Membros da classe

Membros de Classe

- Quando um objeto é criado (instanciado), cada um terá seu conjunto de variáveis de instância
  - Essas variáveis serão alocadas para cada objeto diferente
- As variáveis de classe são comuns a todos os objetos
  - Uma única variável na memória, cuja referência é compartilhada por todas as instâncias
- Variáveis de classe são declaradas utilizando a palavra chave *static*
  - Campos estáticos

unesp

Membros de Classe

- Variáveis de classe podem ser manipuladas por qualquer instância da classe, mas não há a necessidade de ter uma instância
  - Ela existe independente de qualquer instância

```
myObject.staticField;
MyClass.staticField;   <-------  preferível
```

- Exemplo de aplicação
  - Suponha que queiramos associar um ID para cada objeto Bycicle que for criado (em qualquer lugar)
  - Queremos fazer isso de forma serial, ou seja: 1, 2, 3, ...

Membros de Classe

-

```
public class Bicycle {
    private int cadence;
    private int gear;
    private int speed;
    private int id;
    private static int numberOfBicycles = 0;

    public Bicycle(int cadence, int speed, int gear){
        this.gear = gear;
        this.cadence = cadence;
        this.speed = speed;
        id = ++numberOfBicycles;
    }

    public int getID() {
        return id;
    }
    ...
}
```

Membros de Classe

- Assim como campos, é possível definir métodos de classe (ou métodos estáticos)
- Valem as mesmas regras para campos
  - Métodos estáticos existem independentemente das instâncias
  - A chamada à métodos estáticos deve ser feita pelo nome da classe (convenção)
- Uma aplicação comum de métodos estáticos é o acesso à campos estáticos

```
public static int getNumberOfBicycles() {
    return numberOfBicycles;
}
```

unesp

Membros de Classe

- Considerações importantes sobre acessos entre membros de instância e de classes
  - Métodos de instância PODEM acessar variáveis e métodos de instância diretamente
  - Métodos de instância PODEM acessar variáveis e métodos de classe diretamente
  - Métodos de classe PODEM acessar variáveis e métodos de classe diretamente
  - Métodos de classe **NÃO PODEM** acessar variáveis e métodos de instância diretamente
  - Métodos de classe **NÃO PODEM** usar a palavra-chave *this*, pois não há instância que *this* deva representar

Membros de Classe

-

```
public class Bicycle {
    private int cadence;
    private int gear;
    private int speed;
    private int id;
    private static int numberOfBicycles = 0;

    public Bicycle(int cadence, int speed, int gear){
        this.gear = gear;
        this.cadence = cadence;
        this.speed = speed;
        id = ++numberOfBicycles;
    }

    public static void printBikeId() {
        System.out.println("Bike number " + id);
    }
    ...
}
```

Membros de Classe

- Constantes
  - A combinação dos modificadores *static* e *final* é usada para criar constantes
  - Modificador *final* indica que a variável não pode ser alterada
    - Seu valor deve ser definido junto com a declaração
    - Tentar alterar uma constante gera erro de compilação
  - Por que usar modificar *static* para definir constantes?
    - Cada instância teria uma variável constante

unesp

Inicializando Variáveis de Classe

- A inicialização de variáveis de instâncias pode ser feita de duas formas
  - Associando valores à variável no mesmo comando de sua declaração
  - Dentro de um construtor

```
public class Bicycle {
    private int cadence = 10;
    private int gear = 0;
    private int speed = 0;

    public Bicycle() {}
}
```

```
public class Bicycle {
    private int cadence;
    private int gear;
    private int speed;

    public Bicycle(){
        gear = 0;
        cadence = 10;
        speed = 0;
    }
}
```

Inicializando Variáveis de Classe

- A inicialização de variáveis de instâncias pode ser feita de duas formas
  - Associando valores à variável no mesmo comando de sua declaração
  - Dentro de um construtor
- Para variáveis de classe (*static*), só a primeira opção é possível, já que construtores só fazem sentido para instâncias
- O que fazer quando uma variável de classe exige uma inicialização um pouco mais complexa?
  - Por exemplo, copiar elementos de um array

unesp

Inicializando Variáveis de Classe

- Java provê um bloco de inicialização estático

```
static {
    // whatever code is needed for initialization goes here
}
```

- Uma classe pode conter vários blocos deste tipo
- Eles podem estar em qualquer posição no corpo da classe
- O compilador irá chamar todos os blocos estáticos, na ordem em que foram definidos

Inicializando Variáveis de Classe

- Uma forma alternativa ao bloco estático é usar um método privado e estático
  - A vantagem é que o método pode ser chamado novamente em alguma situação futura, para reinicializar as variáveis de classe

```
class Whatever {
    public static varType MY_VAR = initializeClassVariable();

    private static varType initializeClassVariable() {
        // initialization code goes here
    }
}
```

unesp

Inicio do Programa

- Todo projeto em Java precisa de um método público e estático chamado **main**
- Este método deve ser declarado dentro de alguma classe
- Contudo, por ser estático, não há necessidade de uma instância para chamá-lo
  - No início do programa, não há instâncias de nenhuma classe

```
public class Principal {
    public static void main(String[] args) {
        // the program starts here
    }
}
```

unesp

Pacotes

Pacotes

- A estrutura de pacotes é importante para organizar classes relacionadas de alguma forma
  - Lembre-se que os modificadores protected e *package-private* estão relacionados aos pacotes
  - Facilita para o programador na hora de buscar por classes que realizam determinada tarefa
  - Exemplos:
    - java.lang
    - java.io
  - Duas (ou mais) classes podem ter o mesmo nome se estiverem em pacotes diferentes

unesp

Pacotes

- Convenção para nomes
  - Todas as letras minúsculas
  - Empresas devem criar uma estrutura de pacotes que use o endereço web ao contrário
  - Exemplo: www.example.com
    - Se a empresa acima criar um pacote chamado meupacote, ele seria criado em com.example.meupacote
  - Todos os pacotes da API java estão nos pacotes java. ou javax.

Pacotes

- O pacote do qual a classe faz parte precisa ser declarado no início do código-fonte que define a classe
- Um pacote também reflete a estrutura de diretórios do código fonte
  - Essa estrutura deve ser respeitada, caso contrário o código não compila

```
package veiculo;

public class Bicicleta {
    // corpo de classe
}
```

unesp

Importação de Pacotes e Classes

- Para usar uma classe é preciso importá-la, colocando todo seu caminho na estrutura dos pacotes
  - Importações vem depois do comando package

```
package veiculo;

import java.util.Scanner;

public class Bicicleta {
      Scanner sc = new Scanner(System.in);
      ...
 }
```

unesp

Importação de Pacotes e Classes

- Para usar uma classe é preciso importá-la, colocando todo seu caminho na estrutura dos pacotes
  - Importações vem depois do comando package
  - Também é possível importar **TODAS** as classe de um pacote de uma vez

```
package veiculo;

import java.util.*;

public class Bicicleta {
    Scanner sc = new Scanner(System.in);
    …
}
```

unesp

JavaDoc

JavaDoc

- A documentação do código é identificada por um tipo especial de comentário
  - /** */
- Cada entrada deve ser colocada logo antes de
  - Definição da classe
  - Campos
  - Construtores
  - Métodos
- As entradas do JavaDoc são compostas por
  - Descrição
  - Bloco de tags

unesp

JavaDoc

```
/**
 * Returns an <code>Image<code> object that can then be painted on
 * the screen. The <code>url</code> argument must specify an absolute
 * {@link URL}. The <code>name</code> argument is a specifier that
 * is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param url an absolute URL giving the base location of the image
 * @param name the location of the image, relative to the url argument
 * @return the image at the specified URL
 * @see Image
 */
public Image getImage(URL url, String name) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null;
    }
}
```

unesp

Method Detail

getImage

```
public java.awt.Image getImage(java.net.URL url,
                               java.lang.String name)
```

Returns an `Image` object that can then be painted on the screen. The `url` argument must specify an absolute `URL`. The `name` argument is a specifier that is relative to the url argument.

This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

Parameters:

`url` - an absolute `URL` giving the base location of the image

`name` - the location of the image, relative to the url argument

Returns:

the image at the specified `URL`

See Also:

`Image`

JavaDoc

- A primeira linha da entrada JavaDoc será usada na tabela que resume os métodos
- Tags de HTML podem ser usadas dentro do JavaDoc, uma vez que o texto será convertido para HTML
- A primeira linha que começar com “@” indica que o bloco de descrição terminou
- Separe a descrição das tags finais por uma linha vazia

unesp

JavaDoc

Tag & Parameter	Usage	Applies to	Since
@author *John Smith*	Describes an author.	Class, Interface, Enum	
@version *version*	Provides software version entry. Max one per Class or Interface.	Class, Interface, Enum	
@since *since-text*	Describes when this functionality has first existed.	Class, Interface, Enum, Field, Method	
@see *reference*	Provides a link to other element of documentation.	Class, Interface, Enum, Field, Method	
@param *name description*	Describes a method parameter.	Method	
@return *description*	Describes the return value.	Method	
@exception *classname description* **@throws** *classname description*	Describes an exception that may be thrown from this method.	Method	

JavaDoc

@deprecated *description*	Describes an outdated method.	Class, Interface, Enum, Field, Method	
{**@inheritDoc**}	Copies the description from the overridden method.	Overriding Method	1.4.0
{**@link** *reference*}	Link to other symbol.	Class, Interface, Enum, Field, Method	
{**@value** *#STATIC_FIELD*}	Return the value of a static field.	Static Field	1.4.0
{**@code** *literal*}	Formats literal text in the code font. It is equivalent to <code> {@literal}</code>.	Class, Interface, Enum, Field, Method	1.5.0
{**@literal** *literal*}	Denotes literal text. The enclosed text is interpreted as not containing HTML markup or nested javadoc tags.	Class, Interface, Enum, Field, Method	1.5.0

JavaDoc

```
// import statements

/**
 * This class is intended to illustrate the JavaDoc structure only.
 *
 * @author Firstname Lastname <address @ example.com>
 * @version 1.6 (current version number of program)
 * @since 1.2 (the version of the package this class was first added to)
 */
public class Test {
    /**
     * Description of the variable here.
     */
    private int debug = 0;

    // remaining class body
}
```

Classes Aninhadas

Classes Aninhadas

- Em Java, é possível declarar uma classe dentro de outra classe

```
class ClasseExterna {
    ...

    class ClasseInterna {
        ...
    }
}
```

- A classe aninhada é um membro da classe externa
  - Pode ser declarada com qualquer tipo de acesso
  - A classe aninhada tem acesso a todos os membros da classe externa, mesmo os privados

Classes Aninhadas

- Por que usar classes aninhadas?
- Maneira lógica de agrupar classes que são usadas apenas por uma outra classe
  - Classe auxiliar
- Aumenta o encapsulamento
  - As classes aninhadas tem acesso total aos membros da classe que a possui
  - Assim, não é preciso relaxar o acesso ao mundo externo
- Melhor manutenção e entendimento do código

unesp

Classes Aninhadas

- Objetos de classes aninhadas existem dentro de uma instância da classe externa
  - Lembre-se que a classe aninhada é considerada um membro da classe externa
- Para instanciar uma classe aninhada é preciso utilizar uma instância (objeto já alocado) da classe externa

```
ClasseExterna extObj = new ClasseExterna()
ClasseExterna.ClasseInterna interObj = extObj.new InnerClass();
```

unesp

Classes Aninhadas

```
public class Class1 {
    protected InnerClass1 ic;

    public Class1() {
        ic = new InnerClass1();
    }
    public void displayStrings() {
        System.out.println(ic.getString() + ".");
        System.out.println(ic.getAnotherString() + ".");
    }
    public static void main(String[] args) {
        Class1 c1 = new Class1();
        c1.displayStrings();
    }

    protected class InnerClass1 {
        public String getString() {
            return "InnerClass1: getString invoked";
        }
        public String getAnotherString() {
            return "InnerClass1: getAnotherString invoked";
        }
    }
}
```

Classes Aninhadas

- Existem outros tipos de classes aninhadas, que não trataremos aqui
  - Classes aninhadas estáticas
  - Classes locais
    - Declaradas no corpo de um método
  - Classes anônimas
    - Também declaradas no corpo de um método, mas sem definição de nome

Tipo *enum*

- Tipo especial de dado, que define os possíveis valores associados ao tipo
  - Variáveis de um tipo *enum* só pode assumir os valores pré-definidos
  - Por se tratarem de constantes, são declarados em maiúscula
  - Valores de *enum* podem ser usados em *switch*

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

Classes Aninhadas

- O tratamento de *enum* no Java é poderoso
  - O corpo de um *enum* podem conter campos e métodos
  - O compilador Java adiciona alguns métodos especiais quando um *enum* é criado
  - Ex: método estático **values()**, que retorna um array com todos os valores definidos no *enum*
  - Contudo, diferentemente de classes normais, todo enum herda a classe java.lang.Enum implicitamente

unesp

Listas

unesp

- Em Java, existem várias classes de lista que fazem parte do Java Collections Framework. Segue alguns dos tipos mais comuns de listas e suas características e vantagens:
  - ArrayList
  - LinkedList
  - Vector
  - Stack
  - ....

ArrayList

- Classe de lista baseada em matriz que implementa a interface List.
- É redimensionável, o que significa que seu tamanho pode ser alterado dinamicamente.
- É indexado e permite duplicatas e elementos nulos.

unesp

ArrayList

```java
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args) {
        // Criar um ArrayList de Strings
        ArrayList<String> myList = new ArrayList<>();

        // Adicionar elementos
        myList.add("Item1");
        myList.add("Item2");
        myList.add("Item3");

        myList.size();
        myList.remove(1);
        System.out.println(myList.get(1));

        // Iterar pelos elementos usando um loop for-each
        for (String item : myList) {
            System.out.println(item);
        }
    }
}
```

unesp

Resumo

- Estrutura das classes
  - Campos, Métodos e Construtores
- Objetos
- Passagem de parâmetros para métodos
- Modificadores de acesso
- Membros de classe (estáticos)
- Pacotes
- Javadoc
- Classes aninhadas
- Tipo Enum

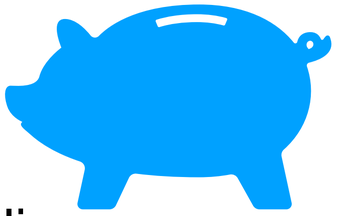
unesp


TIME COST
STRATEGY A
STRATEGY B
ANALYZING WHETHER STRATEGY A OR B IS MORE EFFICIENT
THE REASON I AM SO INEFFICIENT

Continuando com o banco UNESP

- Crie uma classe **Agencia** que tenha os seguintes campos privados:
  - Nome
  - Número da agencia (poderíamos gerar automaticamente?)
  - Número do banco
  - Contas (várias…)
- Para cada um dos campos disponibilize um método ***set*** e ***get*** ***(quando for o caso)***
- Faça um método para buscarConta (que recebe um numero de conta, o que mais?)

- Crie uma classe **Banco** que tenha os seguintes campos privados:
  - Nome
  - CNPJ (pode usar long)
  - Deposito R$ (todos valores depositados)
  - Número do banco (poderíamos para gerar automaticamente?)
  - Agencias



- Para cada um dos campos disponibilize um método ***set*** e ***get*** ***(quando for o caso)***
- Faça um método para depósito (que recebe um valor ‘só?’) e outro método para saque (que recebe valor e retorna ele ‘?')

unesp

Pratique!

- Implemente a classe Motorcycle
- A classe deve ter os campos
  - speed, gear, id, numberOfMotorcycles (estático)
  - Quando a velocidade passar de limiares determinados, gear deve ser alterada automaticamente
  - Associe um ID para cada objeto, baseado no total de objetos. Porém, não use esse valor diretamente
    - Por exemplo: numberOfMotorcycles + 1000
- Implemente os métodos
  - getGear, gearUp, getSpeed, speedUp, applyBreaks, getID
  - Pense no tipo de acesso que cada um deve ter
- A classe Motorcycle deve ter pelo menos dois construtores: o default (sem parametros) e um que recebe *speed* e *gear*

unesp

- Crie uma classe que simule um campeonato de motocicletas
- Entradas formatadas do usuário devem identificar os eventos do campeonato a cada instante
- Por exemplo
  - 1001 +2 1003 -3
    - Moto 1001 aumentou a velocidade em duas unidades
    - Moto 1003 diminuiu a velocidade em 3 unidades
  - Após cada entrada do usuário, atualize os objetos e imprima na tela o estado de todos os objetos
    - Moto 1001: Velocidade=23, Marcha=3.
    - Moto 1002: Velocidade=10, Marcha=2.
    - ...

- Se você quiser, pode dar mais emoção
  - Baseado na velocidade de cada moto, calcule a distância percorrida por cada uma entre um evento e outro (entrada de dados)
    - Assuma que entre uma entrada e outra há X segundos
  - Neste caso, a posição de cada motocicleta precisa ser controlada e atualizada
  - Após cada evento (entrada de dados), a distância percorrida e a posição atual de cada motocicleta também deve ser informada
    - Moto 1001: Velocidade=23, Marcha=3, Distancia=240m, Posicao=2.
    - Moto 1002: Velocidade=10, Marcha=2, Distancia=190m, Posicao=5.
    - ...

Bibliografia

- DEITEL, H. M. & DEITEL, P.J. "Java : como programar", Bookman, 2017.

- Agradecimentos. Material baseado nos slides:
  - Luiz E. Virgilio da Silva. Notas de Aula de Programação Orientada a Objetos. Classes e Objetos (ICMC/USP).