operação `read()`. Similarmente, um dado pode ser enviado para um dispositivo de saída habilitado através de uma operação `write()`.



A manipulação de entrada e saída de dados em Java é suportada através de classes do pacote `java.io`. Essas classes oferecem as funcionalidades para manipular a entrada e saída de *bytes*, adequadas para a transferência de dados binários, e para manipular a entrada e saída de caracteres, adequadas para a transferência de textos.

Como a velocidade de operação de dispositivos de entrada e saída é várias ordens de grandeza mais lenta que a velocidade de processamento, *buffers* são tipicamente utilizados para melhorar a eficiência dessas operações de leitura e escrita.

Outra funcionalidade associada à transferência de dados está relacionada à conversão de formatos, tipicamente entre texto e o formato interno de dados binários. Essa e outras funcionalidades são suportadas através do oferecimento de filtros que podem ser agregados aos objetos que correspondem aos mecanismos elementares de entrada e saída.

Dois dispositivos básicos de entrada e saída presentes em qualquer computador de propósito geral são o teclado (entrada) e o monitor (saída). Esses dispositivos já são habilitados pelo sistema operacional no início da execução de qualquer programa, sendo denominados dispositivos padrões.

Outro dispositivo de entrada e saída de vital importância é disco, manipulado pelo sistema operacional e por linguagens de programação através do conceito de arquivos.

Sendo Java uma linguagem de programação orientada a objetos, seria de se esperar que, além das funcionalidades usuais de entrada e saída, ela oferecesse também alguma funcionalidade para transferência de objetos. O mecanismo de serialização suporta essa funcionalidade.

Arquivos Binários 12.2:

Fontes de dados manipuladas como seqüências de bytes são tratadas em Java pela classe ***InputStream*** e suas classes derivadas. Similarmente, saídas de seqüências de bytes são tratadas por ***OutputStream*** e suas classes derivadas.

InputStream

A classe abstrata `InputStream` oferece a funcionalidade básica para a leitura de um byte ou de uma seqüência de bytes a partir de alguma fonte. Os principais métodos dessa classe incluem:

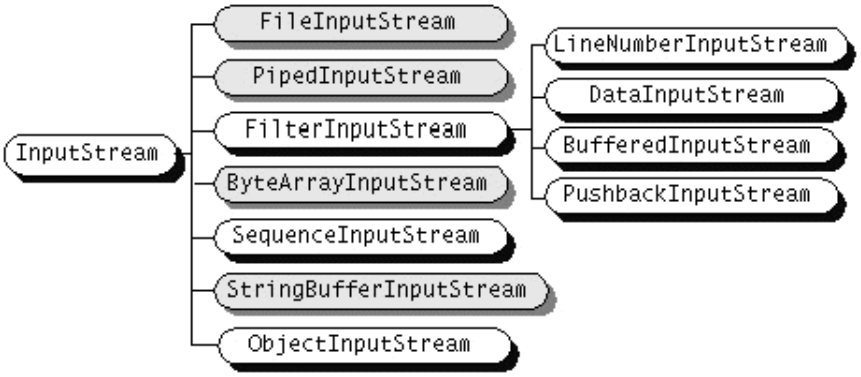
Method Summary

`int`	**`available`**`()` Returns the number of bytes that can be read (or skipped over) from this input stream without blocking by the next caller of a method for this input stream.
`void`	**`close`**`()` Closes this input stream and releases any system resources associated with the stream.
`abstract int`	**`read`**`()` Reads the next byte of data from the input stream.
`int`	**`read`**`(byte[] b)` Reads some number of bytes from the input stream and stores them into the buffer array `b`.
`int`	**`read`**`(byte[] b, int off, int len)` Reads up to `len` bytes of data from the input stream into an array of bytes.
`long`	**`skip`**`(long n)` Skips over and discards `n` bytes of data from this input stream.

Os métodos `read(...)` oferecem a funcionalidade básica de leitura de bytes. O valor de retorno para esses métodos é um inteiro, que pode ser o byte lido (para o caso do método sem argumentos) ou o número de bytes lidos (para os métodos que fazem a leitura para um arranjo de bytes). Em qualquer caso, um valor de retorno `-1` indica que o final do arquivo foi atingido.

Além dos métodos indicados, esta classe tem métodos para lidar com a posição de leitura no arquivo (`mark()` e `reset()`), se essa funcionalidade for suportada (o que pode ser consultado pelo método `markSupported()`).

Classes derivadas de InputStream



As principais subclasses de ***InputStream***, já oferecidas como parte do pacote `java.io`, são:

java.io.ByteArrayInputStream

Valores são originários de um arranjo de bytes;

java.io.FileInputStream

Bytes são originários de um arquivo; usualmente, usado em conjunto com `BufferedInputStream` e `DataInputStream`;

java.io.FilterInputStream

Oferece as definições necessárias para filtrar dados de um `InputStream`. Útil através de alguma de suas classes derivadas:

java.io.BufferedInputStream

Lê transparentemente grandes volumes de bytes, armazenando-os em um buffer interno, melhorando a eficiência de operação para fontes de dados lentas;

java.io.DataInputStream

Permite a leitura de representações binárias dos tipos primitivos de Java, oferecendo métodos tais como `readBoolean()`, `readChar()`, `readDouble` e `readInt()`. É uma implementação da interface *`DataInput`*.

java.io.PushbackInputStream

Oferece métodos `unread()` que permitem repor um ou mais bytes de volta à sua fonte, como se eles não tivessem sido lidos;

java.io.ObjectInputStream

Oferece método `readObject()` para a leitura de objetos que foram serializados para um `ObjectOutputStream`;

java.io.PipedInputStream

Oferece a funcionalidade de leitura de um *pipe* de bytes cuja origem está associada a um objeto `PipedOutputStream`;

java.io.SequenceInputStream

Oferece a funcionalidade para concatenar dois ou mais objetos `InputStream`; o construtor especifica os objetos que serão concatenados e, automaticamente, quando o fim do primeiro objeto é alcançado os bytes passam a ser obtidos do segundo objeto.

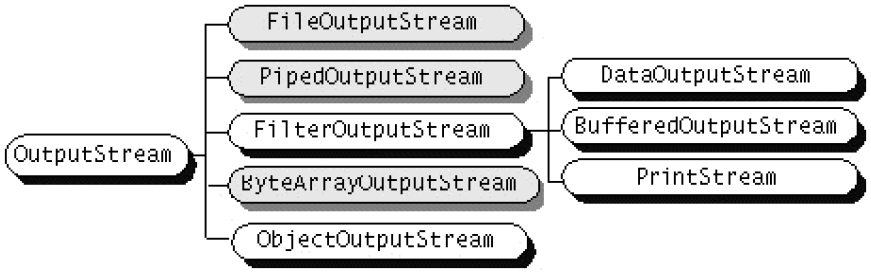
OutputStream

A classe abstrata `OutputStream` oferece a funcionalidade básica para a transferência seqüencial de bytes para algum destino. Os métodos desta classe são:

Method Summary	
`void`	**`close`**`()` Closes this output stream and releases any system resources associated with this stream.
`void`	**`flush`**`()` Flushes this output stream and forces any buffered output bytes to be written out.
`void`	**`write`**`(byte[] b)` Writes `b.length` bytes from the specified byte array to this output stream.
`void`	**`write`**`(byte[] b, int off, int len)` Writes `len` bytes from the specified byte array starting at offset `off` to this output stream.
`abstract void`	**`write`**`(int b)` Writes the specified byte to this output stream.

Os métodos `write(...)` oferecem a funcionalidade básica para escrita de bytes para o destino. O método `flush()` força a saída para o *stream* destino de qualquer byte cuja escrita esteja pendente (por exemplo, armazenado em um buffer interno).

Classes derivadas de OutputStream



As principais subclasses de *`OutputStream`* no pacote `java.io` são:

java.io.ByteArrayOutputStream

Facilidades para escrever para um arranjo de bytes interno, que cresce de acordo com a necessidade e pode ser acessado posteriormente através do método `toByteArray()` ou `toString()`;

java.io.FileOutputStream

Facilidades para escrever em arquivos, usualmente utilizadas em conjunção com as classes `BufferedOutputStream` e `DataOutputStream`;

java.io.FilterOutputStream

Definição de funcionalidades básicas para a filtragem de saída de dados, implementadas em alguma de suas classes derivadas:

java.io.BufferedOutputStream

Armazena bytes em um buffer interno até que o buffer esteja cheio ou o método `flush()` seja invocado;

java.io.DataOutputStream

Permite escrever valores de variáveis de tipos primitivos de Java em um formato binário portátil. É uma implementação da interface *`DataOutput`*.;

java.io.PrintStream

Oferece métodos para apresentar representações textuais dos valores de tipos primitivos Java, através de métodos `print()` e `println()`;

java.io.ObjectOutputStream

Permite armazenar a representação de um objeto serializável em um `OutputStream`;

java.io.PipedOutputStream

Implementa a origem de um *pipe* de bytes, que serão lidos a partir de um objeto PipedInputStream.

Arquivos Texto 12.3:

Aplicações típicas de entrada e saída envolvem a manipulação de arquivos contendo caracteres. Em Java, a especificação de funcionalidades para manipular esse tipo de arquivo estão contidas nas classes abstratas *`Reader`* (e suas classes derivadas) e *`Writer`* (e suas classes derivadas), respectivamente para leitura e para escrita de caracteres.

Reader

A classe `Reader` equivale à classe `InputStream`, com a diferença que ela é voltada para a manipulação de caracteres ao invés da manipulação de bytes. Ela oferece, entre outros, os métodos:

read

public int read() throws IOException

Read a single character. This method will block until a character is available, an I/O error occurs, or the end of the stream is reached.

Returns:

The character read, as an integer in the range 0 to 16383 (`0x00-0xffff`), or -1 if the end of the stream has been reached

Throws: IOException

If an I/O error occurs

read

public int read(char cbuf[]) throws IOException

Read characters into an array. This method will block until some input is available, an I/O error occurs, or the end of the stream is reached.

Parameters:

cbuf - Destination buffer

Returns:

The number of bytes read, or -1 if the end of the stream has been reached

Throws: IOException

If an I/O error occurs

ready

public boolean ready() throws IOException

Tell whether this stream is ready to be read.

Returns:

True if the next read() is guaranteed not to block for input, false otherwise. Note that returning false does not guarantee that the next read will block.

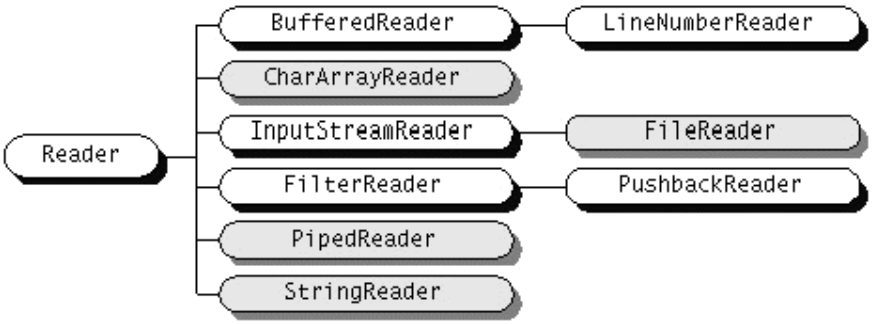
Throws: IOException

If an I/O error occurs

Classes derivadas de Reader

Como `Reader` é uma classe abstrata, não é possível criar diretamente objetos dessa classe. É preciso criar objetos de uma de suas subclasses concretas para ter acesso à funcionalidade especificada por `Reader`.

A documentação da API de Java mostra a seguinte hierarquia de classes derivadas de `Reader`:



BufferedReader incorpora um *buffer* a um objeto `Reader`. LineNumberReader estende essa classe para fazer leitura de um arquivos por linhas, mantendo um registro do número de linhas processadas.

CharArrayReader e StringReader permitem fazer a leitura desde arranjos de caracteres e de objetos `String`, respectivamente.

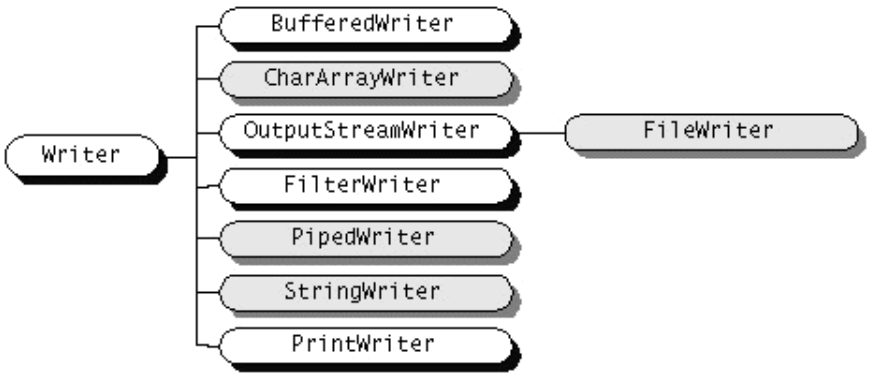
FilterReader é uma classe abstrata para representar classes que implementam algum tipo de filtragem sobre o dado lido. Sua classe derivada, PushbackReader, incorpora a possibilidade de retornar um caráter lido de volta à sua fonte.

InputStreamReader implementa a capacidade de leitura a partir de uma fonte que fornece bytes, traduzindo-os adequadamente para caracteres. Sua classe derivada, FileReader, permite associar essa fonte de dados a um arquivo.

PipedReader faz a leitura a partir de um objeto `PipedWriter`, estabelecendo um mecanismo de comunicação inter-processos (no caso de Java, entre *threads* de uma mesma máquina virtual).

Writer

A classe abstrata `Writer` é a classe análoga a `OutputStream`, que é a raiz para as classes de saída de bytes. Ela é a raiz das classes de saída de dados do tipo caráter:



Os métodos da classe `Writer` incluem o método `write()` (em cinco versões), o método `close()` e o método `flush()`:

write

public void write(int c) throws IOException

Write a single character. The character to be written is contained in the 16 low-order bits of the given integer value; the 16 high-order bits are ignored.

Subclasses that intend to support efficient single-character output should override this

method.

Throws: IOException

If an I/O error occurs

---

public void write(char cbuf[]) throws IOException

Write an array of characters.

Parameters:

cbuf - Array of characters to be written

Throws: IOException

If an I/O error occurs

---

public abstract void write(char cbuf[],
int off,
int len) throws IOException

Write a portion of an array of characters.

Parameters:

cbuf - Array of characters

off - Offset from which to start writing characters

len - Number of characters to write

Throws: IOException

If an I/O error occurs

---

public void write(String str) throws IOException

Write a string.

Parameters:

str - String to be written

Throws: IOException

If an I/O error occurs

---

public void write(String str,
int off,

int len) throws IOException

Write a portion of a string.

Parameters:

str - A String

off - Offset from which to start writing characters

len - Number of characters to write

Throws: IOException

If an I/O error occurs

flush

public abstract void flush() throws IOException

Flush the stream. If the stream has saved any characters from the various write() methods in a buffer, write them immediately to their intended destination. Then, if that destination is another character or byte stream, flush it. Thus one flush() invocation will flush all the buffers in a chain of Writers and OutputStreams.

Throws: IOException

If an I/O error occurs

close

public abstract void close() throws IOException

Close the stream, flushing it first. Once a stream has been closed, further write() or flush() invocations will cause an IOException to be thrown. Closing a previously-closed stream, however, has no effect.

Throws: IOException

If an I/O error occurs

Classes derivadas de Writer

Assim como para a classe `Reader`, as funcionalidades da classe `Writer` são implementadas através de suas subclasses concretas. A hierarquia de classes derivadas de `Writer` no pacote `java.io` é definida como:

class java.io.Writer

class java.io.BufferedWriter

class java.io.CharArrayWriter

class java.io.FilterWriter

class java.io.OutputStreamWriter

class java.io.FileWriter

class java.io.PipedWriter

class java.io.PrintWriter

class java.io.StringWriter

BufferedWriter incorpora um *buffer* a um objeto `Writer`, permitindo uma melhoria de eficiência de escrita ao combinar várias solicitações de escrita de pequenos blocos em uma solicitação de escrita de um bloco maior.

CharArrayWriter e StringWriter permitem fazer a escrita em arranjos de caracteres e em objetos `StringBuffer`, respectivamente.

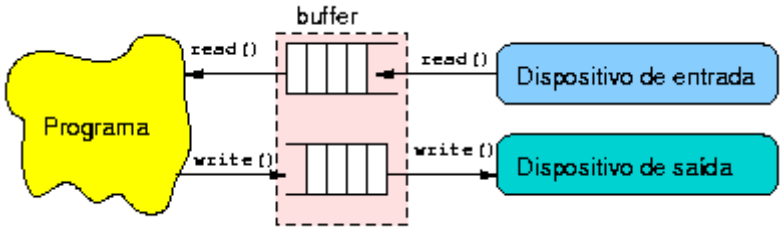
FilterWriter é uma classe abstrata para representar classes que implementam algum tipo de filtragem sobre o dado escrito.

OutputStreamWriter implementa a capacidade de escrita tendo como destino uma seqüência de bytes, traduzindo-os adequadamente desde os caracteres de entrada. Sua classe derivada, FileWriter, permite associar esse destino de dados a um arquivo.

PipedWriter faz a escrita para um objeto `PipedReader`, estabelecendo um mecanismo de comunicação inter-processos (no caso, entre *threads*).

Buffers 12.4:

Essencialmente, um *buffer* é uma entidade intermediária entre duas ou mais outras entidades que produzem e consomem elementos. Internamente, um *buffer* contém uma área de memória que é utilizada para armazenamento temporário de elementos que foram produzidos mas ainda não foram consumidos.



Essa capacidade de um *buffer* manter temporariamente parte dos dados produzidos mas não consumidos torna-o um mecanismo adequado para compatibilizar a operação de entidades operando em diferentes velocidades. Além de oferecer transparentemente as mesmas funcionalidades de leitura e escrita de dados, um *buffer* normalmente oferece operações adicionais que permitem consultar se há dados disponíveis para leitura no *buffer* de entrada ou para solicitar que os dados presentes no *buffer* de saída sejam enviados imediatamente para o dispositivo de saída.

Em Java, há quatro classes que adicionam as funcionalidades de *buffering* às classes fundamentais de entrada e saída. Para classes que manipulam bytes, as classes `BufferedInputStream` e `BufferedOutputStream` são oferecidas. Para a manipulação de entrada e saída de texto, as classes `BufferedReader` e `BufferedWriter`.

BufferedInputStream

A classe `BufferedInputStream` adiciona as funcionalidades de *buffering* a um *InputStream*. Além de implementar os métodos básicos para leitura de bytes especificados naquela classe, BufferedInputStream adiciona o método available(), que retorna um valor inteiro indicando o número de bytes disponível para leitura no *buffer* interno.

BufferedOutputStream

A classe `BufferedOutputStream` adiciona as funcionalidades de *buffering* a um *OutputStream*, implementando as funcionalidades previstas naquela classe.

BufferedReader

Suponha que ao invés de ler o arquivo de texto por caracteres fosse desejado ler uma linha do arquivo de cada vez. Para tanto, é necessário ler os caracteres um a um e agregá-los em um *buffer* até encontrar o caráter de fim de linha. Então esse conjunto de caracteres pode ser transformado em uma *string* e retornada como resultado desse método.

Esse tipo de funcionalidade adicional (ou *decoração*) já é suportado através de objetos da classe `BufferedReader`. Um dos construtores para essa classe é

BufferedReader

public BufferedReader(Reader in)

Create a buffering character-input stream that uses a default-sized buffer.

Parameters:

in - A Reader

e o método que especifica a funcionalidade acima é:

readLine

public String readLine() throws IOException

Read a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.

Returns:

A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached

Throws: IOException

If an I/O error occurs

Assim, caso se deseje ler linhas de texto de um arquivo, é necessário decorar um `FileReader` com a funcionalidade de um `BufferedReader`:

```
String filename;
// filename recebe o nome de algum arquivo texto...
FileReader fr = new FileReader(filename);
BufferedReader bfr = new BufferedReader(fr);
```

A partir desse ponto, pode-se ser linhas do arquivo especificado por `filename` através do método `bfr.readLine()`.

A classe `LineNumberReader`, derivada de `BufferedReader`, mantém um registro interno relativo ao número de linhs lidas. Esse registro é manipulado através dos métodos `int getLineNumber()` e `void setLineNumber(int)`.

BufferedWriter

A classe `BufferedWriter`, além de implementar o contrato especificado na classe *`Writer`*, adiciona um método `void newLine()` que permite lidar com separadores de linha de maneira independente de plataforma.

Filtros 12.5:

Um filtro é uma entidade que "envolve" um dispositivo de entrada ou saída realizando, em adição às funcionalidades básicas de transferência, alguma possibilidade de transformação dos dados sendo transferidos.

Em Java, há diversas classes que disponibilizam filtros que podem ser combinados de acordo com a necessidade da aplicação. Por exemplo, a classe `DataInputStream` é uma extensão de *`InputStream`* que implementa a interface *`DataInput`*, oferecendo um conjunto de métodos para ler grupos de bytes que representam tipos primitivos da linguagem Java. Similarmente, a classe `DataOutputStream` estende *`OutputStream`* e implementa *`DataOutput`*.

Outro filtro derivado de `OutputStream` é oferecido pela classe `PrintStream`, para converter tipos básicos para uma seqüência de bytes imprimível. Similarmente, a classe `PrintWriter` estende a classe `Writer` para converter tipos básicos em seqüências de caracteres. Essas duas classes oferecem os métodos `print()` e `println()` com assinaturas contemplando todos os tipos básicos da linguagem.

As funcionalidades básicas associadas a filtros de entrada e saída estão definidas nas classes *`FilterInputStream`*, *`FilterOutputStream`*, *`FilterReader`* e *`FilterWriter`*, das quais os filtros concretos são derivados.

DataInput

A interface `DataInput` é implementada pelas classes `DataInputStream` e `RandomAccessFile`. Os métodos especificados nesta interface possibilitam a interpretação de uma seqüência de bytes como um tipo primitivo da linguagem Java. Adicionalmente, é também possível interpretar a seqüência de bytes como um objeto `String`.

Em geral, métodos dessa interface geram uma exceção `EOFException` se for solicitada a leitura além do final do arquivo. Em outras situações de erro de leitura, a exceção `IOException` é gerada.

A documentação da API Java especifica os seguintes métodos para essa interface:

Method Summary

`boolean`	**`readBoolean`**`()` Reads one input byte and returns `true` if that byte is nonzero, `false` if that byte is zero.
`byte`	**`readByte`**`()` Reads and returns one input byte.
`char`	**`readChar`**`()` Reads an input `char` and returns the `char` value.
`double`	**`readDouble`**`()` Reads eight input bytes and returns a `double` value.
`float`	**`readFloat`**`()` Reads four input bytes and returns a `float` value.
`void`	**`readFully`**`(byte[] b)` Reads some bytes from an input stream and stores them into the buffer array `b`.
`void`	**`readFully`**`(byte[] b, int off, int len)` Reads `len` bytes from an input stream.
`int`	**`readInt`**`()` Reads four input bytes and returns an `int` value.
`String`	**`readLine`**`()` Reads the next line of text from the input stream.
`long`	**`readLong`**`()` Reads eight input bytes and returns a `long` value.
`short`	**`readShort`**`()` Reads two input bytes and returns a `short` value.
`int`	**`readUnsignedByte`**`()` Reads one input byte, zero-extends it to type `int`, and returns the result, which is therefore in the range `0` through `255`.

int	**readUnsignedShort**() Reads two input bytes and returns an int value in the range 0 through 65535.
String	**readUTF**() Reads in a string that has been encoded using a modified UTF-8 format.
int	**skipBytes**(int n) Makes an attempt to skip over n bytes of data from the input stream, discarding the skipped bytes.

DataOutput

A interface DataOutput é implementada pelas classes DataOutputStream e RandomAccessFile. Métodos especificados nesta interface permitem a conversão de tipos primitivos Java para seqüências de bytes.

Em geral, se há algum impedimento para realizar a saída da seqüência de bytes, a exceção IOException é gerada.

A documentação da API Java especifica os seguintes métodos para essa interface:

Method Summary

void	**write**(byte[] b) Writes to the output stream all the bytes in array b.
void	**write**(byte[] b, int off, int len) Writes len bytes from array b, in order, to the output stream.
void	**write**(int b) Writes to the output stream the eight low-order bits of the argument b.
void	**writeBoolean**(boolean v) Writes a boolean value to this output stream.
void	**writeByte**(int v) Writes to the output stream the eight low- order bits of the argument v.
void	**writeBytes**(String s) Writes a string to the output stream.
void	**writeChar**(int v) Writes a char value, wich is comprised of two bytes, to the output stream.
void	**writeChars**(String s) Writes every character in the string s, to the output stream, in order, two bytes per character.
void	**writeDouble**(double v) Writes a double value, which is comprised of eight bytes, to the output stream.

void	**writeFloat**(float v) Writes a float value, which is comprised of four bytes, to the output stream.
void	**writeInt**(int v) Writes an int value, which is comprised of four bytes, to the output stream.
void	**writeLong**(long v) Writes an long value, which is comprised of four bytes, to the output stream.
void	**writeShort**(int v) Writes two bytes to the output stream to represent the value of the argument.
void	**writeUTF**(String str) Writes two bytes of length information to the output stream, followed by the Java modified UTF representation of every character in the string s.

Dispositivos Padrões de Entrada e Saída 12.6:

Os dispositivos padrões de entrada e saída de dados são abstrações de arquivos oferecidas pelos sistemas operacionais para interação básico com usuários. São tipicamente três dispositivos:

Entrada padrão

Associada inicialmente ao dispositivo teclado, oferece uma interface equivalente a um arquivo seqüencial exclusivamente de leitura.

Saída padrão

Associada inicialmente ao dispositivo monitor, oferece uma interface equivalente a um arquivo seqüencial exclusivamente de escrita.

Saída padrão de erros

É inicialmente associada ao mesmo dispositivo da saída padrão.

Em Java, esses três dispositivos são representados respectivamente pelas variáveis *in*, *out* e *err* da classe *System*.

System

A classe System oferece algumas funcionalidades básicas de interface com o sistema no qual a máquina virtual Java está executando. A documentação desta classe descreve:

Among the facilities provided by the System class are standard input, standard output, and error output streams; access to externally defined "properties"; a means of loading files and libraries; and a utility method for quickly copying a portion of an array.

Entre as facilidades oferecidas estão o conjunto de variáveis correspondentes aos dispositivos padrões de entrada e saída, um método de classe para obter uma leitura de relógio com precisão de milissegundos, `long CurrentTimeMillis()`, e um método de classe para encerrar a execução da máquina virtual Java, `void exit(int status)`, onde por convenção um valor de `status` diferente de zero indica alguma situação anormal de término da execução.

Uma classe intimamente relacionada a `System` é a classe `Runtime`, para a qual existe um único objeto representando o ambiente no qual a máquina virtual Java está executando. Através desta classe é possível disparar novos processos através do método `exec()`.

System.in

A outra variável de `System` associada aos dispositivos padronizados corresponde à entrada padrão:

in

public static final InputStream in

The "standard" input stream. This stream is already open and ready to supply input data. Typically this stream corresponds to keyboard input or another input source specified by the host environment or user.

Da mesma forma que `out`, a variável é estática, pública e final. No entanto, esse é um objeto da classe `java.io.InputStream`, uma classe abstrata que especifica, entre vários métodos, o método `read()`:

read

public abstract int read() throws IOException

Reads the next byte of data from this input stream. The value byte is returned as an `int` in the range `0` to `255`. If no byte is available because the end of the stream has been reached, the value `-1` is returned. This method blocks until input data is available, the end of the stream is detected, or an exception is thrown.

A subclass must provide an implementation of this method.

Returns:

the next byte of data, or `-1` if the end of the stream is reached.

Throws: IOException

if an I/O error occurs.

read

public int read(byte b[]) throws IOException

Reads up to `b.length` bytes of data from this input stream into an array of bytes.

Parameters:

b - the buffer into which the data is read.

Returns:

the total number of bytes read into the buffer, or `-1` is there is no more data because the end of the stream has been reached.

Throws: IOException

if an I/O error occurs.

No entanto, esse método de entrada é muito primitivo, sendo dificilmente usado dessa maneira para obter entradas de um usuário através do teclado. A forma usual para realizar essa tarefa envolve o uso de outras funcionalidades de entrada e saída oferecidas por Java, em particular o uso de *buffers*.

Assim, para ter um objeto associado ao dispositivo padrão de entrada que permite ler uma linha completa através do teclado, a seguinte fórmula é utilizada:

```
BufferedReader in
  = new BufferedReader(new InputStreamReader(System.in));
```

Objetos da classe *BufferedReader* oferecem um método `readLine()` que retorna uma *string* de texto, que neste exemplo seria lida a partir do teclado.

System.out

A variável `out` está associada à apresentação de caracteres na saída padrão, ou seja, na tela do monitor. Na documentação da API Java encontra-se:

out

public static final PrintStream out

Uma simples análise dessa declaração mostra que:

`out` é uma variável de classe e, portanto, pode ser acessada na forma `System.out`;

essa variável é pública;

a variável é final; e

a variável é do tipo *`PrintStream`*.

System.err

A variável `System.err` tem a mesma declaração apresentada por *`System.out`*. Assim, é um objeto do tipo `PrintStream`, ao qual os métodos `print()` e `println()` podem ser aplicados..

Arquivos 12.7:

Um arquivo é uma abstração utilizada por sistemas operacionais para uniformizar a interação entre o ambiente de execução e os dispositivos externos a ele em um computador. Tipicamente, a interação de um programa com um dispositivo através de arquivos passa por três etapas:

abertura ou criação de um arquivo,

transferência de dados, e

fechamento do arquivo.

Em Java, a classe `File` permite representar arquivos nesse nível de abstração. Um dos construtores desta classe recebe como argumento uma *string* que pode identificar, por exemplo, o nome de um arquivo em disco. Os métodos desta classe permitem obter informações sobre o arquivo -- por exemplo, `exists()`, `canRead()`, `canWrite()`, `length()` e `lastModified()` -- e realizar operações sobre o arquivo como um todo, como em `delete()` e `deleteOnExit()`.

Observe que as funcionalidades de transferência seqüencial de dados a partir de ou para um arquivo não é suportada pela classe `File`, mas sim pelas classes `FileInputStream`, `FileOutputStream`, *`FileReader`* e *`FileWriter`*. Todas estas classes oferecem pelo menos um construtor que recebe como argumento um objeto da classe `File` e implementam os métodos básicos de transferência de dados suportados respectivamente por *`InputStream`*, *`OutputStream`*, *`Reader`* e *`Writer`*.

Para aplicações que necessitam manipular arquivos de forma não seqüencial (ou "direta"), envolvendo avanços ou retrocessos arbitrários na posição do arquivo onde ocorrerá a transferência, Java oferece a classe *`RandomAccessFile`*.

FileReader

A classe `FileReader` representa objetos que são arquivos de texto no qual apenas leitura seqüencial pode ocorrer. Essa classe apresenta apenas os métodos construtores:

FileReader

public FileReader(String fileName) throws FileNotFoundException

public FileReader(File file) throws FileNotFoundException

```
public FileReader(FileDescriptor fd)
```

Arquivos de acesso direto

`Reader`, `Writer`, `InputStream` e `OutputStream` oferecem funcionalidades para a leitura ou escrita de seqüências de elementos básicos (caracteres ou bytes).

Há, no entanto, categorias de aplicações que necessitam acesso a pontos arbitrários de um arquivo de dados. Por exemplo, a leitura de registros de um banco de dados não deve ser realizada seqüencialmente.

Para suportar essa funcionalidade, o pacote `java.io` define a classe `RandomAccessFile`. Um dos construtores dessa classe toma como argumentos duas *strings*, a primeira com o nome do arquivo e a segunda com o modo de operação ("r" para leitura apenas, "rw" para leitura e escrita).

Os métodos para a manipulação da posição corrente do ponteiro no arquivo são:

`void seek(long pos)`

Seleciona a posição (em relação ao início do arquivo) para a próxima operação de leitura ou escrita;

`long getFilePointer()`

Retorna a posição atual do ponteiro do arquivo.

Além disso, o método `long length()` retorna o tamanho do arquivo em bytes.

Para a manipulação de conteúdo do arquivo, todos os métodos especificados pelas interfaces *`DataInput`* e *`DataOutput`* (por exemplo, `readInt()` e `writeInt()` para ler e escrever valores do tipo primtivo `int`) são implementados por essa classe.

Serialização 12.8:

O processo de serialização de objetos permite converter a representação de um objeto em memória para uma seqüência de bytes que pode então ser enviada para um `ObjectOutputStream`, que por sua vez pode estar associado a um arquivo em disco ou a uma conexão de rede, por exemplo.

Por exemplo, para serializar um objeto da classe `java.util.Hashtable`, armazenando seu contéudo em um arquivo em disco, a seqüência de passos é:

```
// criar/manipular o objeto
   Hashtable dados = new Hashtable();
   ...
// definir o nome do arquivo
   String filename = ...;
// abrir o arquivo de saída
   File file = new File(filename);
```

```
    if(!file.exists()){
            file.createNewFile();
    }
  FileOutputStream out = new FileOutputStream(file);
// associar ao arquivo o ObjectOutputStream
  ObjectOutputStream s = new ObjectOutputStream(out);
// serializar o objeto
  s.writeObject(dados);
```

O processo inverso, a desserialização, permite ler essa representação de um objeto a partir de um `ObjectInputStream` usando o método `readObject()`:

```
  FileInputStream in = new FileInputStream(file);
  ObjectInputStream s = new ObjectInputStream(in);
  dados = (Hashtable) s.readObject();
```

Serializable

Objetos de classes para os quais são previstas serializações e desserializações devem implementar a interface `java.io.Serializable`, que não define nenhum método ou campo -- é uma interface *marker*, servindo apenas para registrar a semântica de serialização.

A serialização de um objeto deve ser implementada pela definição de um método `writeObject()`, com assinatura:

```
private void writeObject(java.io.ObjectOutputStream out)
   throws IOException
```

Tipicamente, esse método realiza qualquer preparação necessária para a serialização e invoca o método `defaultWriteObject()` da classe `ObjectOutputStream`.

A desserialização é implementada pelo método `readObject()` com assinatura:

```
private void readObject(java.io.ObjectInputStream in)
   throws IOException, ClassNotFoundException;
```

Esse método tipicamente invoca `defaultReadObject()` da classe `ObjectInputStream` e então realiza as ações complementares àquelas realizadas em `writeObject()`, tais como leitura de dados adicionais e inicialização de variáveis.

Serialização é um processo transitivo, ou seja, subclasses serializáveis de classes serializáveis são automaticamente incorporadas à representação serializada do objeto raiz. Para que o processo de desserialização possa operar corretamente, todas as classes envolvidas no processo devem ter um construtor *default* (sem argumentos).

Externalizable

Caso deseje-se controle total sobre o processe de serialização e desserialização de objetos (ao invés de usar os métodos `defaultWriteObject()` e `defaultReadObject()`), a interface `java.io.Externalizable` (uma extensão de `Serializable`) deve ser utilizada. Essa interface define os métodos:

writeExternal

public abstract void writeExternal(ObjectOutput out) throws IOException

The object implements the writeExternal method to save its contents by calling the methods of DataOutput for its primitive values or calling the writeObject method of ObjectOutput for objects, strings and arrays.

Throws: IOException

Includes any I/O exceptions that may occur

readExternal

public abstract void readExternal(ObjectInput in) throws

IOException, ClassNotFoundException

The object implements the readExternal method to restore its contents by calling the methods of DataInput for primitive types and readObject for objects, strings and arrays. The readExternal method must read the values in the same sequence and with the same types as were written by writeExternal.

Throws: IOException

Includes any I/O exceptions that may occur

Throws: ClassNotFoundException

If the class for an object being restored cannot be found.

Atividade java.io 12.9:

A classe `Pessoa` foi definida contendo os seguintes atributos:

```
public class Pessoa {
```

```
    private String nome;
    private String telefone;
    private String email;
    // construtores e métodos seguem...
}
```

Complemente a definição da classe de modo que:

Seja possível ler os dados para a definição de um objeto dessa classe a partir de um arquivo texto com três linhas, a primeira com um nome, a segunda com um telefone e a terceira com um e-mail, como em

Pedro da Silva

(11) 3456-7890

daSilva@pedro.net.br

Seja possível ler/escrever dados de um objeto dessa classe na forma serializada;

Seja possível obter cada item de dado de um objeto dessa classe separadamente; e

Seja possível apresentar os dados obtidos no formato texto para um arquivo especificado (se nenhum for especificado, a saída padrão, `System.out`, deve ser usada).

Use esses métodos no método `main()` de uma classe `TestePessoa` que cria um objeto `Pessoa` a partir dos dados descritos em um arquivo texto cujo nome foi especificado na linha de comando para `TestePessoa`.

Os dados são então salvos em outro arquivo, com o mesmo nome mas extensão `.ser`, em formato serializado. Esse arquivo é lido para outro objeto da classe `Pessoa`.

Finalmente, os dados dos dois objetos são apresentados na tela, assim como o resultado da aplicação do método `equals()` a esses objetos.

Bufferizar ou não bufferizar, eis a questão

Desenvolva uma classe que permita receber um nome de arquivo como argumento e que faça a leitura desse arquivo de duas formas, sem e com *buffer*. Desenvolva um método `main()` que permita realizar a leitura das duas formas e que apresente os tempos utilizados para fazer cada tipo de leitura na saída padrão.

Lendo do teclado

Desenvolva uma classe com métodos para ler seqüências de caracteres do teclado, traduzindo-as para tipos básicos da linguagem Java. Para a atividade, desenvolva pelo menos dois métodos, `getString()` e `getInteger()`, para obter *strings* genéricas e valores inteiros, respectivamente. Inclua na classe um método `main()` que permita testar esses métodos.

Manter agenda

Desenvolva uma classe que mantém um arquivo com dados de uma agenda pessoal (nome, e-mail e telefone, um registro por linha com campos separados por algum caráter) e que ofereça as seguintes funcionalidades:

`void inclui(String nome, String email, String fone)`

acrescenta uma linha ao final do arquivo com os dados fornecidos.

`String[] le(String nome)`

obtém todas as linhas do arquivo que contêm a *string* especificada em `nome`.

`String[] le()`

obtém todos os dados da agenda.

`void mostra(String linha)`

recebe uma linha no formato especificado para a sua agenda e apresente-o na saída padrão de forma mais inteligível -- por exemplo, um campo por linha com rótulos, com registros separados por linhas tracejadas.

Inclua um método `main()` que permita testar essas funcionalidades.

Acerte o número

Nessa atividade você deve implementar em Java um jogo simples. O objetivo do jogo é acertar o valor do número que foi gerado aleatoriamente pelo computador e guardado em um lugar seguro.

O jogo começa com o computador recebendo do usuário um argumento na linha de comando especificando um valor inteiro. Esse valor inteiro corresponde ao limite superior da faixa de valores que o usuário deverá tentar acertar. Se nenhum valor for especificado, o máximo de 1000 será assumido.

Na seqüência, o programa solicita que o usuário entre valores inteiros (entre 0 e o máximo valor) através do teclado. Após cada valor inserido pelo usuário, o computador retorna uma indicação se a tentativa foi correta (o que encerra o jogo, indicando quantas tentativas foram necessárias) ou errada; neste caso, deve ainda indicar se o valor está acima ou abaixo do número secreto, e então solicitar a entrada de um novo valor do usuário.

Unidade 18 - Tratamento de Exceções

Tópicos da Unidade

16.8 Tratamento de Exceções 13.1:

Uma **exceção** é um sinal que indica que algum tipo de condição excepcional ocorreu durante a execução do programa. Assim, exceções estão associadas a condições de erro que não tinham como ser verificadas durante a compilação do programa.

As duas atividades associadas à manipulação de uma exceção são:

geração

A sinalização de que uma condição excepcional (por exemplo, um erro) ocorreu, e

captura

A manipulação (tratamento) da situação excepcional, onde as ações necessárias para a recuperação da situação de erro são definidas.

Para cada exceção que pode ocorrer durante a execução do código, um bloco de ações de tratamento (um *exception handler*) deve ser especificado. O compilador Java verifica e enforça que toda exceção "não-trivial" tenha um bloco de tratamento associado.

O mecanismo de manipulação de exceções em Java, embora apresente suas particularidades, teve seu projeto inspirado no mecanismo equivalente de C++, que por sua vez foi inspirado em Ada. É um mecanismo adequado à manipulação de erros síncronos, para situações onde a recuperação do erro é possível.

Que métodos geram exceção?

Usando métodos que geram exceções

Capturando e tratando exceções

Propagando exceções

A hierarquia de exceções

Capturando várias exceções

Gerando exceções

Que Métodos Geram Exceção? 13.2:

A documentação da API Java indica, para cada método, se ele pode gerar ou não uma ou mais exceções. Por exemplo, na documentação do método `java.lang.Integer.parseInt(String)` lê-se:

parseInt

public static int parseInt(String s)

throws NumberFormatException

Parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits, except that the first character may be an ASCII minus sign `'-'` to indicate a negative value.

Parameters:

s - a string.

Returns:

the integer represented by the argument in decimal.

Throws: NumberFormatException

if the string does not contain a parsable integer.

Nesse método, se o argumento (uma string) não representar adequadamente um valor inteiro decimal (ou seja, composto exclusivamente pelos caracteres dígitos de 0 a 9, opcionalmente com um sinal '-' à frente), não será possível gerar um valor inteiro correspondente e uma exceção de nome `NumberFormatException` será gerada.

Outro exemplo: a documentação da classe `java.io.InputStream` indica que o método `read()` também pode gerar uma exceção:

read

public int read(byte b[]) throws IOException

Reads up to `b.length` bytes of data from this input stream into an array of bytes.

The `read` method of `InputStream` calls the `read` method of three arguments with the arguments `b`, `0`, and `b.length`.

Parameters:

b - the buffer into which the data is read.

Returns:

the total number of bytes read into the buffer, or `-1` if there is no more data because the end of the stream has been reached.

Throws: IOException

if an I/O error occurs.

Usando Métodos que Geram Exceções 13.3:

Java exige que toda exceção seja capturada e tratada. Como toda regra tem sua exceção (sem trocadilhos), exceções relacionadas a condições "triviais" de execução (derivadas da classe `java.lang.RuntimeException`) não precisam, mas podem, ser capturadas e tratadas. Há também situações nas quais o erro *não deve* ser tratado, o que é indicado pela classe `java.lang.Error` e suas derivadas.

O não-tratamento de uma exceção que deve ser tratada é acusado por um erro de compilação.

Exemplo

Considere o seguinte código, cujo objetivo é ler um inteiro a partir da entrada padrão de erros:

```
1  import java.io.*;
2
3  public class ConvInt {
4      public String leLinha() {
5          byte[] tB = new byte[20];
6          System.in.read(tB);
7          String taTudo = new String(tB);
8          String taLimpo = taTudo.trim();
9          return taLimpo;
10     }
11
12     public int leInt() {
13         String s = leLinha();
14         return Integer.parseInt(s);
15     }
16
17     public static void main(String[] args) {
18         ConvInt ci = new ConvInt();
19         System.out.print("Entre inteiro: ");
20         int valor = ci.leInt();
21         System.out.println("Valor lido foi: " + valor);
22     }
```

```
23 }
```

A tentativa de compilação dessa classe gera a seguinte mensagem de erro: `ConvInt.java:6: Exception java.io.IOException must be caught, or it must be declared in the throws clause of this method.`

```
        System.in.read(tB);
                       ^
1 error
```

`System.in` é um objeto da classe `InputStream`, cujo método `read()` invocado na linha 6 pode gerar a exceção `IOException`, como visto. Essa é uma condição de erro que exige tratamento, pois caso contrário o programa prosseguiria em um estado potencialmente inconsistente. Portanto, o compilador Java não gera o *bytecode* para essa classe enquanto um tratamento para esse erro não for incluído no código.

Observe, no entanto, que o método `parseInt()` invocado na linha 14 também pode gerar uma exceção. Porém, o compilador Java não reclamou dessa condição, pois essa exceção está incluída no grupo de `RuntimeException`s.

Capturando e Tratando Exceções 13.4:

A sinalização da exceção é propagada a partir do bloco de código onde ela ocorreu através de toda a pilha de invocações de métodos até que a exceção seja **capturada** por um bloco manipulador de exceção. Eventualmente, se tal bloco não existir em nenhum ponto da pilha de invocações de métodos, a sinalização da exceção atinge o método `main()`, fazendo com que o interpretador Java apresente uma mensagem de erro e aborte sua execução.

Blocos try-catch

A captura e o tratamento de exceções em Java se dá através da especificação de blocos `try`, `catch` e `finally`, definidos através destas mesmas palavras reservadas da linguagem. Um comando `try/catch/finally` obedece à seguinte sintaxe:

```
try {
  // código que inclui comandos/invocações de métodos
  // que podem gerar uma situação de exceção.
}
catch (XException x) {
  // bloco de tratamento associado à condição de
  // exceção XException ou a qualquer uma de suas
  // subclasses, identificada aqui pelo objeto
  // com referência x
```

```
}
catch (YException y) {
   // bloco de tratamento para a situação de exceção
   // YException ou a qualquer uma de suas subclasses
}
finally {
   // bloco de código que sempre será executado após
   // o bloco try, independentemente de sua conclusão
   // ter ocorrido normalmente ou ter sido interrompida
}
```

Os blocos não podem ser separados por outros comandos -- um erro de sintaxe seria detectado pelo compilador Java neste caso. Cada bloco `try` pode ser seguido por zero ou mais blocos `catch`, onde cada bloco `catch` refere-se a uma única exceção.

O bloco `finally` é sempre executado. Em geral, ele inclui comandos que liberam recursos que eventualmente possam ter sido alocados durante o processamento do bloco `try` e que podem ser liberados, independentemente de a execução ter encerrado com sucesso ou ter sido interrompida por uma condição de exceção. A presença desse bloco é opcional.

Exemplo

Para tornar o exemplo anterior funcional, é preciso tratar a exceção que pode ser gerada na execução da linha 6. Para tanto, uma possibilidade é substituir essa linha por

```
            try {
               System.in.read(tB);
            }
            catch (IOException e) {
         System.err.println(e);
            }
```

Nesse caso, o tratamento dado para a ocorrência do erro foi simplesmente apresentar na saída padrão de erros uma mensagem associada à exceção ocorrida. O programa completo fica:

```
1  import java.io.*;
2
3  public class ConvInt2 {
4     public String leLinha() {
5        byte[] tB = new byte[20];
```

```
6       try {
7           System.in.read(tB);
8       }
9       catch (IOException e) {
10          System.err.println(e);
11      }
12      String tS = new String(tB).trim();
13      return tS;
14  }
15
16  public int leInt() {
17      String s = leLinha();
18      return Integer.parseInt(s);
19  }
20
21  public static void main(String[] args) {
22      ConvInt2 ci = new ConvInt2();
23      System.out.print("Entre inteiro: ");
24      int valor = ci.leInt();
25      System.out.println("Valor lido foi: " + valor);
26  }
27 }
```

Propagando Exceções 13.5:

Embora toda exceção deva ser tratada, nem sempre é possível tratar uma exceção no mesmo escopo do método cuja invocação gerou a exceção. Nessas situações, é possível propagar a exceção para um nível acima na pilha de invocações.

Para tanto, o método que está deixando de capturar e tratar a exceção faz uso da cláusula `throws` na sua declaração:

```
void métodoQueNãoTrataExceção() throws Exception {
  invoca.métodoQuePodeGerarExceção();
}
```

Nesse caso, `métodoQueNãoTrataExceção()` reconhece que em seu corpo há a possibilidade de haver a geração de uma exceção mas não se preocupa em realizar o tratamento dessa exceção em seu escopo. Ao contrário, ele repassa essa responsabilidade para o método anterior na pilha de chamadas.

Eventualmente, também o outro método pode repassar a exceção adiante. Porém, pelo menos no `main()` as exceções deverão ser tratadas.

Exemplo

A seguir, o exemplo da aquisição de um valor inteiro a partir da entrada padrão é reapresentado com o uso da cláusula `throws`. Nesse caso, `throws` tem que ser declarado em dois métodos. Primeiro no método `leLinha`, que invoca `read()` e não faz o tratamento de `IOException`. Segundo, no método leInt, que pode receber o `IOException` de `leLinha` mas também não faz seu tratamento. Para ilustrar, nesse segundo caso inclui-se também a declaração de que o método pode também gerar a exceção `NumberFormatException`.

```
1  import java.io.*;
2
3  public class ConvInt3 {
4    public String leLinha()
5            throws IOException {
6        byte[] tB = new byte[20];
7        System.in.read(tB);
8        String taTudo = new String(tB);
9        String taLimpo = taTudo.trim();
10       return taLimpo;
11 }
12
13 public int leInt()
14           throws IOException, NumberFormatException {
15       String s = leLinha();
16       return Integer.parseInt(s);
17 }
18
19 public static void main(String[] args) {
20       ConvInt3 ci = new ConvInt3();
21       System.out.print("Entre inteiro: ");
22          try {
23             int valor = ci.leInt();
24             System.out.println("Valor lido foi: " + valor);
25          }
26          catch (IOException ioe) {
27             System.err.println("Erro de leitura");
28          }
```

```
29          catch (NumberFormatException nfe) {
30              System.err.println("Erro de formato");
31          }
32  }
33 }
```

A Hierarquia de Exceções 13.6:

Uma exceção em Java é um objeto. A classe raiz de todas as exceções é `java.lang.Throwable`. Apenas objetos dessa classe ou de suas clsses derivadas podem ser gerados, propagados e capturados através do mecanismo de tratamento de exceções.

A classe `Throwable` tem duas subclasses:

`java.lang.Exception`

É a raiz das classes derivadas de `Throwable` que indica situações que a aplicação pode querer capturar e realizar um tratamento que permita prosseguir com o processamento.

`java.lang.Error`

É a raiz das classes derivadas de `Throwable` que indica situações que a aplicação não deve tentar tratar. Usualmente indica situações anormais, que não deveriam ocorrer.

Exemplos de exceções já definidas em Java incluem:

`java.lang.ArithmeticException`

indica situações de erros em processamento aritmético, tal como uma divisão inteira por 0.

`java.lang.NumberFormatException`

indica que tentou-se a conversão de uma *string* para um formato numérico, mas seu conteúdo não representava adequadamente um número para aquele formato. É uma subclasse de `java.lang.IllegalArgumentException`.

`java.lang.ArrayIndexOutOfBoundsException`

indica a tentativa de acesso a um elemento de um arranjo fora de seus limites -- ou o índice era negativo ou era maior ou igual ao tamanho do arranjo. É uma subclasse de `java.lang.IndexOutOfBounds`, assim como a classe `java.lang.StringIndexOutOfBounds`.

`java.lang.NullPointerException`

indica que a aplicação tentou usar `null` onde uma referência a um objeto era necessária -- invocando um método ou acessando um atributo, por exemplo.

`java.lang.ClassNotFoundException`

indica que a aplicação tentou carregar uma classe mas não foi possível encontrá-la.

`java.io.IOException`

indica a ocorrência de algum tipo de erro em operações de entrada e saída. É a raiz das classes `java.io.EOFException` (fim de arquivo ou *stream*), `java.io.FileNotFoundException` (arquivo

especificado não foi encontrado) e `java.io.InterruptedIOException` (operação de entrada ou saída foi interrompida), entre outras.

Entre os erros definidos em Java, subclasses de `java.lang.Error`, estão `java.lang.StackOverflowError` e `java.lang.OutOfMemoryError`. São situações onde não é possível uma correção a partir de um tratamento realizado pelo próprio programa que está executando.

Como exceções são objetos, podem incluir atributos. A classe `Throwable` inclui o registro do estado atual da pilha d chamdas de métodos e uma *string* associada a uma mensagem de erro, que pode ser obtida para qualquer exceção através do método `Throwable.getMessage()`.

Outras exceções podem inclui ainda outras informações que possam vir a ser relevantes para o seu tratamento. Por exemplo, `java.io.InterruptedIOException` inclui um atributo inteiro, `bytesTransferred`, para indicar quantos bytes foram transferidos antes da interrupção da operação ocorrer.

Capturando Várias Exceções 13.7:

Quando apresentou-se a estrutura do bloco try-catch, pode ter ficado a impressão que para cada exceção possivelmente gerada em um bloco `try` um bloco `catch` deveria ser declarado.

Graças ao polimorfismo, não é esse o caso. A cláusula `catch` sabe que deve receber como argumento um objeto da classe `Throwable`. Como um objeto da classe derivada "é-um" objeto da superclasse, qualquer objeto-exceção da classe especificada ou de suas classes derivadas podem ser capturadas por um `catch`.

Por exemplo, uma especificação

```
catch (Exception e) {...}
```

implica na captura de todas as exceções, uma vez que a classe `Exception` é a raiz de todas as exceções.

Em geral, se dois blocos `catch`es envolvem duas classes relacionadas na hierarquia de exceções, a superclasse deve vir após a classe derivada na seqüência de blocos `catch`. Portanto, se um bloco `catch` for incluído com argumento `Exception`, como acima, esse deve ser o último bloco na seqüência de *catches*.

Gerando Exceções 13.8:

A sinalização da ocorrência de uma condição excepcional de execução se dá através do comando `throw`, como ilustrado em:

```
public double calculaDivisao (int numerador, int denominador)
      throws Exception {
  if (denominador == 0)
     throw new Exception("Dividiu por zero");
  return ((double) numerador) / denominador;
}
```

Definição de novas exceções

É possível para uma aplicação definir e sinalizar suas próprias exceções. Por exemplo, considere que fosse importante para uma aplicação diferenciar a condição de divisão inteira por zero de outras condições de exceções artiméticas. Neste caso, uma classe DivideByZeroException poderia ser criada:

```
public class DivideByZeroException
        extends ArithmeticException {
  public DivideByZeroException() {
    super("O denominador na divisão inteira tem valor zero");
  }
}
```

A classe contém apenas um construtor, que invoca o construtor da superclasse especificando uma mensagem descrevendo a exceção. Esta é a mensagem que será retornada pelo método `getMessage()` quando invocado sobre um objeto dessa classe.

A sinalização da exceção se dá através do comando `throw`:

```
// Método que usa divisão inteira e pode gerar a exceção:
public double calculaDivisao (int numerador, int denominador)
      throws DivideByZeroException {
  if (denominador == 0)
    throw new DivideByZeroException();

  return (double) numerador / denominador;
}
```

Repassando a exceção

É possível usar o comando `throw` para repassar a sinalização da condição de exceção adiante, como em

```
public void usaDivisao()
      throws DivideByZeroException {
try {
   d = calculaDivisao(x, y);
}
catch (DivideByZeroException dbze) {
   d = 0;
```

```
    throw dbze;
}
```

Nesse caso, a informação associada à exceção (como o seu ponto de origem, registrado internamente no atributo do objeto que contém a pilha de invocações) é preservada. Caso fosse desejado mudar essa informação, uma nova exceção poderia ser gerada ou, caso a exceção seja a mesma, o método `fillInStackTrace()` poderia ser utilizado, como em

```
...
   throw dbze.fillInStackTrace();
...
```

Unidade 19 - Exercícios para programação orientada a objetos

1) Faça um programa em Java que leia 3 números e calcule a sua média.

O programa deve ser modularizado. Utilize uma função para calcular e retornar a média dos números. Essa função deve receber os três números como parâmetro.

2) Crie um programa em Java que leia o raio de uma esfera (do tipo real) e passe esse valor para a função volumeEsfera. Essa função deve calcular o volume da esfera na tela e retornar o seu valor.

Para o cálculo do volume deve ser usada a seguinte fórmula:

Volume = (4.0 / 3.0) * PI * raio3

3) Faça um programa em Java em que o usuário entre com um valor de base e um valor de expoente. O programa deve calcular a potência.

4) Quando um método construtor é chamado?

6) Qual a função de um método modificador numa classe? E de um método recuperador?

7) Analise o seguinte trecho de código e identifique se existe algum erro. Caso exista, explique qual o erro e como solucioná-lo.

```
//classe Retangulo
public class Retângulo{
        private double base, altura;
        //métodos
}

//classe calcula que cria objeto retangulo
public class CalculaRetangulo{
        public static void main(String args[]){
        Retangulo r = new Retangulo()
        r.base = 5.0;
        r.altura = 6.0
        }
}
```

8) A seguinte classe tem dois métodos static. Programe outra classe chamada UsaCalculos.java e demonstre como esses métodos static podem ser chamados dessa classe.

```
public class Calculos{
        public static int potencia(int base, int exp){
        int pote=1;
        for (int i=1;i<=exp;i++)
        pote = pote * base;
        return pote;
        }
        public static int fatorial(int nu){
        int fat=1;
        for (int i=1;i<=nu;i++)
        fat = fat * nu;
        return fat;
        }
}
```

Unidade 20 - Dicas Para Desenvolvimento dos Exercícios e Outros Projetos

1) Comece criando a classe de execução inicial (ou principal) do seu projeto / exercício, com os códigos como se as funcionalidades (métodos e demais propriedades) já existissem;

2) Depois disso, comece a pensar nos métodos que você está usando e pense de que maneira eles poderiam ficar agrupados ou se, pela definição do seu uso, já estariam todos agrupados em uma só classe;

3) Agora sim, crie as classes com os respectivos métodos para que seu projeto possa ser testado, atribuindo os modificadores de acesso e demais regras a esses métodos e propriedades;

4) Para testar seus exercícios usando informações fornecidas por um usuário através de entrada via teclado, você pode utilizar uma classe de Java que oferece dois métodos de fácil interação (entrada e saída): **JOptionPane;**

5) Essa classe JOptionPane possui dois métodos "static" bem fáceis de serem usados:
- String valor = JOptionPane.showInputDialog("Texto aqui");
- JOptionPane.showMessageDialog(null, "Texto de saída aqui");

6) A primeira opção exibe uma janela de diálogo com uma caixa de entrada de dados. Qualquer informação que você digitar nessa caixa de texto será recebida como uma String;

7) A segunda opção exibe uma janela de diálogo com o texto fornecido no segundo parâmetro. No primeiro parâmetro (null), informamos se essa janela de exibição será vinculada à outra janela já aberta antes, que não será o nosso caso.