Módulo 1

Introdução à Programação 1

Lição 10

Criando nossas classes

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

Agora que já estudamos como usar as classes existentes na biblioteca de classes do Java, estudaremos como criar nossas próprias classes. Nesta lição, para facilmente entender como criá-las, realizaremos um exemplo de classe no qual adicionaremos dados e funcionalidades à medida em que avançarmos no curso.

Criaremos uma classe com informações de um Estudante e com operações necessárias para seu registro.

Algumas observações devem ser feitas quanto à sintaxe que será usada nesta e nas próximas seções:

*	-	significa que pode haver nenhuma ou diversas ocorrências na linha em que for aplicada
<descrição>	-	indica a substituição deste trecho por um certo valor, ao invés de digitá-lo como está
[]	-	indica que esta parte é opcional

Ao final desta lição, o estudante será capaz de:

- Criar nossas classes
- Declarar atributos e métodos para as classes
- Usar o objeto **this** para acessar dados de instância
- Utilizar **overloading** de métodos
- Importar e criar pacotes
- Usar modificadores de acesso para controlar o acesso aos elementos de uma classe

2. Definindo nossas classes

Antes de escrever sua classe, primeiro pense onde e como sua classe será usada. Pense em um nome apropriado para a classe e liste todas as informações ou propriedades que deseje que ela tenha. Liste também os métodos que serão usados para a classe.

Para definir uma classe, escrevemos:

```
<modificador>* class <nome> {
      <declaraçãoDoAtributo>*
      <declaraçãoDoConstrutor>*
      <declaraçãoDoMétodo>*
}
```

onde:

<modificador>	é um modificador de acesso, que pode ser usado em combinação com outros
<nome>	nome da sua classe
<declaraçãoDoAtributo>	atributos definidos para a classe
<declaraçãoDoConstrutor>	método construtor
<declaraçãoDoMétodo>	métodos da classe

Dicas de programação:

1. Lembre-se de que, para a declaração da classe, o único modificador de acesso válido é o **public**. De uso exclusivo para a classe que possuir o mesmo nome do arquivo externo.

Nesta lição, criaremos uma classe que conterá o registro de um **estudante**. Como já identificamos o objetivo da nossa classe, agora podemos nomeá-la. Um nome apropriado para nossa classe seria **StudentRecord**.

Para definir nossa classe, escrevemos:

```
public class StudentRecord {
       // adicionaremos mais código aqui
}
```

onde:

public	modificador de acesso e significa que qualquer classe pode acessar esta
class	palavra-chave usada para criar uma classe
StudentRecord	identificador único que identifica a classe

Dicas de programação:

1. Pense em nomes apropriados para a sua classe. Não a chame simplesmente de classe XYZ ou qualquer outro nome aleatório.
2. Os nomes de classes devem ser iniciadas por letra MAIÚSCULA.
3. O nome do arquivo de sua classe obrigatoriamente possui o MESMO NOME da sua classe pública.

3. Declarando Atributos

Para declarar um certo atributo para a nossa classe, escrevemos:

```
<modificador>* <tipo> <nome> [= <valorInicial>];
```

onde:

modificador	tipo de modificador do atributo
tipo	tipo do atributo
nome	pode ser qualquer identificador válido
valorInicial	valor inicial para o atributo

Relacionaremos a lista de atributos que um registro de estudante pode conter. Para cada informação, listaremos os tipos de dados apropriados que serão utilizados. Por exemplo, não seria ideal usar um tipo **int** para o **nome do estudante** ou **String** para a **nota do estudante**.

Abaixo, por exemplo, temos algumas informações que podemos adicionar ao registro do estudante:

nome	-	String
endereço	-	String
idade	-	int
nota de matemática	-	double
nota de inglês	-	double
nota de ciências	-	double

Futuramente, é possível adicionar mais informações. Para este exemplo, utilizaremos somente estas.

3.1. Atributos de Objeto

Agora que temos uma lista de todos os atributos que queremos adicionar à nossa classe, vamos adicioná-los ao nosso código. Uma vez que queremos que estes atributos sejam únicos para cada objeto (ou para cada estudante), devemos declará-los como atributos de objeto.

Por exemplo:

```
public class StudentRecord {
    private String name;
    private String address;
    private int    age;
    private double mathGrade;
```

```
    private double englishGrade;
    private double scienceGrade;
}
```

onde:

private significa que os atributos são acessíveis apenas de dentro da classe. Outros objetos não podem acessar diretamente estes atributos.

Dicas de programação:

1. Declare todas os atributos de objeto na parte superior da declaração da classe.
2. Declare cada atributo em uma linha.
3. Atributos de objeto, assim como qualquer outro atributo devem iniciar com letra MINÚSCULA.
4. Use o tipo de dado apropriado para cada atributo declarado.
5. Declare atributos de objetos como **private** de modo que somente os métodos da classe possam acessá-los diretamente.

3.2. Atributos de Classe ou Atributos Estáticos

Além das atributos de objeto, podemos também declarar atributos de classe ou atributos que pertençam à classe como um todo. O valor destes atributos é o mesmo para todos os objetos da mesma classe. Suponha que queiramos saber o número total de registros criados para a classe. Podemos declarar um atributo estático que armazenará este valor. Vamos chamá-lo de **studentCount**.

Para declarar um atributo estático:

```
public class StudentRecord {
    // atributos de objeto declarados anteriormente

    private static int studentCount;
}
```

usamos a palavra-chave **static** para indicar que é um atributo estático.

Então, nosso código completo deve estar assim:

```
public class StudentRecord {
    private String name;
    private String address;
```

```
        private int age;
        private double mathGrade;
        private double englishGrade;
        private double scienceGrade;

        private static int studentCount;
    }
```

4. Declarando Métodos

Antes de discutirmos quais métodos que a nossa classe deverá conter, vejamos a sintaxe geral usada para a declaração de métodos.

Para declararmos métodos, escrevemos:

```
<modificador>* <tipoRetorno> <nome>(<argumento>*) {
      <instruções>*
}
```

onde:

<modificador>	pode ser utilizado qualquer modificador de acesso
<tipoRetorno>	pode ser qualquer tipo de dado (incluindo **void**)
<nome>	pode ser qualquer identificador válido
<argumento>	argumentos recebidos pelo método separados por vírgulas. São definidos por: `<tipoArgumento> <nomeArgumento>`

4.1. Métodos assessores

Para que se possa implementar o princípio do encapsulamento, isto é, não permitir que quaisquer objetos acessem os nossos dados de qualquer modo, declaramos campos, ou atributos, da nossa classe como particulares. Entretanto, há momentos em que queremos que outros objetos acessem estes dados particulares. Para que possamos fazer isso, criamos **métodos assessores**.

Métodos assessores são usados para ler valores de atributos de objeto ou de classe. O método assessor recebe o nome de **get<NomeDoAtributo>**. Ele retorna um valor.

Para o nosso exemplo, queremos um método que possa ler o nome, endereço, nota de inglês, nota de matemática e nota de ciências do estudante.

Vamos dar uma olhada na implementação deste:

```
public class StudentRecord {
    private String name;
    :
    :
    public String getName() {
```

```
            return name;
        }
    }
```

onde:

public	significa que o método pode ser chamado por objetos externos à classe
String	é o tipo do retorno do método. Isto significa que o método deve retornar um valor de tipo **String**
getName	o nome do método
()	significa que o nosso método não tem nenhum argumento

A instrução:

```
return name;
```

no método, significa que retornará o conteúdo do atributo **name** ao método que o chamou. Note que o tipo do retorno do método deve ser do mesmo tipo do atributo utilizado na declaração **return**. O seguinte erro de compilação ocorrerá caso o método e o atributo de retorno não tenham o mesmo tipo de dados:

```
StudentRecord.java:14: incompatible types
found   : int
required: java.lang.String
                return name;
                       ^
1 error
```

Outro exemplo de um método assessor é o método getAverage:

```
public class StudentRecord {
    private String name;
    :
    :
    public double getAverage(){
        double result = 0;
        result =
          (mathGrade+englishGrade+scienceGrade)/3;
        return result;
    }
}
```

O método **getAverage** calcula a média das 3 notas e retorna o resultado.

4.2. Métodos modificadores

Para que outros objetos possam modificar os nossos dados, disponibilizamos métodos que possam gravar ou modificar os valores dos atributos de objeto ou de classe. Chamamos a estes métodos **modificadores**. Este método é escrito como **set<NomeDoAtributoDeObjeto>**.

Vamos dar uma olhada na implementação de um método modificador:

```
public class StudentRecord {
    private String name;
    :
    :
    public void setName(String temp) {
        name = temp;
    }
}
```

onde:

public	significa que o método pode ser chamado por objetos externos à classe
void	significa que o método não retorna valor
setName	o nome do método
(String temp)	argumento que será utilizado dentro do nosso método

A instrução:

```
name = temp;
```

atribuir o conteúdo de **temp** para **name** e, portanto, alterar os dados dentro do atributo de objeto **name**.

Métodos modificadores não retornam valores. Entretanto, eles devem receber um argumento com o mesmo tipo do atributo no qual estão tratando.

4.3. Múltiplos comandos return

É possível ter vários comandos **return** para um método desde que eles não pertençam ao mesmo bloco. É possível utilizar constantes para retornar valores, ao invés de atributos.

Por exemplo, considere o método:

```
public String getNumberInWords(int num) {
    String defaultNum = "zero";
```

```
        if (num == 1) {
            return "one"; // retorna uma constante
        } else if( num == 2) {
            return "two"; // retorna uma constante
        }
        // retorna um atributo
        return defaultNum;
    }
```

4.4. Métodos estáticos

Para o atributo estático **studentCount**, podemos criar um método estático para obter o seu conteúdo.

```
public class StudentRecord {
   private static int studentCount;

   public static int getStudentCount(){
      return studentCount;
   }
}
```

onde:

public	significa que o método pode ser chamado por objetos externos à classe
static	significa que o método é estático e deve ser chamado digitando-se [NomeClasse].[nomeMétodo]
int	é o tipo do retorno do método. Significa que o método deve retornar um valor de tipo **int**
getStudentCount	nome do método
()	significa que o método não tem nenhum argumento

Por enquanto, **getStudentCount** retornará sempre o valor zero já que ainda não fizemos nada na nossa classe para atribuir o seu valor. Modificaremos o valor de **studentCount** mais tarde, quando discutirmos construtores.

Dicas de programação:

1. Nomes de métodos devem iniciar com letra MINÚSCULA.
2. Nomes de métodos devem conter verbos
3. Sempre faça documentação antes da declaração do método. Use o estilo **javadoc** para isso.

4.5. Exemplo de Código Fonte para a classe StudentRecord

Aqui está o código para a nossa classe **StudentRecord**:

```
public class StudentRecord {
    private String name;
    private String address;
    private int    age;
    private double mathGrade;
    private double englishGrade;
    private double scienceGrade;

    private static int studentCount;

    /**
     * Retorna o nome do estudante
     */
    public String getName(){
        return name;
    }

    /**
     * Muda o nome do estudante
     */
    public void setName( String temp ){
        name = temp;
    }

    // outros métodos modificadores aqui ....

    /**
     * Calcula a média das classes de inglês, matemática
     * e ciências
     */
    public double getAverage(){
        double result = 0;
        result =
          (mathGrade+englishGrade+scienceGrade)/3;
        return result;
    }
    /**
     * Retorna o número de ocorrências em StudentRecords
     */
    public static int getStudentCount(){
        return studentCount;
    }
}
```

Aqui está um exemplo do código de uma classe que utiliza a nossa classe

StudentRecord.

```
public class StudentRecordExample {
    public static void main( String[] args ){
        // criar três objetos para StudentRecord
        StudentRecord annaRecord = new StudentRecord();
        StudentRecord beahRecord = new StudentRecord();
        StudentRecord crisRecord = new StudentRecord();

        // enviar o nome dos estudantes
        annaRecord.setName("Anna");
        beahRecord.setName("Beah");
        crisRecord.setName("Cris");

        // mostrar o nome de anna
        System.out.println(annaRecord.getName());

        //mostrar o número de estudantes
        System.out.println("Count=" +
            StudentRecord.getStudentCount());
    }
}
```

A saída desta classe é:

```
Anna
Count = 0
```

5. this

O objeto ***this*** é usado para acessar atributos de objeto ou métodos da classe. Para entender isso melhor, tomemos o método **setAge** como exemplo. Suponha que tenhamos o seguinte método para **setAge**:

```
public void setAge(int age){
    age = age; // Não é uma boa prática
}
```

O nome do argumento nesta declaração é **age**, que tem o mesmo nome do atributo de objeto **age**. Já que o argumento **age** é a declaração mais próxima do método, o valor do argumento **age** será usado. Na instrução:

```
age = age;
```

estamos simplesmente associando o valor do argumento **age** para si mesmo! Isto não é o que queremos que aconteça no nosso código. A fim de corrigir esse erro, usamos o objeto **this**. Para utilizar o objeto **this**, digitamos:

```
this.<nomeDoAtributo>
```

O ideal é reescrever o nosso método do seguinte modo:

```
public void setAge(int age){
    this.age = age;
}
```

Este método irá atribuir o valor do argumento **age** para a atributo de objeto **age** do objeto **StudentRecord**.

6. Overloading de Métodos

Nas nossas classes, podemos necessitar de criar métodos que tenham os mesmos nomes, mas que funcionem de maneira diferente dependendo dos argumentos que informamos. Esta capacidade é chamada de **overloading de métodos**.

Overloading de métodos permite que um método com o mesmo nome, entretanto com diferentes argumentos, possa ter implementações diferentes e retornar valores de diferentes tipos. Ao invés de inventar novos nomes todas as vezes, o overloading de métodos pode ser utilizado quando a mesma operação tem implementações diferentes.

Por exemplo, na nossa classe **StudentRecord**, queremos ter um método que mostre as informações sobre o estudante. Entretanto, queremos que o método **print** mostre dados diferentes dependendo dos argumentos que lhe informamos. Por exemplo, quando não enviamos qualquer argumento queremos que o método **print** mostre o nome, endereço e idade do estudante. Quando passamos 3 valores **double**, queremos que o método mostre o nome e as notas do estudante.

Temos os seguintes métodos dentro da nossa classe **StudentRecord**:

```
public void print(){
    System.out.println("Name:" + name);
    System.out.println("Address:" + address);
    System.out.println("Age:" + age);
}

public void print(double eGrade, double mGrade, double
sGrade)
    System.out.println("Name:" + name);
    System.out.println("Math Grade:" + mGrade);
    System.out.println("English Grade:" + eGrade);
    System.out.println("Science Grade:" + sGrade);
}
```

Quando tentamos chamar estes métodos no método **main**, criado para a classe **StudantRecordExample:**

```
public static void main(String[] args) {
    StudentRecord annaRecord = new StudentRecord();

    annaRecord.setName("Anna");
    annaRecord.setAddress("Philippines");
    annaRecord.setAge(15);
```

```
        annaRecord.setMathGrade(80);
        annaRecord.setEnglishGrade(95.5);
        annaRecord.setScienceGrade(100);

        // overloading de métodos
        annaRecord.print();
        annaRecord.print(
            annaRecord.getEnglishGrade(),
            annaRecord.getMathGrade(),
            annaRecord.getScienceGrade());
    }
```

teremos a saída para a primeira chamada ao método **print**:

```
Name:Anna
Address:Philippines
Age:15
```

e, em seguida, a saída para a segunda chamada ao método **print**:

```
Name:Anna
Math Grade:80.0
English Grade:95.5
Science Grade:100.0
```

Lembre-se sempre que métodos **overload** possuem as seguintes propriedades:

1. o mesmo nome
2. argumentos diferentes
3. tipo do retorno igual ou diferente

7. Declarando Construtores

Discutimos anteriormente o conceito de construtores. Construtores são importantes na criação de um objeto. É um método onde são colocadas todas as inicializações.

A seguir, temos as propriedades de um construtor:

1. Possuem o **mesmo nome da classe**
2. Construtor é um método, entretanto, somente as seguintes informações podem ser colocadas no cabeçalho do construtor:
   - Escopo ou identificador de acessibilidade (como **public**)
   - Nome do construtor
   - Argumentos, caso necessário
3. Não retornam valor
4. São executados automaticamente na utilização do operador **new** durante a instanciação da classe

Para declarar um construtor, escrevemos:

```
[modificador] <nomeClasse> (<argumento>*) {
    <instrução>*
}
```

7.1. Construtor Padrão (default)

Toda classe tem o seu construtor padrão. O **construtor padrão** é um construtor público e sem argumentos. Se não for definido um construtor para a classe, então, implicitamente, é assumido um construtor padrão.

Por exemplo, na nossa classe **StudentRecord**, o construtor padrão é definido do seguinte modo:

```
public StudentRecord() {
}
```

7.2. Overloading de Construtores

Como mencionamos, construtores também podem sofrer **overloading**, por exemplo, temos aqui quatro construtores:

```
public StudentRecord() {
    // qualquer código de inicialização aqui
}
public StudentRecord(String temp){
```

```
        this.name = temp;
    }
    public StudentRecord(String name, String address) {
        this.name = name;
        this.address = address;
    }
    public StudentRecord(double mGrade, double eGrade,
        double sGrade) {
        mathGrade = mGrade;
        englishGrade = eGrade;
        scienceGrade = sGrade;
    }
```

7.3. Usando Construtores

Para utilizar estes construtores, temos as seguintes instruções:

```
public static void main(String[] args) {
    // criar três objetos para o registro do estudante
    StudentRecord annaRecord = new
        StudentRecord("Anna");
    StudentRecord beahRecord =
        new StudentRecord("Beah", "Philippines");
    StudentRecord  crisRecord =
        new StudentRecord(80,90,100);
    // algum código aqui
}
```

Antes de continuarmos, vamos retornar o atributo estático **studentCount** que declaramos agora a pouco. O objetivo de **studentCount** é contar o número de objetos que são instanciados com a classe **StudentRecord**. Então, o que desejamos é incrementar o valor de **studentCount** toda vez que um objeto da classe **StudentRecord** é instanciado. Um bom local para modificar e incrementar o valor de **studentCount** é nos construtores, pois são sempre chamados toda vez que um objeto é instanciado. Por exemplo:

```
public StudentRecord() {
    studentCount++; // adicionar um estudante
}
public StudentRecord(String name) {
    studentCount++; // adicionar um estudante
    this.name = name;
}
public StudentRecord(String name, String address) {
    studentCount++; // adicionar um estudante
    this.name = name;
    this.address = address;
```

```
}
public StudentRecord(double mGrade, double eGrade,
    double sGrade) {
    studentCount++; // adicionar um estudante
    mathGrade = mGrade;
    englishGrade = eGrade;
    scienceGrade = sGrade;
}
```

7.4. Utilizando o this()

Chamadas a construtores podem ser cruzadas, o que significa ser possível chamar um construtor de dentro de outro construtor. Usamos a chamada **this()** para isso. Por exemplo, dado o seguinte código,

```
public StudentRecord() {
    this("some string");
}
public StudentRecord(String temp) {
    this.name = temp;
}
public static void main( String[] args ) {
    StudentRecord annaRecord = new StudentRecord();
}
```

Dado o código acima, quando se executa a instrução do método **main**, será chamado o **primeiro** construtor. A instrução inicial deste construtor resultará na chamada ao **segundo** construtor.

Há algum detalhes que devem ser lembrados na utilização da chamada ao construtor por **this()**:

1. A chamada ao construtor **DEVE SEMPRE OCORRER NA PRIMEIRA LINHA DE INSTRUÇÃO**
2. **UTILIZADO PARA A CHAMADA DE UM CONSTRUTOR**. A chamada ao **this()** pode ser seguida por outras instruções.

Como boa prática de programação, é ideal nunca construir métodos que repitam as instruções. Buscamos a utilização de **overloading** com o objetivo de evitarmos essa repetição. Deste modo, reescreveremos os construtores da classe **StudentRecord** para:

```
public StudentRecord() {
    studentCount++; // adicionar um estudante
}
public StudentRecord(String name) {
    this();
```

```
        this.name = name;
    }
    public StudentRecord(String name, String address) {
        this(name);
        this.address = address;
    }
    public StudentRecord(double mGrade, double eGrade,
        double sGrade) {
        this();
        mathGrade = mGrade;
        englishGrade = eGrade;
        scienceGrade = sGrade;
    }
```

8. Pacotes

São utilizados para agrupar classes e interfaces relacionadas em uma única unidade (discutiremos interfaces mais tarde). Esta é uma característica poderosa que oferece um mecanismo para gerenciamento de um grande grupo de classes e interfaces e evita possíveis conflitos de nome.

8.1. Importando Pacotes

Para utilizar classes externas ao pacote da classe, é necessário importar os pacotes dessas classes. Por padrão, todos as suas classes Java importam o pacote **java.lang**. É por isso que é possível utilizar classes como *String* e *Integer* dentro da sua classe, mesmo não tendo importado nenhum pacote explicitamente.

A sintaxe para importar pacotes é como segue:

```
import <nomeDoPacote>.<nomeDaClasse>;
```

Por exemplo, necessitar utilizar a classe **Color** dentro do pacote **awt**, é necessário a seguinte instrução:

```
import java.awt.Color;
```

ou:

```
import java.awt.*;
```

A primeira linha de instrução importa especificamente a classe **Color** enquanto que a seguinte importa todas as classes do pacote **java.awt**.

Outra maneira de importar classes de outros pacotes é através da referência explícita ao pacote. Isto é feito utilizando-se o nome completo do pacote para declaração do objeto na classe:

```
java.awt.Color color;
```

8.2. Criando pacotes

Para criar os nossos pacotes, escrevemos:

```
package <nomeDoPacote>;
```

Suponha que desejamos criar um pacote onde colocaremos a nossa classe **StudentRecord** juntamente com outras classes relacionadas. Chamaremos o

nosso pacote de **schoolClasses**.

A primeira coisa que temos que fazer é criar uma pasta chamada **schoolClasses**. Em seguida, copiar para esta pasta todas as classes que pertençam a este pacote. Adicione a seguinte instrução no arquivo da classe, esta linha deve ser colocada antes da definição da classe. Por exemplo:

```
package schoolClasses;

public class StudentRecord {
    // instruções da classe
}
```

Pacotes podem ser aninhados. Neste caso, o interpretador espera que a estrutura de diretórios contendo as classes combinem com a hierarquia dos pacotes.

8.3. Definindo a variável de ambiente CLASSPATH

Suponha que colocamos o pacote **schoolClasses** sob o diretório C:\. Precisamos que a **classpath** aponte para este diretório de tal forma que quando executemos a classe, a JVM seja capaz de enxergar onde está armazenada.

Antes de discutirmos como ajustar a variável **classpath**, vamos ver um exemplo sobre o que aconteceria se esta não fosse ajustada.

Suponha que sigamos os passos para compilar e executar a classe **StudentRecord** que escrevemos:

```
C:\schoolClasses>javac StudentRecord.java

C:\schoolClasses>java StudentRecord
Exception in thread "main" java.lang.NoClassDefFoundError: StudentRecord
(wrong name: schoolClasses/StudentRecord)
        at java.lang.ClassLoader.defineClass1(Native Method)
        at java.lang.ClassLoader.defineClass(Unknown Source)
        at java.security.SecureClassLoader.defineClass(Unknown Source)
        at java.net.URLClassLoader.defineClass(Unknown Source)
        at java.net.URLClassLoader.access$100(Unknown Source)
        at java.net.URLClassLoader$1.run(Unknown Source)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClassInternal(Unknown Source)
```

Surge o erro **NoClassDefFoundError**, que significa que o Java desconhece onde procurar por esta classe. A razão disso é que a sua classe

StudentRecord pertence a um pacote denominado **schoolClasses**. Se desejamos executar esta classe, teremos que dizer ao Java o seu nome completo **schoolClasses.StudentRecord**. Também teremos que dizer à JVM onde procurar pelos nossos pacotes, que, neste caso, é no C:\. Para fazer isso, devemos definir a variável **classpath**.

Para definir a variável **classpath** no Windows, digitamos o seguinte na linha de comando:

```
C:\schoolClasses>set classpath=C:\
```

onde C:\ é o diretório onde colocamos os pacotes. Após definir a variável **classpath**, poderemos executar a nossa classe em qualquer pasta, digitando:

```
C:\schoolClasses>java schoolClasses.StudentRecord
```

Para sistemas baseados no Unix, suponha que as nossas classes estejam no diretório **/usr/local/myClasses**, escrevemos:

```
export classpath=/usr/local/myClasses
```

Observe que é possível definir a variável **classpath** em qualquer lugar. É possível definir mais de um local de pesquisa; basta separá-los por ponto-e-vírgula (no Windows) e dois-pontos (nos sistemas baseados em Unix). Por exemplo:

```
set classpath=C:\myClasses;D:\;E:\MyPrograms\Java
```

e para sistemas baseados no Unix:

```
export classpath=/usr/local/java:/usr/myClasses
```

9. Modificadores de Acesso

Quando estamos criando as nossas classes e definindo as suas propriedades e métodos, queremos implementar algum tipo de restrição para se acessar esses dados. Por exemplo, ao necessitar que um certo atributo seja modificado apenas pelos métodos dentro da classe, é possível esconder isso dos outros objetos que estejam usando a sua classe. Para implementar isso, no Java, temos os **modificadores de acesso**.

Existem quatro diferentes tipos de modificadores de acesso: **public**, **private**, **protected** e **default**. Os três primeiros modificadores são escritos explicitamente no código para indicar o acesso, para o tipo **default**, não se utiliza nenhuma palavra-chave.

9.1. Acesso padrão

Especifica que os elementos da classe são acessíveis somente aos métodos internos da classe e às suas subclasses. Não há palavra chave para o modificador **default**; sendo aplicado na ausência de um modificador de acesso. Por exemplo :

```
public class StudentRecord {
    // acesso padrão ao atributo
    int name;

    // acesso padrão para o método
    String getName(){
        return name;
    }
}
```

O atributo de objeto **name** e o método **getName()** podem ser acessados somente por métodos internos à classe e por subclasses de **StudentRecord**. Falaremos sobre subclasses em próximas lições.

9.2. Acesso público

Especifica que os elementos da classe são acessíveis tanto internamente quanto externamente à classe. Qualquer objeto que interage com a classe pode ter acesso aos elementos públicos da classe. Por exemplo:

```
public class StudentRecord {
    // acesso público o atributo
    public int name;

    // acesso público para o método
```

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

O atributo de objeto **name** e o método **getName()** podem ser acessados a partir de outros objetos.

9.3. Acesso protegido

Especifica que somente classes no mesmo pacote podem ter acesso aos atributos e métodos da classe. Por exemplo:

```
public class StudentRecord {
    //acesso protegido ao atributo
    protected int name;

    //acesso protegido para o método
    protected String getName(){
        return name;
    }
}
```

O atributo de objeto **name** e o método **getName()** podem ser acessados por outros objetos, desde que o objetos pertençam ao mesmo pacote da classe **StudentRecord**.

9.4. Acesso particular

Especifica que os elementos da classe são acessíveis somente na classe que o definiu. Por exemplo:

```
public class StudentRecord {
    // acesso particular ao atributo
    private int name;

    // acesso particular para o método
    private String getName(){
        return name;
    }
}
```

O atributo de objeto **name** e o método **getName()** podem ser acessados somente por métodos internos à classe.

Dicas de programação:

1. Normalmente, os atributos de objeto de uma classe devem ser declarados particulares e a classe pode fornecer métodos assessores e modificadores para estes.

10. Exercícios

10.1. Registro de Agenda

Sua tarefa é criar uma classe que contenha um Registro de Agenda. A **tabela 1** descreve as informações que um Registro de Agenda deve conter:

Atributos/Propriedades	Descrição
Nome	Nome da pessoa
Endereço	Endereço da pessoa
Número de Telefone	Número de telefone da pessoa
email	Endereço eletrônico da pessoa

Tabela 1: Atributos e Descrições dos Atributos

Crie os seguintes métodos:

1. Forneça todos os métodos assessores e modificadores necessários para todos os atributos.
2. Construtores.

10.2. Agenda

Crie uma classe Agenda que possa conter entradas de objetos tipo Registro de Agenda (utilize a classe criada no primeiro exercício). Devem ser oferecidos os seguintes métodos para a agenda:

1. Adicionar registro
2. Excluir registro
3. Visualizar registros
4. Modificar um registro

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.