Módulo 1

Introdução à Programação I

Lição 11

Herança, polimorfismo e interfaces

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 será discutido como uma classe pode herdar as propriedades de outra classe já existente. Uma classe que faz isso é chamada de **subclasse**, e sua classe pai é chamada de **superclasse**. Será também discutida a aplicação automática dos métodos de cada objeto, independente da **subclasse** deste. Esta propriedade é conhecida como polimorfismo. E, por último, discutiremos sobre interfaces, que ajudam a reduzir no esforço de programação.

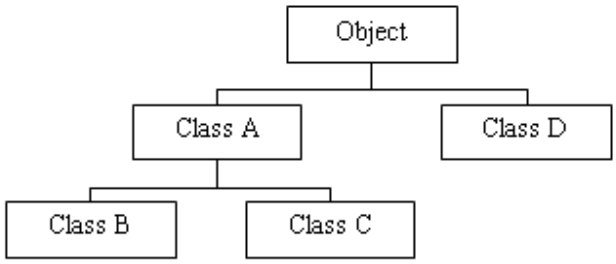
Ao final desta lição, o estudante será capaz de:

- Definir superclasses e subclasses
- Criar override de métodos das superclasses
- Criar métodos final e classes final

2. Herança

Todas as classes, incluindo as que compõem a API Java, são subclasses da classe Object. Um exemplo de hierarquia de classes é mostrado com a figura 1.

A partir de uma determinada classe, qualquer classe acima desta na hierarquia de classes é conhecida como uma **superclasse** (ou classe Pai). Enquanto que qualquer classe abaixo na hierarquia de classes é conhecia como uma **subclasse** (ou classe Filho).



Class hierarchy in Java.

Figura 1: Hierarquia de Classes

Herança é um dos principais princípios em orientação a objeto. Um comportamento (método) é definido e codificado uma única vez em uma única classe e este comportamento é herdado por todas suas subclasses. Uma subclasse precisa apenas implementar as diferenças em relação a sua classe pai, ou seja, adaptar-se ao meio em que vive.

2.1. Definindo Superclasses e Subclasses

Para herdar uma classe usamos a palavra-chave **extends.** Ilustraremos criando uma classe pai de exemplo. Suponha que tenhamos uma classe pai chamada **Person**.

```
public class Person {
    protected String name;
    protected String address;

    /**
     * Construtor Padrão
     */
    public Person(){
        System.out.println("Inside Person:Constructor");
        name = "";
        address = "";
    }
    /**
     * Construtor com 2 parâmetros
     */
    public Person( String name, String address ){
        this.name = name;
        this.address = address;
    }
    /**
     * Métodos modificadores e acessores
     */
    public String getName(){
        return name;
```

```
    }
    public String getAddress(){
        return address;
    }
    public void setName( String name ){
        this.name = name;
    }
    public void setAddress( String add ){
        this.address = add;
    }
}
```

Os atributos **name** e **address** são declarados como **protected**. A razão de termos feito isto é que queremos que estes atributos sejam acessíveis às subclasses dessa classe. Se a declararmos com o modificador **private**, as subclasses não estarão aptas a usá-los. Todas as propriedades de uma superclasse que são declaradas como **public**, **protected** e **default** podem ser acessadas por suas subclasses.

Vamos criar outra classe chamada **Student**. E, como um estudante também é uma pessoa, concluímos que iremos estender a classe **Person**, então, poderemos herdar todas as propriedades existêntes na classe **Person**. Para isto, escrevemos:

```
public class Student extends Person {
      public Student(){
            System.out.println("Inside Student:Constructor");
            //Algum código aqui
      }

      // Algum código aqui
}
```

O fluxo de controle é mostrado na figura 2.

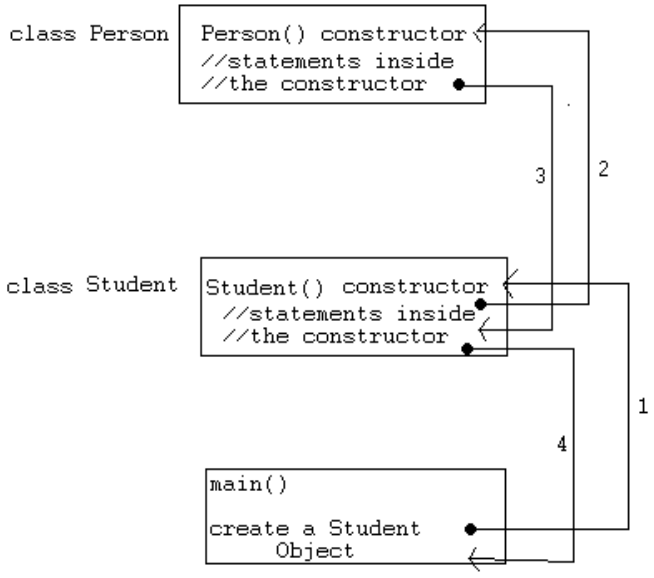


Figura 2: Fluxo do Programa

Quando a classe **Student** for instanciada, o construtor padrão da superclasse **Person** é invocado implicitamente para fazer as inicializações necessárias. Após isso, as instruções dentro do construtor da subclasse são executadas. Para ilustrar, considere o seguinte código:

```
public static void main( String[] args ){
      Student anna = new Student();
}
```

No código, criamos um objeto da classe **Student**. O resultado da execução deste programa é:

```
Inside Person:Constructor
Inside Student:Constructor
```

2.2. super

Uma subclasse pode, explicitamente, chamar um construtor de sua superclasse imediata. Isso é feito utilizando uma chamada ao objeto **super**. Uma chamada ao **super** no construtor de uma subclasse irá resultar na execução de um construtor específico da superclasse baseado nos argumentos passados.

Por exemplo, dada a seguinte instrução para a classe **Student**:

```
public Student(){
      super( "SomeName", "SomeAddress" );
      System.out.println("Inside Student:Constructor");
}
```

Este código chama o segundo construtor de sua superclasse imediata (a classe **Person**) e a executa. Outro código de exemplo é mostrado abaixo:

```
public Student(){
      super();
      System.out.println("Inside Student:Constructor");
}
```

Este código chama o construtor padrão de sua superclasse imediata (a classe **Person**) e o executa.

Devemos relembrar, quando usamos uma chamada ao objeto **super**:

1. A instrução **super()** DEVE SER A PRIMEIRA INSTRUÇÃO EM UM CONSTRUTOR.
2. As instruções **this()** e **super()** não podem ocorrer simultaneamente no mesmo construtor.

O objeto **super** é uma referência aos membros da superclasse (assim como o objeto **this** é da sua própria classe). Por exemplo:

```
public Student() {
      super.name = "person name"; // Nome da classe pai
      this.name = "student name"; // Nome da classe atual
}
```

2.3. Override de Métodos

Se, por alguma razão, uma classe derivada necessita que a implementação de algum método seja diferente da superclasse, o **polimorfismo por override** pode vir a ser muito útil. Uma subclasse pode modificar um método definido em sua superclasse fornecendo uma nova implementação para aquele método.

Supondo que tenhamos a seguinte implementação para o método **getName** da superclasse **Person**:

```
public class Person {
    ...
    public String getName(){
        System.out.println("Parent: getName");
        return name;
    }
    ...
}
```

Para realizar um polimorfismo por override no método **getName** da subclasse **Student**, escrevemos:

```
public class Student extends Person {
    ...
    public String getName(){
        System.out.println("Student: getName");
        return name;
    }
    ...
}
```

Então, quando invocarmos o método **getName** de um objeto da classe **Student**, o método chamado será o de **Student**, e a saída será:

```
Student: getName
```

É possível chamar o método **getName** da superclasse, basta para isso:

```
public class Student extends Person {
    ...
    public String getName() {
        super.getName();
        System.out.println("Student: getName");
        return name;
    }
    ...
}
```

Inserimos uma chamada ao objeto super e a saída será:

```
Parent: getName
Student: getName
```

2.4. Métodos final e Classes final

Podemos declarar classes que não permitem a herança. Estas classes são chamadas classes finais. Para definir que uma classe seja final, adicionamos a palavra-chave **final** na declaração da classe (na posição do modificador). Por exemplo, desejamos que a classe **Person** não possa ser herdada por nenhuma outra classe, escrevemos:

```
public final class Person {
      // Código da classe aqui
}
```

Muitas classes na API Java são declaradas **final** para certificar que seu comportamento não seja herdado e, possivelmente , modificado. Exemplos, são as classes Integer, Double e Math.

Também é possível criar métodos que não possam ser modificados pelos filhos, impedindo o polimorfismo por override. Estes métodos são o que chamamos de métodos finais. Para declarar um método final, adicionamos a palavra-chave final na declaração do método (na posição do modificador). Por exemplo, se queremos que o método getName da classe **Person** não possa ser modificado, escrevemos:

```
public final String getName(){
      return name;
}
```

Caso o programador tente herdar uma classe final, ocorrerá um erro de compilação. O mesmo acontecerá ao se tentar fazer um **override** de um método final.

3. Polimorfismo

Considerando a classe pai **Person** e a subclasse **Student** do exemplo anterior, adicionaremos outra subclasse a **Person**, que se chamará **Employee**. Abaixo está a hierarquia de classes que ilustra o cenário:

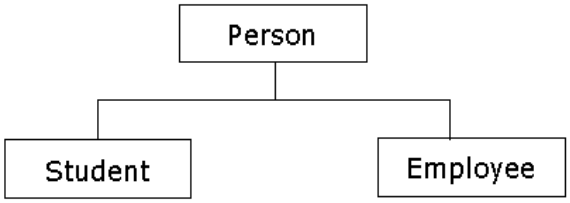


*Figura 3: Hieraquia para a classe **Person** e suas subclasses.*

Podemos criar uma referência do tipo da superclasse para a subclasse. Por exemplo:

```
public static main( String[] args ) {
    Person ref;
    Student studentObject = new Student();
    Employee employeeObject = new Employee();
    ref = studentObject; //Person ref: ponteiro para um Student

    // algum código aqui
}
```

Supondo que tenhamos um método **getName** em nossa superclasse **Person**, iremos realizar uma modificação deste nas subclasses **Student** e **Employee**:

```
public class Person {
    public String getName(){
        System.out.println("Person Name:" + name);
        return name;
    }
}
public class Student extends Person {
    public String getName(){
        System.out.println("Student Name:" + name);
        return name;
    }
}
public class Employee extends Person {
    public String getName(){
        System.out.println("Employee Name:" + name);
        return name;
    }
}
```

Voltando ao método **main**, quando tentamos chamar o método **getName** da referência **ref** do tipo **Person**, o método **getName** do objeto **Student** será chamado. Agora, se atribuirmos **ref** ao objeto **Employee**, o método **getName** de **Employee** será chamado.

```
public static main(String[] args) {
    Person ref;
    Student studentObject = new Student();
    Employee employeeObject = new Employee();
    ref = studentObject; //ponteiro de referência para um Student
    String temp = ref.getName(); //getName de Student é chamado
```

```
        System.out.println(temp);
        ref = employeeObject; // ponteiro de referência Person para um
                              // objeto Employee
        String temp = ref.getName(); // getName de Employee
                                     // classe é chamada
        System.out.println(temp);
    }
```

A capacidade de uma referência mudar de comportamento de acordo com o objeto a que se refere é chamada de **polimorfismo**. O polimorfismo permite que múltiplos objetos de diferentes subclasses sejam tratados como objetos de uma única superclasse, e que automaticamente sejam selecionados os métodos adequados a serem aplicados a um objeto em particular, baseado na subclasse a que ele pertença.

Outro exemplo que demonstra o polimorfismo é realizado ao passar uma referência a métodos. Supondo que exista um método estático **printInformation** que recebe como parâmetro um objeto do tipo **Person**, pode-se passar uma referência do tipo **Employee** e do tipo **Student**, porque são subclasses do tipo **Person**.

```
public static main(String[] args) {
    Student studentObject = new Student();
    Employee employeeObject = new Employee();
    printInformation(studentObject);
    printInformation(employeeObject);
}
public static printInformation(Person p){
    ...
}
```

4. Classes Abstratas

Para criar métodos em classes devemos, necessariamente, saber qual o seu comportamento. Entretanto, em muitos casos não sabemos como estes métodos se comportarão na classe que estamos criando, e, por mera questão de padronização, desejamos que as classes que herdem desta classe possuam, obrigatoriamente, estes métodos.

Por exemplo, queremos criar uma superclasse chamada **LivingThing**. Esta classe tem certos métodos como **breath**, **sleep** e **walk**. Entretanto, existem tantos métodos nesta superclasse que não podemos generalizar este comportamento. Tome por exemplo, o método **walk** (andar). Nem todos os seres vivos andam da mesma maneira. Tomando os humanos como exemplo, os humanos andam sobre duas pernas, enquanto que outros seres vivos como os cães andam sobre quatro. Entretanto, existem muitas características que os seres vivos têm em comum, isto é o que nós queremos ao criar uma superclasse geral.

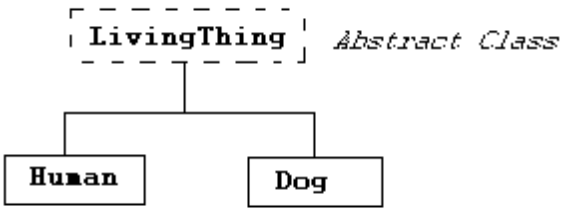


Figura 4: Classe Abstrata

Para realizarmos isto, teremos que criar uma superclasse que possua alguns métodos com implementações e outros não. Este tipo de classe é chamada de **classe abstrata**.

Uma **classe abstrata** é uma classe que não pode gerar um objeto. Freqüentemente aparece no topo de uma hierarquia de classes no modelo de programação orientada a objetos.

Os métodos nas **classes abstratas** que não têm implementação são chamados de **métodos abstratos**. Para criar um método abstrato, apenas escreva a assinatura do método sem o corpo e use a palavra-chave **abstract**. Por exemplo:

```
public abstract void someMethod();
```

Agora, vamos criar um exemplo de classe abstrata:

```
public abstract class LivingThing {
    public void breath(){
        System.out.println("Living Thing breathing...");
    }
    public void eat(){
        System.out.println("Living Thing eating...");
    }
    /**
     * método abstrato walk
     * Queremos que este método seja criado pela
     * subclasse de LivingThing
     */
    public abstract void walk();
}
```

Quando uma classe estende a classe abstrata **LivingThing**, ela é obrigada a implementar o método abstrato **walk**. Por exemplo:

```
public class Human extends LivingThing {
    public void walk(){
        System.out.println("Human walks...");
    }
}
```

Se a classe Human não implementar o método **walk**, será mostrada a seguinte mensagem de erro de compilação:

```
Human.java:1: Human is not abstract and does not override abstract
method walk() in LivingThing
public class Human extends LivingThing
       ^
1 error
```

Dicas de programação:

1. Use classes abstratas para definir muitos tipos de comportamentos no topo de uma hierarquia de classes de programação orientada a objetos. Use suas subclasses para prover detalhes de implementação da classe abstrata.

5. Interfaces

Uma interface é um tipo especial de classe que contém unicamente métodos abstratos ou atributos finais. Interfaces, por natureza, são abstratas.

Interfaces definem um padrão e o caminho público para especificação do comportamento de classes. Permitem que classes, independente de sua localização na estrutura hierárquica, implementem comportamentos comuns.

5.1. Porque utilizar Interfaces?

Utilizamos interfaces quando queremos classes não relacionadas que implementem métodos similares. Através de interfaces, podemos obter semelhanças entre classes não relacionadas sem forçar um relacionamento artificial entre elas.

Tomemos como exemplo a classe **Line**, contém métodos que obtém o tamanho da linha e compara o objeto **Line** com objetos de mesma classe. Considere também que tenhamos outra classe, **MyInteger**, que contém métodos que comparam um objeto **MyInteger** com objetos da mesma classe. Podemos ver que ambas classes têm os mesmos métodos similares que os comparam com outros objetos do mesmo tipo, entretanto eles não são relacionados. Para se ter certeza de que essas classes implementem os mesmos métodos com as mesmas assinaturas, utilizamos as interfaces. Podemos criar uma interface **Relation** que terá declarada algumas assinaturas de métodos de comparação. A interface **Relation** pode ser implementada da seguinte forma:

```
public interface Relation {
    public boolean isGreater(Object a, Object b);
    public boolean isLess(Object a, Object b);
    public boolean isEqual(Object a, Object b);
}
```

Outra razão para se utilizar interfaces na programação de objetos é revelar uma interface de programação de objeto sem revelar essas classes. Como veremos mais adiante, podemos utilizar uma interface como tipo de dados.

Finalmente, precisamos utilizar interfaces como mecanismo alternativo para herança múltipla, que permite às classes em ter mais de uma superclasse. A herança múltipla não está implementada em Java.

5.2. Interface vs. Classe Abstrata

A principal diferença entre uma **interface** e uma **classe abstrata** é que a classe abstrata pode possuir métodos implementados (reais) ou não implementados (abstratos). Na interface, todos os métodos são obrigatoriamente abstratos e públicos, tanto que para esta, a palavra-chave **abstract** ou **public** é opcional.

5.3. Interface vs. Classe

Uma característica comum entre uma interface e uma classe é que ambas são tipos. Isto significa que uma interface pode ser usada no lugar onde uma classe é esperada. Por exemplo,

dadas a classe **Person** e a interface **PersonInterface**, as seguintes declarações são válidas:

```
PersonInterface pi = new Person();
Person pc = new Person();
```

Entretanto, não se pode criar uma instância de uma interface sem implementá-la. Um exemplo disso é:

```
PersonInterface   pi = new PersonInterface(); //ERRO DE
                                              //COMPILAÇÃO !!!
```

Outra característica comum é que ambas, interfaces e classes, podem definir métodos, embora uma interface não possa tê-los implementados. Já uma classe pode.

5.4. Criando Interfaces

Para criarmos uma interface, utilizamos:

```
[public] [abstract] interface <NomeDaInterface> {
    < [public] [final] <tipoAtributo> <atributo> = <valorInicial>; >*
    < [public] [abstract] <retorno> <nomeMetodo>(<parametro>*); >*
}
```

Como exemplo, criaremos uma interface que define o relacionamento entre dois objetos de acordo com a "ordem natural" dos objetos:

```
interface Relation {
    boolean isGreater(Object a, Object b);
    boolean isLess(Object a, Object b);
    boolean isEqual( Object a, Object b);
}
```

Para implementar esta **interface**, usaremos a palavra chave "implements". Por exemplo:

```
/**
 * Esta classe define um segmento de linha
 */
public class Line implements Relation {
    private double x1;
    private double x2;
    private double y1;
    private double y2;
    public Line(double x1, double x2, double y1, double y2) {
        this.x1 = x1;
        this.x2 = x2;
        this.y2 = y2;
        this.y1 = y1;
    }
    public double getLength(){
        double length = Math.sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
        return length;
    }
    public boolean isGreater( Object a, Object b){
        double aLen = ((Line)a).getLength();
        double bLen = ((Line)b).getLength();
        return (aLen > bLen);
```

```
    }
    public boolean isLess( Object a, Object b){
        double aLen = ((Line)a).getLength();
        double bLen = ((Line)b).getLength();
        return (aLen < bLen);
    }
    public boolean isEqual( Object a, Object b){
        double aLen = ((Line)a).getLength();
        double bLen = ((Line)b).getLength();
        return (aLen == bLen);
    }
}
```

Quando a classe implementa uma interface, deve-se implementar **todos** os métodos desta, caso contrário será mostrado o erro:

```
Line.java:4: Line is not abstract and does not override abstract
method isGreater(java.lang.Object,java.lang.Object) in Relation
public class Line implements Relation
       ^
1 error
```

Dicas de programação:

1. Use interface para criar uma definição padrão de métodos em classes diferentes. Uma vez que o conjunto de definições de métodos é criado, e pode ser escrito um método simples para manipular todas as classes que implementam a interface.

5.5. Relacionamento de uma Interface para uma Classe

Como vimos nas seções anteriores, a classe pode implementar uma interface e para isso prover o código de implementação para todos os métodos definidos na interface.

Outro detalhe a se notar na relação entre uma interface e uma classe. A classe pode apenas estender uma única superclasse, mas pode implementar diversas interfaces. Um exemplo de uma classe que implementa diversas interfaces:

```
public class Person
  implements PersonInterface, LivingThing, WhateverInterface {
      //algumas linhas de código
}
```

Outro exemplo de uma classe que estende de outra superclasse e implementa interfaces:

```
public class ComputerScienceStudent extends Student
  implements PersonInterface, LivingThing {
      // algumas linhas de código
}
```

Uma interface não é parte de uma hierarquia de classes. Classes não relacionadas podem implementar a mesma interface.

5.6. Herança entre Interfaces

Interfaces não são partes de uma hierarquia de classes. Entretanto, interfaces podem ter relacionamentos entre si. Por exemplo, suponha que tenhamos duas interfaces, **StudentInterface** e **PersonInterface**. Se **StudentInterface** estende **PersonInterface**, esta herda todos os métodos declarados em **PersonInteface**.

```
public interface PersonInterface {
    ...
}
public interface StudentInterface extends PersonInterface {
    ...
}
```

6. Exercícios

6.1. Estendendo StudentRecord

Neste exercício, queremos criar um registro mais especializado de **Student** que contém informações adicionais sobre um estudante de Ciência da Computação. Sua tarefa é estender a classe **StudentRecord** que foi implementada nas lições anteriores e acrescentar atributos e métodos que são necessários para um registro de um estudante de Ciência da Computação. Utilize **override** para modificar alguns métodos da superclasse **StudentRecord**, caso seja necessário.

6.2. A classe abstrata Shape

Crie uma classe abstrata chamada **Shape** com os métodos abstratos **getArea()** e **getName()**. Escreva duas de suas subclasses **Circle** e **Square**. E acrescente métodos adicionais a estas subclasses.

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.