Módulo 1

Introdução à Programação I

Lição 9

Trabalhando com Bibliotecas de Classe

Versão 1.0 - Jan/2007

Autor
Florence Tiu Balagtas

Equipe
Joyce Avestro
Florence Balagtas
Rommel Feria
Reginald Hutcherson
Rebecca Ong
John Paul Petines
Sang Shin
Raghavan Srinivas
Matthew Thompson

Necessidades para os Exercícios

Sistemas Operacionais Suportados

NetBeans IDE 5.5 para os seguintes sistemas operacionais:

- Microsoft Windows XP Profissional SP2 ou superior
- Mac OS X 10.4.5 ou superior
- Red Hat Fedora Core 3
- Solaris™ 10 Operating System (SPARC® e x86/x64 Platform Edition)

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

- Microsoft Windows 2000 Profissional SP4
- Solaris™ 8 OS (SPARC e x86/x64 Platform Edition) e Solaris 9 OS (SPARC e x86/x64 Platform Edition)
- Várias outras distribuições Linux

Configuração Mínima de Hardware

Nota: IDE NetBeans com resolução de tela em 1024x768 pixel

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	500 MHz Intel Pentium III workstation ou equivalente	512 MB	850 MB
Linux	500 MHz Intel Pentium III workstation ou equivalente	512 MB	450 MB
Solaris OS (SPARC)	UltraSPARC II 450 MHz	512 MB	450 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Série 1.8 GHz	512 MB	450 MB
Mac OS X	PowerPC G4	512 MB	450 MB

Configuração Recomendada de Hardware

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	1 GB
Linux	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	850 MB
Solaris OS (SPARC)	UltraSPARC IIIi 1 GHz	1 GB	850 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Series 1.8 GHz	1 GB	850 MB
Mac OS X	PowerPC G5	1 GB	850 MB

Requerimentos de Software

NetBeans Enterprise Pack 5.5 executando sobre Java 2 Platform Standard Edition Development Kit 5.0 ou superior (JDK 5.0, versão 1.5.0_01 ou superior), contemplando a Java Runtime Environment, ferramentas de desenvolvimento para compilar, depurar, e executar aplicações escritas em linguagem Java. Sun Java System Application Server Platform Edition 9.

- Para **Solaris**, **Windows**, e **Linux**, os arquivos da JDK podem ser obtidos para sua plataforma em http://java.sun.com/j2se/1.5.0/download.html
- Para **Mac OS X**, Java 2 Plataform Standard Edition (J2SE) 5.0 Release 4, pode ser obtida diretamente da Apple's Developer Connection, no endereço: http://developer.apple.com/java (é necessário registrar o download da JDK).

Para mais informações:
http://www.netbeans.org/community/releases/55/relnotes.html

Colaboradores que auxiliaram no processo de tradução e revisão

Alexandre Mori
Alexis da Rocha Silva
Aline Sabbatini da Silva Alves
Allan Wojcik da Silva
André Luiz Moreira
Andro Márcio Correa Louredo
Antoniele de Assis Lima
Antonio Jose R. Alves Ramos
Aurélio Soares Neto
Bruno da Silva Bonfim
Bruno dos Santos Miranda
Bruno Ferreira Rodrigues
Carlos Alberto Vitorino de Almeida
Carlos Alexandre de Sene
Carlos André Noronha de Sousa
Carlos Eduardo Veras Neves
Cleber Ferreira de Sousa
Cleyton Artur Soares Urani
Cristiano Borges Ferreira
Cristiano de Siqueira Pires
Derlon Vandri Aliendres
Fabiano Eduardo de Oliveira
Fábio Bombonato
Fernando Antonio Mota Trinta
Flávio Alves Gomes
Francisco das Chagas
Francisco Marcio da Silva
Gilson Moreno Costa
Givailson de Souza Neves
Gustavo Henrique Castellano
Hebert Julio Gonçalves de Paula
Heraldo Conceição Domingues
Hugo Leonardo Malheiros Ferreira
Ivan Nascimento Fonseca
Jacqueline Susann Barbosa
Jader de Carvalho Belarmino
João Aurélio Telles da Rocha
João Paulo Cirino Silva de Novais
João Vianney Barrozo Costa
José Augusto Martins Nieviadonski
José Leonardo Borges de Melo
José Ricardo Carneiro
Kleberth Bezerra G. dos Santos
Lafaiete de Sá Guimarães
Leandro Silva de Morais
Leonardo Leopoldo do Nascimento
Leonardo Pereira dos Santos
Leonardo Rangel de Melo Filardi
Lucas Mauricio Castro e Martins
Luciana Rocha de Oliveira
Luís Carlos André
Luís Octávio Jorge V. Lima
Luiz Fernandes de Oliveira Junior
Luiz Victor de Andrade Lima
Manoel Cotts de Queiroz
Marcello Sandi Pinheiro
Marcelo Ortolan Pazzetto
Marco Aurélio Martins Bessa
Marcos Vinicius de Toledo
Maria Carolina Ferreira da Silva
Massimiliano Giroldi
Mauricio Azevedo Gamarra
Mauricio da Silva Marinho
Mauro Cardoso Mortoni
Mauro Regis de Sousa Lima
Namor de Sá e Silva
Néres Chaves Rebouças
Nolyanne Peixoto Brasil Vieira
Paulo Afonso Corrêa
Paulo José Lemos Costa
Paulo Oliveira Sampaio Reis
Pedro Antonio Pereira Miranda
Pedro Henrique Pereira de Andrade
Renato Alves Félix
Renato Barbosa da Silva
Reyderson Magela dos Reis
Ricardo Ferreira Rodrigues
Ricardo Ulrich Bomfim
Robson de Oliveira Cunha
Rodrigo Pereira Machado
Rodrigo Rosa Miranda Corrêa
Rodrigo Vaez
Ronie Dotzlaw
Rosely Moreira de Jesus
Seire Pareja
Sergio Pomerancblum
Silvio Sznifer
Suzana da Costa Oliveira
Tásio Vasconcelos da Silveira
Thiago Magela Rodrigues Dias
Tiago Gimenez Ribeiro
Vanderlei Carvalho Rodrigues Pinto
Vanessa dos Santos Almeida
Vastí Mendes da Silva Rocha
Wagner Eliezer Roncoletta

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

- **Brasil** – Tiago Flach
- **Guiné Bissau** – Alfredo Cá, Bunene Sisse e Buon Olossato Quebi – ONG Asas de Socorro

Coordenação do DFJUG

- **Daniel deOliveira** – JUGLeader responsável pelos acordos de parcerias
- **Luci Campos -** Idealizadora do DFJUG responsável pelo apoio social
- **Fernando Anselmo -** Coordenador responsável pelo processo de tradução e revisão, disponibilização dos materiais e inserção de novos módulos
- **Regina Mariani -** Coordenadora responsável pela parte jurídica
- **Rodrigo Nunes -** Coordenador responsável pela parte multimídia
- **Sérgio Gomes Veloso -** Coordenador responsável pelo ambiente JEDI™ (Moodle)

Agradecimento Especial

John Paul Petines – Criador da Iniciativa JEDI™
Rommel Feria – Criador da Iniciativa JEDI™

1. Objetivos

Nesta lição, abordaremos alguns conceitos básicos da Programação Orientada a Objetos ou POO. Mais adiante, discutiremos o conceito de classes e objetos e como usar as classes e seus membros. Comparação, conversão e *casting* de objetos, também serão vistos. Por enquanto, o foco será o uso das classes já definidas nas bibliotecas Java e, posteriormente, discutiremos como criar nossas próprias classes.

Ao final desta lição, o estudante será capaz de:

- Explicar o que é Programação Orientada a Objetos e alguns dos seus conceitos
- Diferenciar entre classes e objetos
- Diferenciar atributos e métodos de objeto de atributos e métodos de classe
- Explicar o que são métodos, como invocá-los e como enviar argumentos
- Identificar o escopo de um atributo
- Realizar conversões entre tipos de dados primitivos e entre objetos
- Comparar objetos e determinar suas classes

2. Introdução à Programação Orientada a Objeto

Programação Orientada a Objetos (POO) refere-se ao conceito de **objetos** como elemento básico das classes. O mundo físico é constituído por objetos tais como carro, leão, pessoa, dentre outros. Estes objetos são caracterizados pelas suas **propriedades** (ou **atributos**) e seus **comportamentos**.

Por exemplo, um objeto "carro" tem as propriedades, tipo de câmbio, fabricante e cor. O seu comportamento pode ser 'virar', 'frear' e 'acelerar'. Igualmente, podemos definir diferentes propriedades e comportamentos para um leão. Veja exemplos na **Tabela 1**.

Objeto	***Propriedades***	***Comportamentos***
carro	tipo de câmbio fabricante cor	virar frear acelerar
leão	peso cor apetite (faminto ou saciado) temperamento (dócil ou selvagem)	rugir dormir caçar

Tabela 1: Exemplos de objetos do mundo real

Com tais descrições, os objetos do mundo físico podem ser facilmente modelados como objetos de software usando as **propriedades como atributos** e os **comportamentos como métodos**. Estes atributos e métodos podem ser usados em softwares de jogos ou interativos para simular objetos do mundo real! Por exemplo, poderia ser um objeto de 'carro' numa competição de corrida ou um objeto de 'leão' num aplicativo educacional de zoologia para crianças.

3. Classes e Objetos

3.1. Diferenças entre Classes e Objetos

No mundo do computador, um **objeto** é um componente de software cuja estrutura é similar a um objeto no mundo real. Cada objeto é composto por um conjunto de **atributos** (propriedades) que são as variáveis que descrevem as características essenciais do objeto e, consiste também, num conjunto de **métodos** (comportamentos) que descrevem como o objeto se comporta. Assim, um objeto é uma coleção de atributos e métodos relacionados. Os atributos e métodos de um objeto Java são formalmente conhecidos como **atributos e métodos de objeto**, para distinguir dos atributos e métodos de classes, que serão discutidos mais adiante.

A **classe** é a estrutura fundamental na Programação Orientada a Objetos. Ela pode ser pensada como um gabarito, um protótipo ou, ainda, uma **planta** para a construção de um objeto. Ela consiste em dois tipos de elementos que são chamados **atributos** (ou propriedades) e **métodos**. Atributos especificam os tipos de dados definidos pela classe, enquanto que os métodos especificam as operações. Um objeto é uma **instância** de uma classe.

Para diferenciar entre classes e objetos, vamos examinar um exemplo. O que temos aqui é uma classe Carro que pode ser usada pra definir diversos objetos do tipo carro. Na tabela mostrada abaixo, Carro A e Carro B são objetos da classe Carro. A classe tem os *campos* número da placa, cor, fabricante e velocidade que são preenchidos com os valores correspondentes do carro A e B. O carro também tem alguns métodos: acelerar, virar e frear.

Classe Carro		**Objeto Carro A**	**Objeto Carro B**
Atributos de Objeto	Número da placa	ABC 111	XYZ 123
	Cor	Azul	Vermelha
	Fabricante	Mitsubishi	Toyota
	Velocidade	50 km/h	100 km/h
Métodos de Objeto	Método Acelerar		
	Método Girar		
	Método Frear		

Tabela 2: Exemplos da classe Carro e seus objetos

Quando construídos, cada objeto adquire um conjunto novo de estado. Entretanto, as implementações dos métodos são compartilhadas entre todos os objetos da mesma classe.

As classes fornecem o benefício do **Reutilização de Classes** (ou seja, utilizar a mesma classe em vários projetos). Os programadores de software podem reutilizar as classes várias vezes para criar os objetos.

3.2. Encapsulamento

Encapsulamento é um princípio que propõe ocultar determinados elementos de uma classe das demais classes. Ao colocar uma proteção ao redor dos atributos e criar métodos para prover o acesso a estes, desta forma estaremos prevenindo contra os efeitos colaterais indesejados que podem afetá-los ao ter essas propriedades modificadas de forma inesperada.

Podemos prevenir o acesso aos dados dos nossos objetos declarando que temos controle desse acesso. Aprenderemos mais sobre como Java implementa o encapsulamento quando discutirmos mais detalhadamente sobre as classes.

3.3. Atributos e Métodos de Classe

Além dos atributos de objeto, também é possível definir **atributos de classe,** que são atributos que pertencem à classe como um todo. Isso significa que possuem o mesmo valor para todos os objetos daquela classe. Também são chamados de **atributos estáticos.**

Para melhor descrever os atributos de classe, vamos voltar ao exemplo da classe **Carro**. Suponha que a classe **Carro** tenha um atributo de classe chamado **Contador**. Ao mudarmos o valor de **Contador** para 2, todos os objetos da classe **Carro** terão o valor 2 para seus atributos Contador.

Classe Carro		**Objeto Carro A**	**Objeto Carro B**
Atributos de Objeto	Número da placa	ABC 111	XYZ 123
	Cor	Azul	Vermelho
	Fabricante	Mitsubishi	Toyota
	Velocidade	50 km/h	100 km/h
Atributos de Classe	Contador = 2		
Métodos de Objeto	Método Acelerar		
	Método virar		
	Método Frear		

Tabela 3: Atributos e métodos da classe Carro

3.4. Instância de Classe

Para criar um objeto ou uma instância da classe, utilizamos o operador **new**. Por exemplo, para criar uma instância da classe String, escrevemos o seguinte código:

```
String  str2 = new String("Hello world!");
```

ou, o equivalente:

```
String str2 = "Hello world!";
```

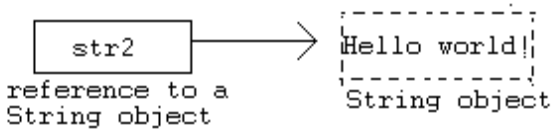


Figura 1: Instanciação de uma classe

O operador **new** aloca a memória para o objeto e retorna uma **referência** para essa alocação. Ao criar um objeto, invoca-se, na realidade, o **construtor** da classe. O **construtor** é um método onde todas as inicializações do objeto são declaradas e possui o mesmo nome da classe.

4. Métodos

4.1. O que são métodos e porque usar métodos?

Nos exemplos apresentados anteriormente, temos apenas um método, o método main(). Em Java, nós podemos definir vários métodos e podemos chamá-los a partir de outros métodos.

Um **método** é um trecho de código distinto que pode ser chamado por qualquer outro método para realizar alguma função específica.

Métodos possuem as seguintes características:

- Podem ou não retornar um valor
- Podem aceitar ou não argumentos
- Após o método encerrar sua execução, o fluxo de controle é retornado a quem o chamou

O que é necessário para se criar métodos? Porque não colocamos todas as instruções dentro de um grande método? O foco destas questões é chamado de **decomposição**. Conhecido o problema, nós o separamos em partes menores, que torna menos crítico o trabalho de escrever grandes classes.

4.2. Chamando Métodos de Objeto e Enviando Argumentos

Para ilustrar como chamar os métodos, utilizaremos como exemplo a classe **String**. Pode-se usar a documentação da API Java para conhecer todos os atributos e métodos disponíveis na classe **String**. Posteriormente, iremos criar nossos próprios métodos.

Para chamar um **método** a partir de um objeto, escrevemos o seguinte:

```
nomeDoObjeto.nomeDoMétodo([argumentos]);
```

Vamos pegar dois métodos encontrados na classe String como exemplo:

Declaração do método	Definição
public char charAt(int index)	Retorna o caractere especificado no índice. Um índice vai de 0 até length() - 1. O primeiro caractere da seqüência está no índice 0, o seguinte, no índice 1, e assim sucessivamente por todo o array.
public boolean equalsIgnoreCase (String anotherString)	Compara o conteúdo de duas Strings, ignorando maiúsculas e minúsculas. Duas strings são consideradas iguais quando elas têm o mesmo tamanho e os caracteres das duas strings são iguais, sem considerar caixa alta ou baixa.

Tabela 4: Exemplos de Métodos da classe String

Usando os métodos:

```
String str1 = "Hello";
char x = str1.charAt(0); // retornará o caracter H
                         // e o armazenará no atributo x
```

```
String      str2 = "hello";

// aqui será retornado o valor booleano true
boolean  result = str1.equalsIgnoreCase(str2);
```

4.3. Envio de Argumentos para Métodos

Em exemplos anteriores, enviamos atributos para os métodos. Entretanto, não fizemos nenhuma distinção entre os diferentes tipos de atributos que podem ser enviados como argumento para os métodos. Há duas formas para se enviar argumentos para um método, o primeiro é envio por valor e o segundo é envio por referência.

4.3.1. Envio por valor

Quando ocorre um **envio por valor**, a chamada do método faz uma cópia do valor do atributo e o reenvia como argumento. O método chamado não modifica o valor original do argumento mesmo que estes valores sejam modificados durante operações de cálculo implementadas pelo método. Por exemplo:

```
public class TestPassByValue {
     public static void main( String[] args ){
       int i = 10;
       //exibe o valor de i
       System.out.println( i );

       //chama o método test
       //envia i para o método test
       test( i );

       //exibe o valor de i não modificado
       System.out.println( i );
      }

      public static void test( int j ){
       //muda o valor do argumento j
      j = 33;
      }
}
```

No exemplo dado, o método **test** foi chamado e o valor de **i** foi enviado como argumento. O valor de **i** é copiado para o atributo do método **j**. Já que **j** é o atributo modificado no método **test**, não afetará o valor do atributo **i**, o que significa uma cópia diferente do atributo.

Como padrão, todo tipo primitivo, quando enviado para um método, utiliza a forma de **envio por valor**.

4.3.2. Envio por referência

Quando ocorre um **envio por referência**, a referência de um objeto é enviada para o método chamado. Isto significa que o método faz uma cópia da referência do objeto enviado. Entretanto, diferentemente do que ocorre no envio por valor, o método pode modificar o objeto para o qual a referência está apontando. Mesmo que diferentes referências sejam usadas nos métodos, a localização do dado para o qual ela aponta é a mesma.

Por exemplo:

```
class TestPassByReference {
    public static void main(String[] args) {
        // criar um array de inteiros
        int []ages      = {10, 11, 12};
        // exibir os valores do array
        for (int i=0; i < ages.length; i++) {
            System.out.println( ages[i] );
        }
        // chamar o método test e enviar a
        // referência para o array
        test( ages );

        // exibir os valores do array
        for (int i=0; i < ages.length; i++) {
            System.out.println(ages[i]);
        }
    }
    public static void test( int[] arr ){
        // mudar os valores do array
        for (int i=0; i < arr.length; i++) {
            arr[i] = i + 50;
        }
    }
}
```

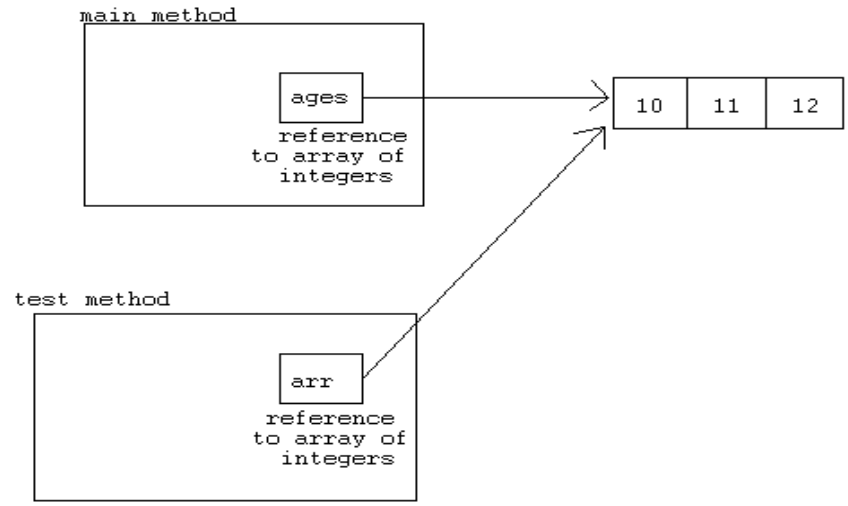


Figura 2: Exemplo de Envio por referência

Dicas de programação:

1. Um erro comum sobre **envio por referência** acontece quando criamos um método para fazer trocas (**swap**) usando referência. Note que Java manipula objetos 'por referência', entretanto envia-se a referência para um método 'por valor'. Como conseqüência, não se escreve um método padrão para fazer troca de valores (**swap**) entre objetos.

4.4. Chamando métodos estáticos

Métodos estáticos são métodos que podem ser invocados sem que um objeto tenha sido instanciado pela classe (sem utilizar a palavra-chave *new*). Métodos estáticos pertencem a classe como um todo e não ao objeto especifico da classe. Métodos estáticos são diferenciados dos métodos de objeto pela declaração da palavra-chave *static* na definição do método.

Para chamar um método estático, digite:

```
NomeClasse.nomeMétodoEstático(argumentos);
```

Alguns métodos estáticos, que já foram usados em nossos exemplos são:

```
// Converter a String 10 em um atributo do tipo inteiro
int i = Integer.parseInt("10");

// Retornar uma String representando um inteiro sem o sinal da
// base 16
String hexEquivalent = Integer.toHexString(10);
```

4.5. Escopo de um atributo

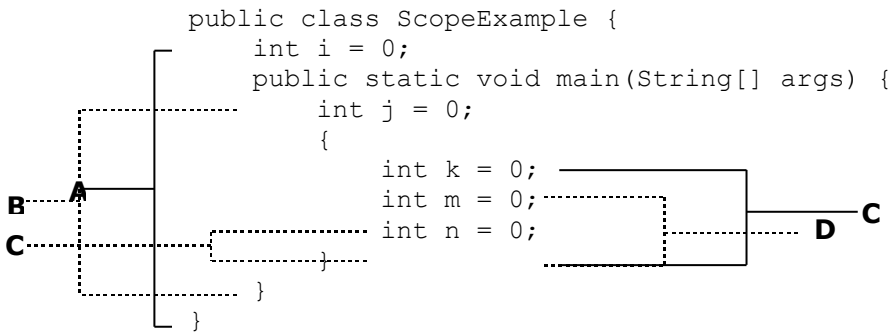
Além do atributo ter um nome e um tipo, ele também possui um escopo. O **escopo** determina onde o atributo é acessível dentro da classe. O escopo também determina o tempo de vida do atributo ou quanto tempo o atributo irá existir na memória. O escopo é determinado pelo local onde o atributo é declarado na classe.

Para simplificar, vamos pensar no escopo como sendo algo existente entre as chaves {...}. A chave à direita é chamada de chave de saída do bloco (**outer**) e a chave à esquerda é chamada chave de entrada do bloco (**inner**).

Ao declarar atributos fora de um bloco, eles serão visíveis (usáveis) inclusive pelas linhas da classe dentro do bloco. Entretanto, ao declarar os atributo dentro do bloco, não será possível utilizá-los fora do bloco.

O escopo de um atributo é dito local quando é declarado dentro do bloco. Seu escopo inicia com a sua declaração e vai até a chave de saída do bloco.

Por exemplo, dado o seguinte fragmento de código:



O código acima representa cinco escopos indicado pelas letras. Dados os atributos **i**, **j**, **k**, **m** e **n**, e os cinco escopos **A**, **B**, **C**, **D** e **E**, temos os seguintes escopos para cada atributo:

```
O escopo do atributo i é A.
```

```
O escopo do atributo j é B.
O escopo do atributo k é C.
O escopo do atributo m é D.
O escopo do atributo n é E.
```

Dado dois métodos: **main** e **test** teremos o seguinte exemplo:

```
class TestPassByReference {
    public static void main(String[] args) {
        // criar um array de inteiros
        int []ages = {10, 11, 12};

        //exibir os valores do array
        for (int i=0; i < ages.length; i++){
            System.out.println( ages[i] );
        }

        // chamar o método test e enviar a referência para o array
        test(ages);

        //exibe novamente os valores do array
        for (int i = 0; i < ages.length; i++) {
            System.out.println( ages[i] );
        }
    }
    public static void test(int[] arr) {
        // modificar os valores do array
        for (int i = 0; i < arr.length; i++) {
            arr[i] = i + 50;
        }
    }
}
```


Para o método **main**, os escopos dos atributos são:

```
ages[] - escopo A
i em B - escopo B
i em C – escopo C
```

E, no método **test**, os escopos dos atributos são:

```
arr[]  - escopo D
i em E - escopo E
```

Quando atributos são declarados, o identificador deve ser único no escopo. Isto significa que se você tiver a seguinte declaração:

```
{
      int test = 10;
      int test = 20;
}
```

o compilador irá gerar um erro pois deve-se ter um nome único para o atributos dentro do bloco. Entretanto, é possível ter atributos definidos com o mesmo nome, se não estiverem declarados no mesmo bloco. Por exemplo:

```
public class TestBlock {
    int test = 10;
```

```
        public void test() {
            System.out.print(test);
            int test = 20;
            System.out.print(test);
        }
        public static void main(String[] args) {
            TestBlock testBlock = new TestBlock();
            testBlock.test();
        }
    }
```

Quando a primeira instrução **System.out.print** for invocada, exibirá o valor **10** contido na primeira declaração do atributo **test**. Na segunda instrução **System.out.print**, o valor **20** é exibido, pois é o valor do atributos **test** neste escopo.

Dicas de programação:

1. Evite ter atributos declarados com o mesmo nome dentro de um método para não causar confusão.

5. *Casting*, Conversão e Comparação de Objetos

Nesta seção, vamos aprender como realizar um ***casting***. ***Casting***, ou ***typecasting***, é o processo de conversão de um certo tipo de dado para outro. Também aprenderemos como converter tipos de dados primitivos para objetos e vice-versa. E, finalmente, aprenderemos como comparar objetos.

5.1. Casting de Tipos Primitivos

Casting entre tipos primitivos permite converter o valor de um dado de um determinado tipo para outro tipo de dado primitivo. O *casting* entre primitivos é comum para os tipos numéricos.

Há um tipo de dado primitivo que não aceita o *casting*, o tipo de dado **boolean**.

Como demonstração de *casting* de tipos, considere que seja necessário armazenar um valor do tipo **int** em um atributo do tipo **double**. Por exemplo:

```
int numInt = 10;
double numDouble = numInt; // cast implícito
```

uma vez que o atributo de destino é **double**, pode-se armazenar um valor cujo tamanho seja menor ou igual aquele que está sendo atribuído. O tipo é convertido implicitamente.

Quando convertemos um atributo cujo tipo possui um tamanho maior para um de tamanho menor, necessariamente devemos fazer um ***casting* explícito**. Esse possui a seguinte forma:

```
(tipoDado)valor
```

onde:

tipoDado é o nome do tipo de dado para o qual se quer converter o valor
valor é um valor que se quer converter.

Por exemplo:

```
double valDouble = 10.12;
int valInt = (int)valDouble; //converte valDouble para o tipo int
double x = 10.2;
int y = 2;
int result = (int)(x/y); //converte o resultado da operação para int
```

Outro exemplo é quando desejamos fazer um *casting* de um valor do tipo **int** para **char**. Um caractere pode ser usado como **int** porque para cada caractere existe um correspondente numérico que representa sua posição no conjunto de caracteres. O *casting* **(char)65** irá produzir a saída 'A'. O código numérico associado à letra maiúscula A é 65, segundo o conjunto de caracteres ASCII. Por exemplo:

```
char valChar = 'A';
System.out.print((int)valChar); //casting explícito produzirá 65
```

5.2. Casting de Objetos

Para objetos, a operação de *casting* também pode ser utilizada para fazer a conversão para outras classes, com a seguinte restrição: a classe de origem e a classe de destino devem ser da mesma família, relacionadas por herança; uma classe deve ser subclasse da outra. Veremos mais em lições posteriores sobre herança.

Analogamente à conversão de valores primitivos para um tipo maior, alguns objetos não necessitam ser convertidos explicitamente. Em conseqüência de uma subclasse conter todas as informações da sua superclasse, pode-se usar um objeto da subclasse em qualquer lugar onde a superclasse é esperada.

Por exemplo, um método que recebe dois argumentos, um deles do tipo **Object** e outro do tipo **Window**. Pode-se enviar um objeto de qualquer classe como argumento **Object** porque todas as classes Java são subclasses de **Object**. Para o argumento **Window**, é possível enviar apenas suas subclasses, tais como **Dialog**, **FileDialog**, **Frame** (ou quaisquer de subclasses de suas subclasses, indefinidamente). Isso vale para qualquer parte da classe, não apenas dentro da chamadas do método. Para um objeto definido como uma classe Window, é possível atribuir objetos dessa classe ou qualquer uma de suas subclasses para esse objeto sem o *casting*.

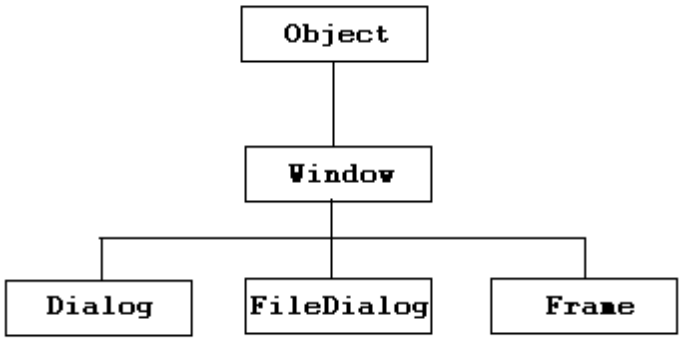


Figura 3: Exemplo de Hierarquia de Classes

O contrário também é verdadeiro. Uma superclasse pode usada quando uma subclasse é esperada. Entretanto, nesse caso, o *casting* é necessário porque as subclasses contém mais métodos que suas superclasses, e isso acarreta em perda de precisão. Os objetos das superclasses podem não dispor de todo o comportamento necessário para agir como um objeto da subclasse. Por exemplo, se uma operação faz a chamada a um método de um objeto da classe **Integer**, usando um objeto da classe **Number**, ele não terá muitos dos métodos que foram especificados na classe **Integer**. Erros ocorrerão se você tentar chamar métodos que não existem no objeto de destino.

Para usar objetos da superclasse onde uma subclasse é esperada, é necessário fazer o ***casting* explícito**. Nenhuma informação será perdida no *casting*, entretanto, ganhará todos os atributos e métodos que a subclasse define. Para fazer o *casting* de um objeto para outro, utiliza-se a mesma operação utilizada com os tipos primitivos:

Para fazer o *casting*:

```
(nomeClasse)objeto
```

onde:

nomeClasse é o nome da classe destino
objeto é a referência para o objeto origem que se quer converter

Uma vez que *casting* cria uma referência para o antigo objeto de **nomeClasse**; ele continua a existir depois do *casting*.

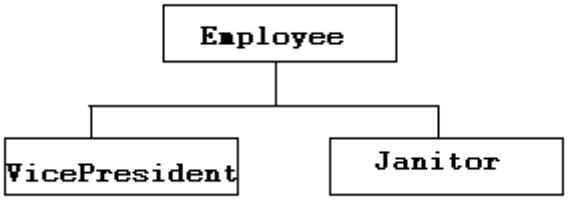


Figura 4: Hierarquia de classes para a superclasse Employee

O exemplo a seguir realiza um *casting* de um objeto da classe **VicePresident** para um objeto da classe **Employee**. **VicePresident** é uma subclasse de **Employee** com mais informações, supondo que tenha sido definido que a classe **VicePresident** tem mais privilégios que um **Employee**.

```
Employee emp = new Employee();
VicePresident veep = new VicePresident();
emp = veep; // casting não é necessário de baixo para cima
veep = (VicePresident)emp; // necessita do casting explícito
```

5.3. Convertendo Tipos Primitivos para Objetos e Vice-Versa

Não se pode fazer sob qualquer circunstância é um *casting* de um objeto para um tipo de dado primitivo, ou vice-versa. Tipos primitivos e objetos são muito diferentes em Java e não se pode fazer o *casting* automaticamente entre os dois ou intercambiar o seu uso.

Como alternativa, o pacote **java.lang** oferece classes que fazem a correspondência para cada tipo primitivo: Float, Boolean, Byte, dentre outros. Muitas dessas classes têm o mesmo nome dos tipos de dados primitivos, exceto pelo fato que o nome dessas classe começam com letra maiúscula (Short ao invés de short, Double ao invés de double, etc). Há duas classes que possuem nomes que diferem do correspondente tipo de dado primitivo: **Character** é usado para o tipo **char** e **Integer** usado para o tipo **int**. Estas classes são chamadas de **Wrapper Class**.

Versões anteriores a 5.0 de Java tratam os tipos de dados e suas **Wrapper Class** de forma muito diferente e uma classe não será compilada com sucesso se for utilizado uma quando deveria usar a outra. Por exemplo:

```
Integer ten = 10;
Integer two = 2;
System.out.println(ten + two);
```

Usando as classes que correspondem a cada um dos tipos primitivos, é possível criar objetos que armazenam este mesmo valor. Por exemplo:

```
// A declaração seguinte cria um objeto de uma classe Integer
// com o valor do tipo int 7801 (primitivo -> objeto)
Integer dataCount = new Integer(7801);

// A declaração seguinte converte um objeto Integer para um
// tipo de dado primitivo int. O resultado é o valor 7801
int newCount = dataCount.intValue();

// Uma conversão comum de se fazer é de String para um tipo
// numérico (objeto -> primitivo)
String pennsylvania = "65000";
int penn = Integer.parseInt(pennsylvania);
```

Dicas de programação:

1. A classe Void representa vazio em Java. Deste modo, não existem motivos para ela ser usada na conversão de valores primitivos e objetos. Ela é um tratador para a palavra-chave *void*, que é utilizada na assinatura de métodos indicando que estes não retornam valor.

5.4. Comparando Objetos

Em lições prévias, aprendemos sobre operadores para comparar valores — igualdade, negação, menor que, etc. Muitos desses operadores trabalham apenas com dados de tipo primitivo, não com objetos. Ao tentar utilizar outros tipos de dados, o compilador produzirá erros.

Uma exceção para esta regra são os operadores de igualdade: == (igual) e != (diferente). Quando aplicados a objetos, estes operadores não fazem o que se poderia supor. Ao invés de verificar se um objeto tem o mesmo valor de outro objeto, eles determinam se os dois objetos comparados pelo operador têm a mesma referência. Por exemplo:

```
String valor1 = new String;
Integer dataCount = new Integer(7801);
```

Para comparar objetos de uma classe e ter resultados apropriados, deve-se implementar e chamar métodos especiais na sua classe. Um bom exemplo disso é a classe String.

É possível ter dois objetos String diferentes que contenham o mesmo valor. Caso se empregue o operador **==** para comparar objetos, estes serão considerados diferentes. Mesmo que seus conteúdos sejam iguais, eles não são o mesmo objeto.

Para ver se dois objetos String têm o mesmo conteúdo, um método chamado **equals()** é utilizado. O método compara cada caractere presente no conteúdo das Strings e retorna **true** se ambas strings tiverem o mesmo valor.

O código seguinte ilustra essa comparação,

```
class EqualsTest {
   public static void main(String[] args) {
     String str1, str2;
     str1 = "Free the bound periodicals.";
     str2 = str1;

     System.out.println("String1: " + str1);
     System.out.println("String2: " + str2);
     System.out.println("Same object? " + (str1 == str2));

     str2 = new String(str1);

     System.out.println("String1: " + str1);
     System.out.println("String2: " + str2);
     System.out.println("Same object? " + (str1 == str2));
     System.out.println("Same value? " + str1.equals(str2));
   }
 }
```

A saída dessa classe é a seguinte:

```
String1: Free the bound periodicals.
String2: Free the bound periodicals.
Same object? true
String1: Free the bound periodicals.
String2: Free the bound periodicals.
Same object? false
Same value? true
```

Vamos entender o processo envolvido.

```
String str1, str2;
str1 = "Free the bound periodicals.";
str2 = str1;
```

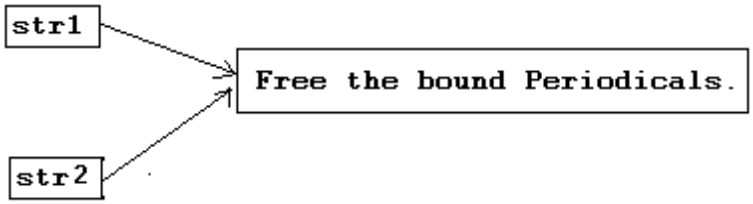


Figura 5: Duas referências apontando para o mesmo objeto

A primeira parte dessa classe declara dois atributos (str1 e str2) e atribui a literal "Free the bound periodicals." a str1, e depois atribui esse valor a str2. Como visto anteriormente, str1 e str2 agora apontam para o mesmo objeto, e o teste de igualdade prova isso.

Em seguida, temos a seguinte instrução:

```
str2 = new String(str1);
```

Na segunda parte da classe, cria-se um novo objeto String com o mesmo valor de str1 e faz-se a atribuição de str2 para esse novo objeto String. Agora temos dois diferentes tipos de objetos na str1 e str2, ambos com o mesmo conteúdo. Testando para ver se eles são o mesmo objeto usando o operador de igualdade obtemos a resposta esperada: **false** — eles não são o mesmo objeto na memória. Utilizando o método **equals()** recebemos a resposta esperada: **true** — eles tem o mesmo conteúdo.

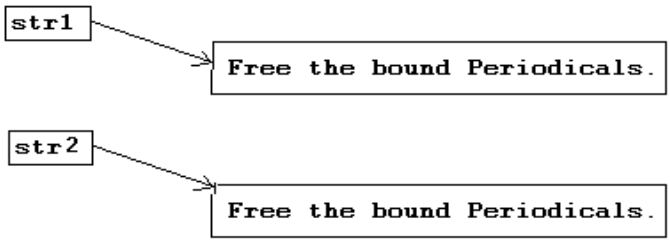


Figura 6: As referências apontam agora para objetos diferentes

Dicas de programação:

1. Porque não se pode ter a mesma literal quando mudamos str2, a não ser quando utilizamos o **new**? Literais String são otimizadas em Java; se uma String é criada utilizando uma literal e cria-se outra String com os mesmos caracteres,

> Java tem inteligência suficiente para retornar apenas a posição em memória do primeiro objeto String criado. Ou seja, ambas strings se referem ao mesmo objeto como se fosse um apelido. Para obrigar ao Java criar um novo objeto String, deve-se utilizar o operador new.

5.5. Determinando a Classe de um Objeto

Existem duas maneiras de se descobrir a qual classe determinado objeto pertence:

1. Para obter o nome da classe:

Utiliza-se o método **getClass()** que retorna a classe do objeto (onde Class é a classe em si). Esta, por sua vez, possui o método chamado **getName()** que retorna o nome da classe.

Por exemplo:

```
String name = key.getClass().getName();
```

2. Para testar se um objeto qualquer foi instanciado de uma determinada classe:

Utiliza-se a palavra-chave *instanceof*. Esta palavra-chave possui dois operadores: a referência para o objeto à esquerda e o nome da classe à direita. A expressão retorna um lógico dependendo se o objeto é uma instância da classe declarada ou qualquer uma de suas subclasses.

Por exemplo:

```
String ex1 = "Texas";
System.out.println(ex1 instanceof String); // retorna true
String ex2;
System.out.println(ex2 instanceof String); // retorna false
```

6. Exercícios

6.1. Definindo termos

Em suas palavras, defina os seguintes termos:

1. Classe
2. Objeto
3. Instanciação
4. Atributo de objeto
5. Método de objeto
6. Atributo de classe ou atributos estáticas
7. Construtor
8. Método de classe ou métodos estáticos

6.2. Java Scavenger Hunt

Pipoy é um novato na linguagem de programação Java. Ele apenas ouviu que existem Java APIs (Application Programming Interface) prontas para serem usadas em suas classes e ele está ansioso para fazer uns testes com elas. O problema é que **Pipoy** não tem uma cópia da documentação Java e ele também não tem acesso à Internet, deste modo, não há como ele ver as APIs java.

Sua tarefa é ajudar **Pipoy** a procurar as APIs. Você deve informar as classes às quais os métodos pertencem e como o método deve ser declarado com um exemplo de uso deste.

Por exemplo, se **Pipoy** quer saber qual o método que converte uma String para int, sua resposta deve ser:

> **Classe:** Integer
> **Declaração do Método:** public static int parseInt(String value)
> **Exemplo de Uso:**
> ```
> String      strValue = "100";
> int     value = Integer.parseInt( strValue );
> ```

Tenha certeza de que o fragmento de código que você escreveu em seu exemplo de uso compila e que produza o resultado correto. Enfim, não deixe **Pipoy** confuso. (**Dica: Todos os métodos estão no package java.lang**). Caso haja mais de um método para atender à tarefa, utilize apenas um.

Agora vamos iniciar a busca! Aqui estão alguns métodos que **Pipoy** necessita:

1. Procure pelo método que verifica se uma String termina com um determinado sufixo. Por exemplo, se a String dada é "Hello", o método deve retornar true se o sufixo informado é "lo", e false se o sufixo for "alp".
2. Procure pelo método que determina a representação do caractere para um dígito e base específicos. Por exemplo, se o dígito informado é 15 e a base é 16, o método retornará o caractere 'F', uma vez que 'F' é a representação hexadecimal para o número 15 em base 10.
3. Procure por um método que retorna a parte inteira de um valor double. Por exemplo, se a entrada for 3.13, o método deve retornar o valor 3.
4. Procure por um método que determina se um certo caractere é um dígito. Por exemplo, se a entrada for '3', retornará o valor true.
5. Procure por um método que interrompe a execução da Java Virtual Machine corrente.

Parceiros que tornaram JEDI™ possível

Instituto CTS
Patrocinador do DFJUG.

Sun Microsystems
Fornecimento de servidor de dados para o armazenamento dos vídeo-aulas.

Java Research and Development Center da Universidade das Filipinas
Criador da Iniciativa JEDI™.

DFJUG
Detentor dos direitos do JEDI™ nos países de língua portuguesa.

Banco do Brasil
Disponibilização de seus *telecentros* para abrigar e difundir a Iniciativa JEDI™.

Politec
Suporte e apoio financeiro e logístico a todo o processo.

Borland
Apoio internacional para que possamos alcançar os outros países de língua portuguesa.

Instituto Gaudium/CNBB
Fornecimento da sua infra-estrutura de hardware de seus servidores para que os milhares de alunos possam acessar o material do curso simultaneamente.