Módulo 2

Introdução à Programação II

Lição 1

Revisão dos Conceitos Básicos em Java

Versão 1.0 - Mar/2007

Autor
Rebecca Ong

Equipe
Joyce Avestro
Florence Balagtas
Rommel Feria
Rebecca Ong
John Paul Petines
Sun Microsystems
Sun Philippines

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

Antes de entrar em outras características do Java, inicialmente iremos revisar alguns assuntos que vimos no primeiro módulo do curso. Esta lição fornece uma discussão breve sobre os diferentes conceitos de Orientação a Objetos em Java.

Ao final desta lição, o estudante será capaz de:

- Explicar e usar os conceitos básicos de orientação a objetos em seus códigos
  - classes, objetos, atributos, métodos e construtores
- Descrever conceitos avançados de orientação a objetos e aplicá-los na codificação
  - pacote, encapsulamento, abstração, herança, polimorfismo e interface
- Descrever e utilizar as palavras-chaves: *this*, *super*, *final* e *static*
- Diferenciar entre polimorfismo por overloading e override

2. Conceitos de orientação a objetos

2.1. Modelagem Orientada a Objetos

Modelagem Orientada a Objetos é uma técnica que possui foco na modelagem de objetos e classes baseados no cenário do mundo real. Ela dá enfase ao estado, comportamento e interação dos objetos. Possibilita o benefício do desenvolvimento rápido, aumenta a qualidade, fácil manutenção, a facilidade de alterações e a reutilização de software.

2.2. Classe

Permite definir novos tipos de dados. Serve como um referencial, a qual é um modelo para os objetos que é possível criar utilizando este novo tipo de dado.

O modelo de um estudante seria um exemplo de uma classe. Podemos definir que cada aluno terá uma série de qualidades tais como nome, número do estudante e nível escolar.

2.3. Objeto

É uma entidade que possui um estado, um comportamento e uma identidade com um papel bem definido no escopo do problema. É uma instância real de uma classe. Sendo assim, é chamado de **instância da classe**. Criado toda vez que for utilizado a palavra-chave *new*. Em um projeto de registro de estudantes, um exemplo de objeto pode ser uma entidade estudante, como Ana. Ana é um objeto da classe Estudante. Desta forma, as qualidades e habilidades definidas no modelo da classe Estudante são todos aplicáveis a Ana, já que **Ana é uma instância de Estudante**.

Para simplificar, pensamos em uma classe como um termo mais geral se comparado a um objeto.

2.4. Atributo

Refere-se ao elemento dos dados de um objeto. Ele armazena informações sobre o objeto. É também conhecido como dado do objeto, atributo do objeto, propriedade ou campo. No projeto de registro do aluno, alguns atributos da entidade aluno incluem o nome, número do estudante e nível de escolaridade.

2.5. Método

Descreve o comportamento de um objeto. Em linguagens relacionais seria comparado a uma função ou procedimento. Métodos disponíveis para a entidade estudante são genéricos e atendem a escola.

2.6. Construtor

É um tipo especial de método que é utilizado para a construção ou criação de um novo objeto. Lembre-se que construtores não são elementos (atributos, métodos e classes internas de um objeto).

2.7. Pacote

Refere ao agrupamento de classes e sub-classes. Sua estrutura é análoga a de um diretório.

2.8. Encapsulamento

Princípio de ocultar a modelagem ou as informações de implementação que não são referentes ao objeto atual.

2.9. Abstração

Princípio de ignorar os aspectos subjetivos que não são relevantes para o real propósito em prol de se concentrar mais diretamente naqueles que são.

2.10. Herança

Princípio que surge com a relação entre classes. Uma classe é a superclasse ou a classe pai de outra. É relativo às propriedades e aos comportamentos recebidos pelo antecessor. É também conhecida como uma relação "é-um" (is-a). Considere a seguinte hierarquia:

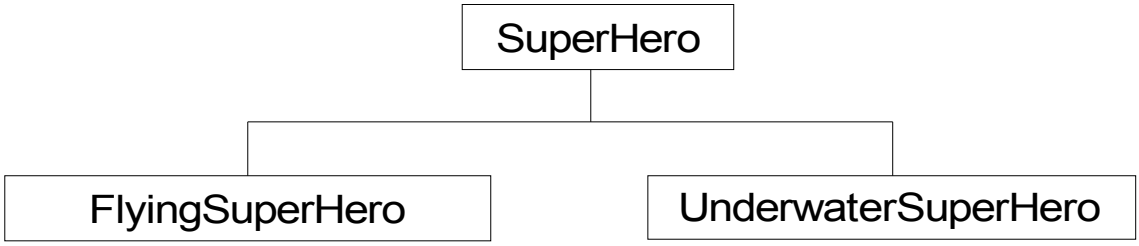

Figura 1: Exemplo de herança

SuperHero é a superclasse das classes *FlyingSuperHero* e *UnderwaterSuperHero*. Note que *FlyingSuperHero* "é-um" *SuperHero e UnderwaterSuperHero* "é-um" *SuperHero*.

2.11. Polimorfismo

Habilidade de um método poder assumir diferentes formas. Literalmente, "poli" significa muitas enquanto "morph" significa forma. Referenciando o exemplo anterior para herança, supomos um método displayPower na classe *SuperHero* que retorna o poder que o super-herói possui, na classe *FlyingSuperHero* este mesmo método poderia retornar "voar" enquanto que na classe *UnderwaterSuperHero* este mesmo método retornaria "respirar embaixo d'água".

2.12. Interface

É um contrato na forma de uma coleção de declarações de métodos e constantes. Quando uma classe implementa uma interface, ela se compromete a implementar todos os métodos declarados nesta.

3. Estrutura de codificação Java

Esta seção resume a sintaxe básica usada para a criação de aplicações em Java.

3.1. Declarando classes em Java

```
<declaraçãoClasse> ::= <modificador> class <nomeClasse> {
  <declaraçãoAtributo>*
  <declaraçãoConstrutor>*
  <declaraçãoMétodo>*
}
```

onde:

<*modificador*> é um modificador de acesso, o qual deve ser combinado com outros tipos de modificadores.

Guia de código:

`*`	Poderá existir nenhuma ou diversas ocorrências da linha onde este símbolo foi aplicado.
`<descrição>`	Indica a descrição de um valor para esta parte.

Lembre-se que para uma classe de alto nível, os únicos modificadores de acesso são *public* e *default* (neste caso, nenhum modificador de acesso precede a palavra-chave *class*).

O seguinte exemplo declara a classe *SuperHero*:

```
class SuperHero {
   String superPowers[];
   void setSuperPowers(String superPowers[])  {
      this.superPowers = superPowers;
   }
   void printSuperPowers() {
      for (int i = 0; i < superPowers.length; i++) {
         System.out.println(superPowers[i]);
      }
   }
}
```

3.2. Declarando Atributos

```
<declaraçãoAtributo> ::=
   <modificador> <tipo> <nome> [= <valorPadrão>];
<tipo> ::=
   byte | short | int | long | char | float | double | boolean
   | <classeQualquer>
```

Guia de código:

`[]`	Indica que esta parte é opcional.

Aqui está um exemplo.

```
public class AttributeDemo {
```

```
   private String studNum;
   public boolean graduating = false;
   protected float unitsTaken = 0.0f;
   String college;
}
```

3.3. Declarando métodos

```
<declaraçãoMétodo> ::=
   <modificador> <tipoRetorno> <nome>(<argumento>*) {
      <instrução>*
   }
<argumento> ::=
   <tipoArgumento> <nomeArgumento>[,]
```

Por exemplo:

```
class MethodDemo {
   int data;
   int getData() {
      return data;
   }
   void setData(int data) {
      this.data = data;
   }
   void setMaxData(int data1, int data2) {
      data = (data1>data2)? data1 : data2;
   }
}
```

3.4. Declarando um construtor

```
<declaraçãoConstrutor> ::=
   <modificador> <nome da classe> (<argumento>*) {
      <instrução;>*
   }
```

Se um construtor não é explicitamente fornecido, um construtor padrão é automaticamente criado. O construtor padrão não possui argumentos e o seu corpo não possui instruções.

Dicas de programação:

1. O nome do construtor deve ser o mesmo nome da classe.
2. O único <modificador> válido para construtores são public, protected, e private.
3. Construtores não possuem valor de retorno.

Considere a seguinte classe:

```
class ConstructorDemo {
   private int data;
   public ConstructorDemo() {
      data = 100;
   }
   ConstructorDemo(int data) {
      this.data = data;
```

```
    }
}
```

3.5. Construir um objeto

Para construir um objeto a partir de uma classe, utilizamos a palavra-chave *new* seguida por uma chamada ao construtor.

```
class ConstructObj {
   int data;
   ConstructObj() {
      /* Inicialização dos dados */
   }
   public static void main(String args[]) {
      ConstructObj obj = new ConstructObj();
   }
}
```

3.6. Acessando elementos dos objetos

Para acessar os elementos de um objeto, utilizamos a notação do "ponto". É usada da seguinte forma:

```
<objeto>.<elemento>
```

O exemplo seguinte, feito com base no anterior com instruções adicionais para acessar os elementos.

```
class ConstructObj {
   int data;
   ConstructObj() {
      /* Dados de inicialização */
   }
   void setData(int data) {
      this.data = data;
   }
   public static void main(String args[]) {
      ConstructObj obj = new ConstructObj();   //instanciamento
      obj.setData(10);        //acesso a setData()
      System.out.println(obj.data); //acesso a data
   }
}
```

A execução desta classe mostrará o seguinte resultado:

```
10
```

3.7. Pacotes

Para indicar que uma determinada classe pertence a um pacote em particular, utilizamos a seguinte sintaxe:

```
<declaraçãoPacote> ::=
   package <nomePacote>;
```

Para importar outros pacotes, use a seguinte sintaxe:

```
<declaraçãoImportação> ::=
   import <nomePacote.elementoAcessado>;
```

Com isso, seu código fonte deve ter o seguinte formato:

```
[<declaraçãoPacote>]
<declaraçãoImportação>*
<declaraçãoClasse>+
```

Guia de código:

\+ que pode ter 1 ou mais ocorrências da linha onde isso foi aplicado.

Vejamos o seguinte exemplo:

```
// Classe pertence ao pacote registration.reports
package registration.reports;

// Importa todas as classes do pacote registration.processing
import registration.processing.*;
// Importa a classe List do pacote java.util
import java.util.List;

// Cria a classe myClass
class MyClass {
   /* detalhes da classe MyClass */
}
```

3.8. Os modificadores de acesso

A tabela seguinte resume os modificadores de acesso em Java.

	private	**default**	***protected***	***public***
Mesma classe	sim	sim	sim	sim
Mesmo pacote	não	sim	sim	sim
Pacotes diferentes (subclasse)	não	não	sim	sim
Pacotes diferentes (não sendo subclasse)	não	não	não	sim

Tabela 1: Modificadores de acesso

3.9. Encapsulamento

Protege os elementos da implementação de uma classe por ser realizado utilizando o modificador de acesso particular na declaração dos atributos.

O exemplo seguinte protege o atributo *secret*. Note que este atributo é indiretamente acessado por outras classes utilizando os métodos **get** e **set**.

```
class Encapsulation {
   private int secret;  //Campo oculto
   public boolean setSecret(int secret) {
      if (secret < 1 || secret > 100) {
         return false;
      }
      this.secret = secret;
      return true;
   }
   public getSecret() {
```

```
        return secret;
    }
}
```

Caso não se deseje que outras classes modifiquem o atributo *secret*, é possível configurar o modificador de acesso do método *setSecret()* como particular (*private*).

3.10. Herança

Para se criar uma classe filha ou uma subclasse com base em uma classe existente, utilizamos a palavra-chave *extends* na declaração da classe. Uma classe pode estender qualquer classe desde que ela não possua o modificador *final*.

Por exemplo. A classe *Point* é a super-classe da classe *ColoredPoint*:

```
import java.awt.*;
class Point {
    int x;
    int y;
}

class ColoredPoint extends Point {
    Color color;
}
```

3.11. Realizando override em métodos

Um método de uma subclasse pode modificar um método de uma super-classe quando a subclasse define um método cuja assinatura é idêntica ao método da super-classe. A assinatura de um método e a informação encontrada no cabeçalho de definição deste. A assinatura inclui o tipo de retorno, o nome e a lista de argumentos do método. Os modificadores de acesso e outras palavras-chaves, tais como *final* e *static*, não estão incluídos.

```
class Superclass {
    void display(int n) {
        System.out.println("super: " + n);
    }
}
class Subclass extends Superclass {
    void display(int k) {      // método overriding
        System.out.println("sub: " + k);
    }
}
class OverrideDemo {
    public static void main(String args[]) {
        Subclass SubObj = new Subclass();
        Superclass SuperObj = SubObj;
        SubObj.display(3);
        ((Superclass)SubObj).display(4);
    }
}
```

A execução desta classe mostrará o seguinte resultado:

```
sub: 3
sub: 4
```

O método chamado é determinado pelo tipo de dado do objeto que invoca o método.

Os modificadores de acesso dos métodos não precisam ser os mesmos. Contudo, os métodos polimórficos devem ter modificadores de acesso igual ou menos restritivos que os métodos originais.

Considere o seguinte exemplo. Vejamos qual dos seguintes métodos com polimorfismo por *override* pode causar um erro no momento de compilação.

```
class Superclass {
   void overriddenMethod() {
   }
}

class Subclass1 extends Superclass {
   public void overriddenMethod() {
   }
}

class Subclass2 extends Superclass {
   void overriddenMethod() {
   }
}

class Subclass3 extends Superclass {
   protected void overriddenMethod() {
   }
}

class Subclass4 extends Superclass {
   private void overriddenMethod() {
   }
}
```

O método *overriddenMethod()* da *Superclass* possui o modificador de acesso *default*. O único modificador mais restrito que esse é o *private*. Sendo assim, *Subclass4* provoca um erro de compilação porque ele tenta modificar o método *overriddenMethod()* na *Superclass* com um modificar private que é mais restritivo.

3.12. Classes e métodos abstratos

A forma genérica para um método abstrato é a seguinte:

```
abstract <modificador> <tipoRetorno> <nome>(<argumento>*);
```

Uma classe contendo um método abstrato deve ser declarada como uma classe abstrata.

```
abstract class <Nome> {
   /* construtores, campos e métodos */
}
```

A palavra-chave *abstract* não pode ser aplicada para construtores ou métodos estáticos. É importante lembrar também que classes abstratas não podem servir para a construção de objetos (desde que seus métodos sejam implementados).

Classes que estendem uma classe abstrata são obrigadas a implementar **todos** os métodos abstratos. Caso contrário a subclasse deverá ser declarada também como *abstract*.

Dicas de programação:

1. Note que declarar um método abstract é muito similar à declaração de uma classe normal exceto que um método abstrato não possui corpo. Sua assinatura é imediatamente finalizada por um ponto e vírgula (;). Por exemplo:

```
abstract class SuperHero {
   abstract void displayPower();
}
class Superman extends SuperHero {
   void displayPower() {
      System.out.println("Fly...");
   }
}

class SpiderMan extends SuperHero {
   void displayPower() {
      System.out.println("Fast...");
   }
}
```

3.13. Interface

Declarar uma interface é basicamente como declarar uma classe, entretanto, ao invés de utilizar a palavra-chave *class*, utilizamos a palavra-chave *interface*. Eis a sintaxe:

```
<declaraçãoInterface> ::=
   <modificador> interface <Nome> {
      <declaraçãoAtributo>*
      [<modificador> <tipoRetorno> <nome>(<argumento>*);]*
   }
```

Os métodos e atributos são obrigatoriamente *public*.

Dicas de programação:

1. Atributos são implicitamente static e final e são obrigados a serem inicializados com um valor constante.
2. Como na declaração de uma classe de alto nível, o único modificador de acesso válido são public e package (caso nenhum modificador de acesso preceder a palavra-chave *class*).

Uma classe pode implementar uma interface existente utilizando a palavra-chave *implements*. Esta classe é obrigada a implementar **todos** os métodos da interface. Uma classe pode implementar mais de uma interface.

O exemplo a seguir demonstra com declarar e utilizar uma interface:

```
interface MyInterface {
   void iMethod();
}
class MyClass1 implements MyInterface {
   public void iMethod() {
      System.out.println("Interface method.");
   }

   void myMethod() {
```

```
      System.out.println("Another method.");
   }
}

class MyClass2 implements MyInterface {
   public void iMethod() {
      System.out.println("Another implementation.");
   }
}

class InterfaceDemo {
   public static void main(String args[]) {
      MyClass1 mc1 = new MyClass1();
      MyClass2 mc2 = new MyClass2();

      mc1.iMethod();
      mc1.myMethod();
      mc2.iMethod();
   }
}
```

A execução desta classe mostrará o seguinte resultado:

```
Interface method.
Another method.
Another implementation.
```

3.14. A palavra-chave this

A palavra-chave *this* pode ser utilizada pela seguintes razões:

1. Diferenciar atributos locais dos atributos de classe.
2. Referenciar objetos que invocam métodos não estáticos.
3. Referenciar outros construtores.

Como exemplo, considere a seguinte classe onde *data* funciona como um atributo e um argumento ao mesmo tempo:

```
class ThisDemo1 {
   int data;
   void method(int data) {
      this.data = data;
      /* this.data refere-se ao atributo
         enquanto data refere-se ao argumento */
   }
}
```

O exemplo seguinte demonstra como este objeto é implicitamente referenciado quando os seus elementos não estáticos são invocados.

```
class ThisDemo2 {
   int data;
   void method() {
      System.out.println(data);     //this.data
   }
   void method2() {
      method();               //this.method();
   }
}
```

Vamos rever o significado de polimorfismo por *overloading*. Um construtor, assim como um

método, pode ser modificado. Métodos diferentes numa mesma classe podem compartilhar o mesmo nome desde que a lista de seus argumentos sejam diferentes. Métodos *overloading* precisam diferir no número ou no tipo de seus argumentos. Este próximo exemplo possui dois construtores e a referência *this* é utilizada para se referenciar a outras versões do construtor.

```
class ThisDemo3 {
   int data;
   ThisDemo3() {
      this(100);
   }
   ThisDemo3(int data) {
      this.data = data;
   }
}
```

Dicas de programação:

1. A chamada para *this()* deve ser a primeira instrução no construtor.

3.15. A palavra-chave super

O uso da palavra-chave *super* está relacionado com herança. É utilizada para chamar construtores da super-classe. Também pode ser utilizada como a palavra-chave *this* para referenciar elementos da super-classe.

A classe a seguir demonstra como a referência *super* é utilizada para chamar o construtor da super-classe.

```
class Person {
   String firstName;
   String lastName;
   Person(String fname, String lname) {
      firstName = fname;
      lastName = lname;
   }
}

class Student extends Person {
   String studNum;
   Student(String fname, String lname, String sNum) {
      super(fname, lname);
      studNum = sNum;
   }
}
```

Dicas de programação:

1. super() refere-se a super-classe imediata. Ela deve ser a primeira instrução do contrutor da subclasse.

Está palavra-chave também pode ser utilizada para referenciar os elementos da super-classe como mostrado no exemplo seguinte.

```
class Superclass{
   int a;
   void display_a(){
```

```
      System.out.println("a = " + a);
   }
}

class Subclass extends Superclass {
   int a;
   void display_a(){
      System.out.println("a = " + a);
   }
   void set_super_a(int n){
      super.a = n;
   }
   void display_super_a(){
      super.display_a();
   }
}

class SuperDemo {
   public static void main(String args[]){
      Superclass SuperObj = new Superclass();
      Subclass SubObj = new Subclass();
      SuperObj.a = 1;
      SubObj.a = 2;
      SubObj.set_super_a(3);
      SuperObj.display_a();
      SubObj.display_a();
      SubObj.display_super_a();
      System.out.println(SubObj.a);
   }
}
```

A execução desta classe mostrará o seguinte resultado:

```
a = 1
a = 2
a = 3
2
```

3.16. A palavra-chave static

A palavra-chave *static* pode ser aplicada a elementos de uma classe. Permite que atributos ou métodos das classes sejam acessados antes que qualquer instância da classe seja criada.

Um atributo de classe possui o acesso global para a classe. Isto significa que o atributo pode ser acessado por todas as instâncias da classe.

Métodos estáticos podem ser invocados sem necessitar criar uma instância da classe. Todavia, eles só acessam os elementos estáticos da classe. Além disso, eles não podem fazer referencia aos objetos *this* ou *super*.

A palavra-chave *static* pode também ser aplicada aos free-blocks. Estes são chamados de blocos estáticos. Estes blocos são executados somente uma vez quando a classe é carregada. Eles são usualmente utilizados para inicializar atributos estáticos da classe.

```
class Demo {
    static int a = 0;
    static void staticMethod(int i) {
        System.out.println(i);
    }
```

```
    static {      //static block
        System.out.println("This is a static block.");
        a += 1;
    }
}

class StaticDemo {
   public static void main(String args[]) {
      System.out.println(Demo.a);
      Demo.staticMethod(5);
      Demo d = new Demo();
      System.out.println(d.a);
      d.staticMethod(0);
      Demo e = new Demo();
      System.out.println(e.a);
      d.a += 3;
      System.out.println(Demo.a+", " +d.a +", " +e.a);
   }
}
```

A execução desta classe mostrará o seguinte resultado:

```
This is a static block.
1
5
1
0
1
4, 4, 4
```

3.17. A palavra-chave final

A palavra-chave *final* pode ser utilizada para atributos, métodos e classes. Para lembrar da função da palavra-chave *final*, simplesmente lembre-se que ela restringe a modificação de atributos, métodos e classes.

O valor de um atributo *final* não pode ser modificado uma vez que esse valor foi determinado. Por exemplo:

```
final int data = 10;
```

A instrução seguinte irá causar um erro de compilação:

```
data++;
```

Um método final não pode ser modificado na classe filha:

```
final void myMethod() { //em uma classe pai
}
```

myMethod não poderá mais ser realizado o polimorfismo por override na classe filha.

Uma classe final não poderá mais ser herdada ao contrário das classes comuns.

```
final public class MyClass {
}
```

Dicas de programação:

1. A ordem de digitação das palavras-chaves *final* e *public* podem ser trocadas.
2. Essa instrução irá provocar um erro de compilação, pois MyClass não pode ser extendida.

```
public WrongClass extends MyClass {
}
```

3.18. Classes internas

Uma classe interna é uma classe declarada dentro de outra classe.

```
class OuterClass {
   int data = 5;
   class InnerClass {
      int data2 = 10;
      void method() {
         System.out.println(data);
         System.out.println(data2);
      }
   }
   public static void main(String args[]) {
      OuterClass oc = new OuterClass();
      InnerClass ic = oc.new InnerClass();
      System.out.println(oc.data);
      System.out.println(ic.data2);
      ic.method();
   }
}
```

A execução desta classe mostrará o seguinte resultado:

```
5
10
```

Para acessarmos os elementos da classe interna, precisamos de uma instância da classe interna. Métodos de uma classe interna podem acessar diretamente os elementos da classe externa.

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.