Introdução à Orientação a Objetos

5185/31 e 6888/1 – Paradigma de Programação Imperativa
e Orientada a Objetos

Profa. Valéria D. Feltrim

UEM – CTC – DIN

1ª AULA PRÁTICA

Conceitos básicos

Introdução

- Um programa O.O. é estruturado de forma a representar os objetos e as suas relações no mundo real
  - O modelo imperativo se baseia na forma como a máquina processa instruções
    - Resolver um problema é estabelecer uma sequência de passos a ser executada
  - O modelo orientado a objetos busca abstrair a máquina e focar nos objetos do domínio e nas suas relações
    - Resolver um problema é estabelecer uma coleção de objetos que se comunicam para alcançar um objetivo
- Como todo paradigma de programação, a grande "sacada" está na **modelagem do problema**
  - A LP apenas fornece os recursos necessários para que tal modelagem possa ser implementada
  - Algumas LPs são puramente O.O. (*Smalltalk*), enquanto outras incluem suporte à programação O.O. (*C++*) → discutiremos isso mais tarde!
- Por enquanto, faremos uma introdução a programação O.O. e a LP **Java**

Introdução ao Java

- Um arquivo de código Java tem extensão **.java** e contém **uma ou mais classes, mas apenas uma classe pública**
  - O nome do arquivo .java deve ser o mesmo usado para nomear a classe pública contida no arquivo
  - Quando compilado, se tudo correr bem, um ou mais arquivos .class serão gerados (um para cada classe do .java)
    - Os arquivos .class é que serão usados pela JVM para execução
  - Quando um arquivo .class é passado para a JVM, ela busca pelo método ***main*** da classe para iniciar a execução
    - Nem toda classe terá um método *main*, mas toda aplicação terá pelo menos uma classe com esse método

Ver linha de comando: javac e java

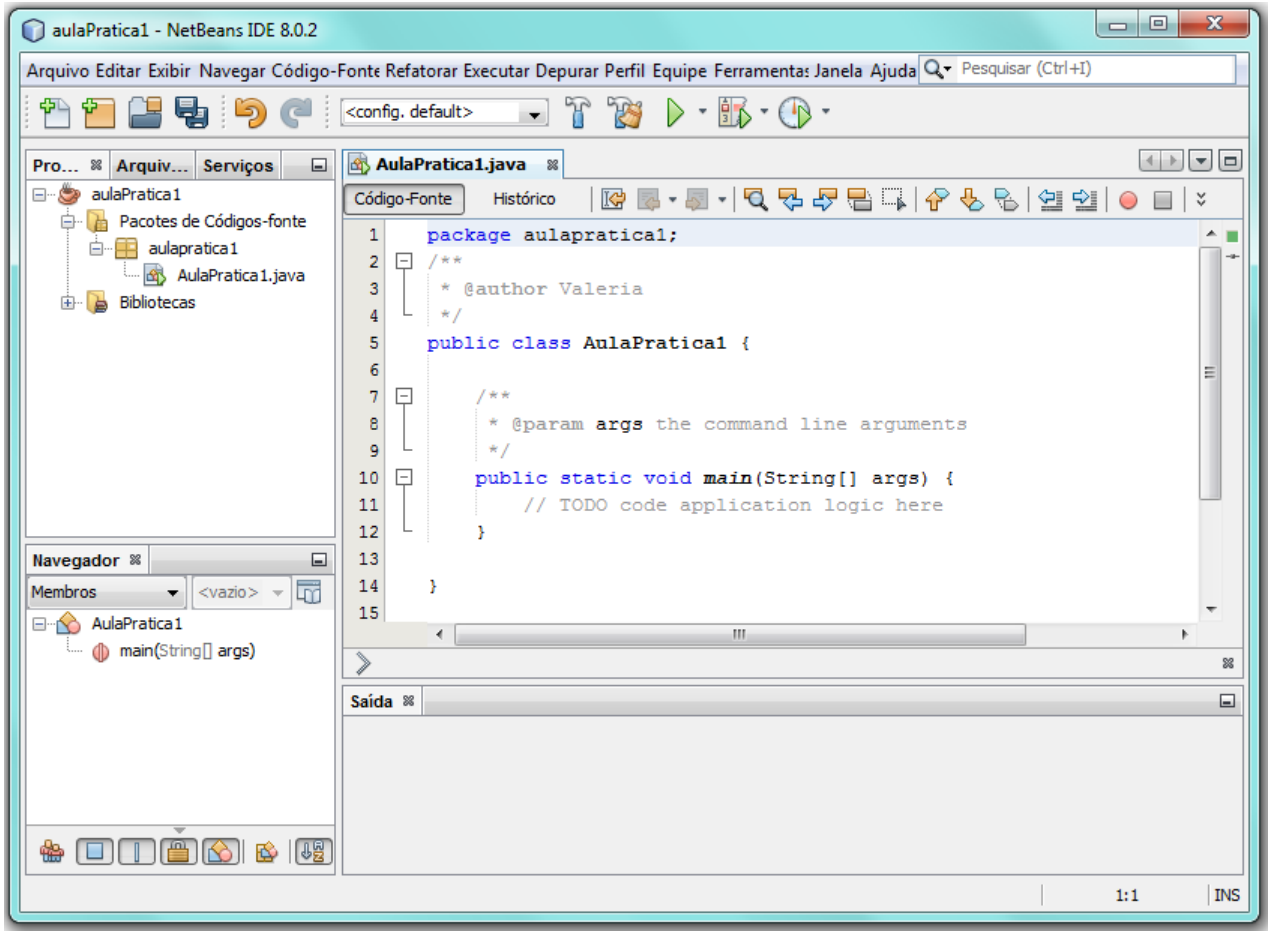
Teste.java

Todo método *main* tem exatamente este cabeçalho

```
public class Teste{
    public static void main(String[] args) {
        System.out.println("Welcome to Java Programming!");
    }
}
```

Introdução ao Java

- Programação utilizando uma IDE ajuda bastante
  - Netbeans, Eclipse, etc.
- Cada nova aplicação corresponderá a um projeto
  - Terá uma classe principal → *main*()
  - Poderá ter tantas classes quanto forem necessárias

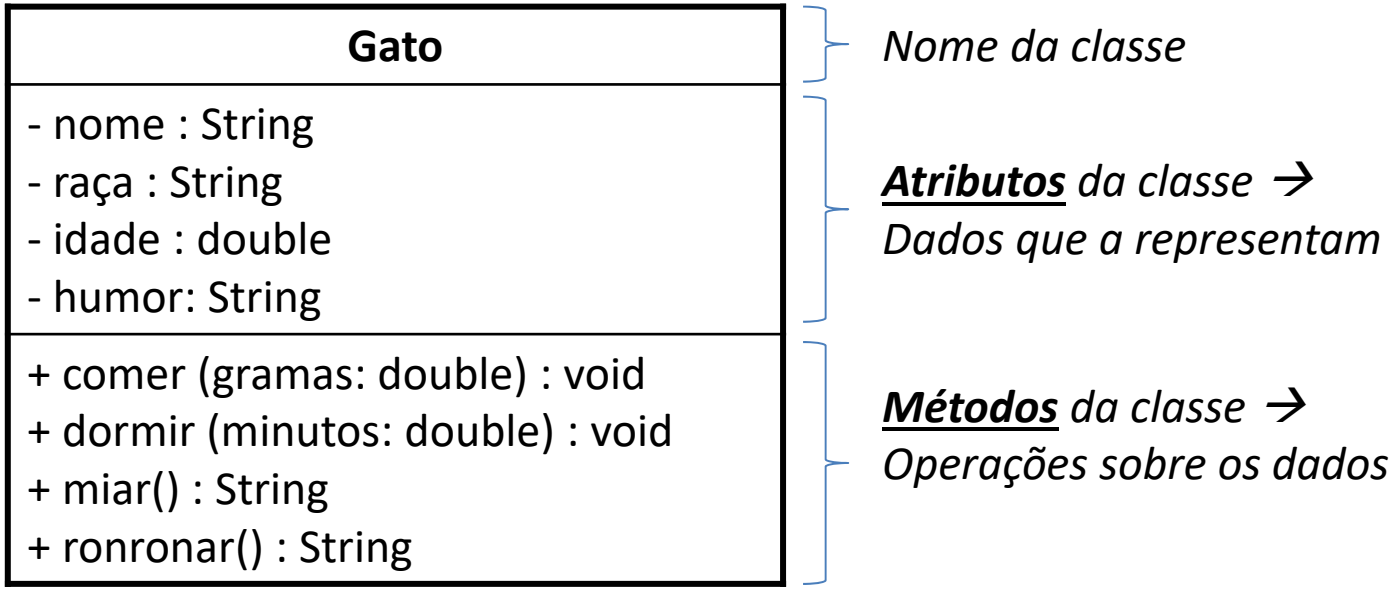


Classes e objetos

- Um **objeto** é uma abstração de alguma entidade e tem estado e comportamento
  - **O estado** do objeto é definido pelos valores dos seu **atributos**
  - **O comportamento** é definido pelos seus **métodos**
- **Objetos** são criados instanciando-se alguma classe
- **Classes** são como "formas" ou "modelos" para a criação de objetos
  - Convenção Java → nomes de classes começam com letra maiúscula
  - Elas definem os atributos e os métodos que os objetos da classe terão → **definem um novo tipo** (como um TAD)
  - Em Java, todo objeto é criado por meio do operador ***new*** seguido do nome da classe
    - O resultado de um *new* sempre é uma referência para o objeto criado
    - Lembre-se que em Java todos os objetos são dinâmicos no *heap*

Classes

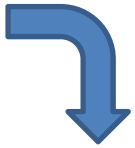
- Exemplo: Representação da classe **Gato**



Classes

- Exemplo:

Gato
- nome : String - raça : String - idade : double - estado: String
+ comer (gramas: double) : void + dormir (minutos: double) : void + miar() : String + ronronar() : String

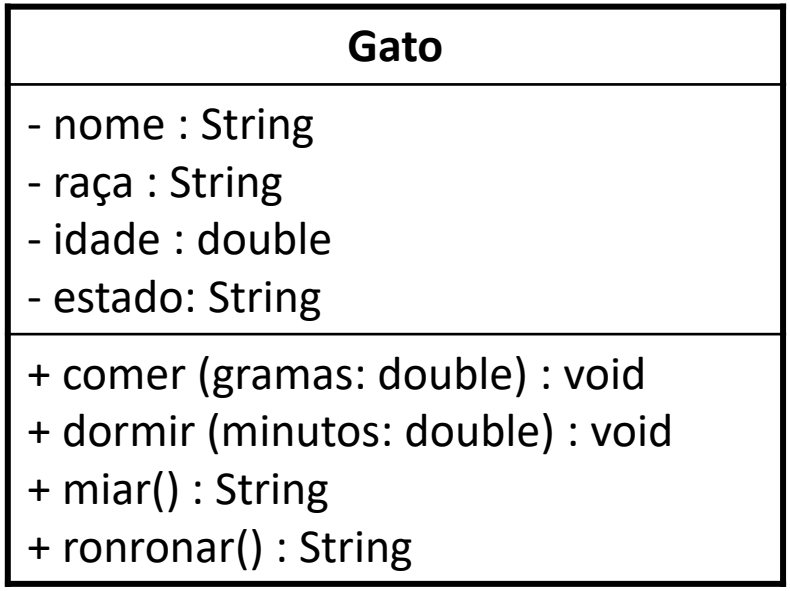


```
public class Gato {

    private String nome;
    private String raça;
    private double idade;
    private String estado;

    public void comer(double g){}

    public void dormir(double m){}

    public String miar(){}

    public String ronronar(){}

}
```

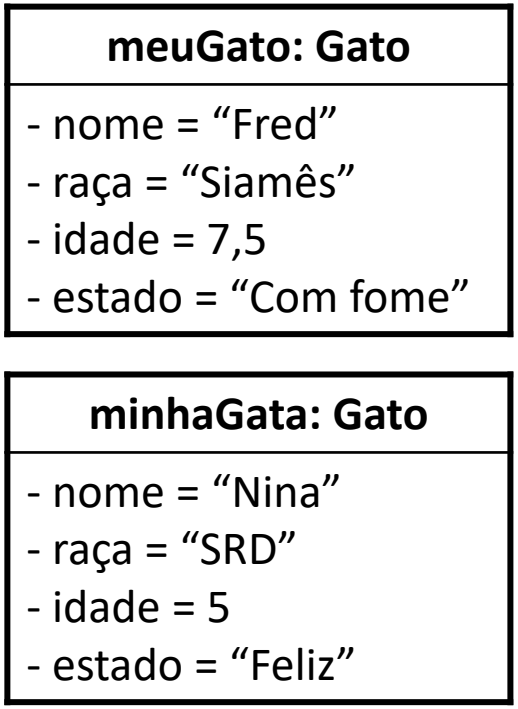
Classes

- Exemplo:

Classe Gato:



Objetos Gato:



Classes

```
public class Gato {

   private String nome;
   private String raça;
   private double idade;
   private String estado;

   public void comer(double g){}

   public void dormir(double min){}

   public String miar(){}

   public String ronronar(

}
```

```
public class ControleGatos{
   public static void main(String[] args) {
      Gato meuGato = new Gato();
      Gato minhaGata = new Gato();
      ...
      meuGato.comer(20);
      minhaGata.dormir(30);
      ...
      System.out.println(meuGato.miar());
      ...
   }
}
```

Controle de acesso Java

- Os **modificadores de acesso** se aplicam em todos os níveis
  - Classes, atributos e métodos
- Tudo o que é privado (**`private`**) é de acesso exclusivo da classe
  - Geralmente, os atributos são privados de modo que apenas os métodos da classe podem acessá-los
- Apenas o que é público (**`public`**) pode ser acessado fora da classe
  - O que é público compõem a interface (ou protocolo) da classe
  - Apenas métodos públicos podem ser invocados pelos objetos da classe
- Tudo o que for protegido (**`protected`**) é de acesso exclusivo da classe e de suas subclasses

Métodos *getter* e *setter*

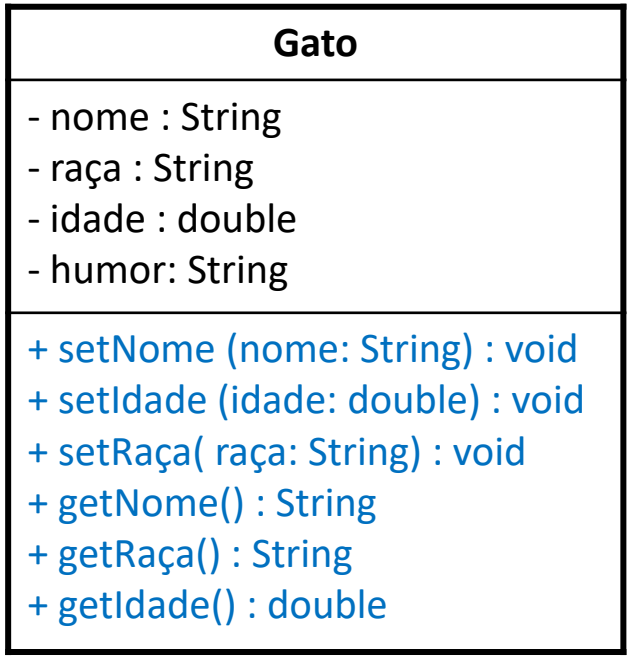
- Se os atributos são privados, precisamos de métodos públicos para acessá-los e modificá-los
  - Se isso for permitido na modelagem do problema
- Os métodos de acesso são chamados de ***getters*** e ***setters*** e servem para recuperar (*get*) e atribuir (*set*) valores aos dados privados de maneira confiável

```
public class ControleGatos{
   public static void main(String[] args) {
      Gato meuGato = new Gato();
      Gato minhaGata = new Gato();
      ...
      meuGato.nome = “Fred”;
      ...
      System.out.println(meuGato.idade);
   }
}
```

Métodos *getter* e *setter*

- Se os atributos são privados, precisamos de métodos públicos para acessá-los e modificá-los
  - Se isso for permitido na modelagem do problema
- Os métodos de acesso são chamados de ***getters*** e ***setters*** e servem para recuperar (*get*) e atribuir (*set*) valores aos dados privados de maneira confiável

```
public class ControleGatos{
   public static void main(String[] args) {
      Gato meuGato = new Gato();
      Gato minhaGata = new Gato();
      ...
      meuGato.setNome("Fred");
      ...
      System.out.println(meuGato.getIdade());
   }
}
```



Classes

```
public class Gato {
   private String nome;
   private String raça;
   private double idade;
   private String estado;

   public void setNome(String nome){
     this.nome = nome;
   }

   public String getNome(){
     return this.nome;
   }

   public void setIdade(double idade){
     if ((idade < 0) || (idade > 20)) {
       System.out.println(idade+" não é uma idade válida para um gato.");
       this.idade = -1;
     }
     else {
       this.idade = idade;
     }
   }
   ...
}
```

Classes

```
public class Gato {
   private String nome;
   private String raça;
   private double idade;
   private String humor;

   public void setNome(String nome){
     this.nome = nome;
   }

   public String getN
     return this.nome
   }

   public void setIda
     if ((idade < 0)
       System.out.pri
       this.idade = -
     }
     else {
       this.idade = i
     }
   }
   ...
}
```

```
public class ControleGatos{
   public static void main(String[] args) {
      Gato meuGato = new Gato();
      meuGato.setNome("Fred");
      meuGato.setRaça("Siames");
      meuGato.setidade(7,5);

      System.out.println(meuGato.getNome());
      ...
   }
}
```

Construtores

- Os **construtores** servem para inicializar os objetos de uma classe
- A classe pode ter **um ou mais construtores**, desde que o protocolo de cada construtor seja diferente (**construtores sobrecarregados**)
- Construtores sempre têm o **mesmo nome da classe** e não retornam nenhum valor (nem mesmo **`void`**)
- Em Java, se nenhum construtor for especificado para a classe, um **construtor padrão** será aplicado
  - Inicializa variáveis numéricas primitivas com 0, booleanas com **`false`** e referências com **`null`**
  - Se um construtor for especificado, o construtor padrão nunca será chamado

Construtores

```
public class Gato {

   private String nome;
   private String raça;
   private double idade;
   private String estado;

   Gato(String n, String r, double i) {
      this.nome = n;
      this.raça = r;
      setIdade(i);
      this.estado = “Feliz”;
   }

   ...
}
```

Construtores

```
public class Gato {

   private String nome;
   private String raça;
   private double idade;
   private String estado;

   Gato(String n, String r, double i) {
      this.nome = n;
      this.raça = r;
      setIdade(i);
      this.estado = “Feliz”;
   }

   Gato(){
      this(null, null, 0);
   }
   ...
}
```

Construtores

```
public class Gato {

   private String nome;
   private String raça;
   private double idade;
   private String estado;

   Gato(String n, String r, double i) {
      this.nome = n;
      this.raç
      setIdade
      this.est
   }

   Gato(){
      this(nul
   }
   ...
}
```

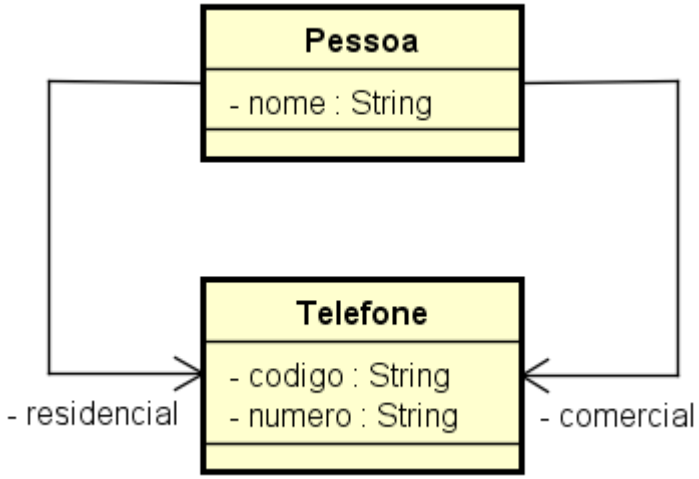
```
public class ControleGatos{
   public static void main(String[] args) {
      Gato meuGato = new Gato();
      meuGato.setNome("Fred");
      meuGato.setRaça("Siames");
      meuGato.setidade(7,5);

      Gato minhaGata = new Gato("Nina", "SRD", 5);
      ...
   }
}
```

2ª AULA PRÁTICA

Associação entre objetos

Associação

- As **associações** definem as relações entre os objetos das diferentes classes
  - **Relações TEM-UM (*has-a*)** → são implementadas por **associações**
  - Relações do tipo É-UM (*is-a*) são implementadas por meio de herança



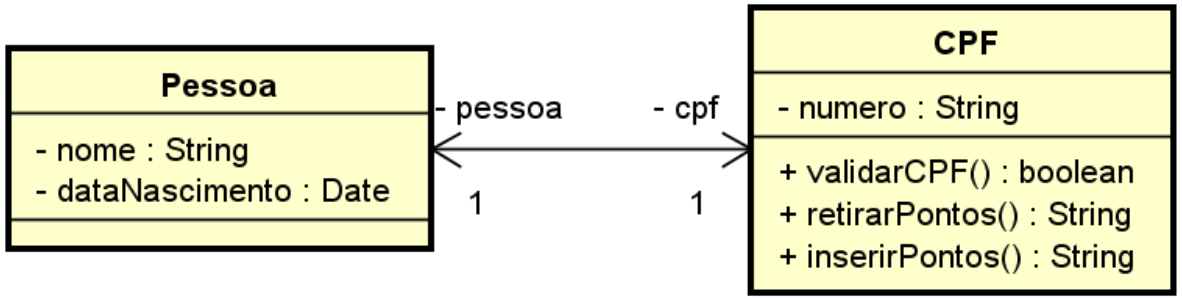

```
public class Pessoa{
    private String nome;
    private Telefone residencial;
    private Telefone comercial;
}
```

```
public class Telefone{
    private String codigo;
    private String numero;
}
```

Veja que a **associação** entre Pessoa e Telefone implica na existência de uma referência a um objeto Telefone na classe Pessoa.

Associação

- Associação **um-para-um** bidirecional

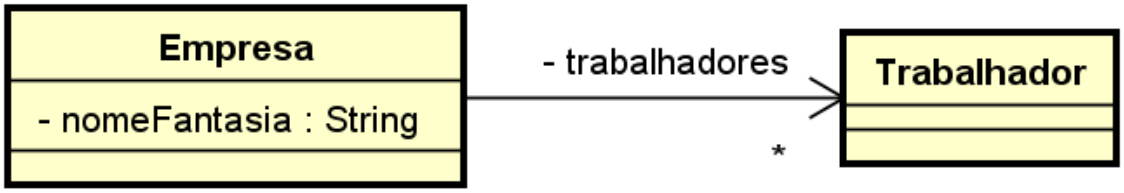



```
public class Pessoa {
   private String nome;
   private Date dataNascimento;
   private Cpf cpf;
}
```

```
public class Cpf {
   private String numero;
   private Pessoa pessoa;
}
```

Associação

- Associação **um-para-muitos** unidirecional




```
public class Empresa {

   private String nomeFantasia;
   private List<Trabalhador> trabalhadores;
}
```

Coleções Java

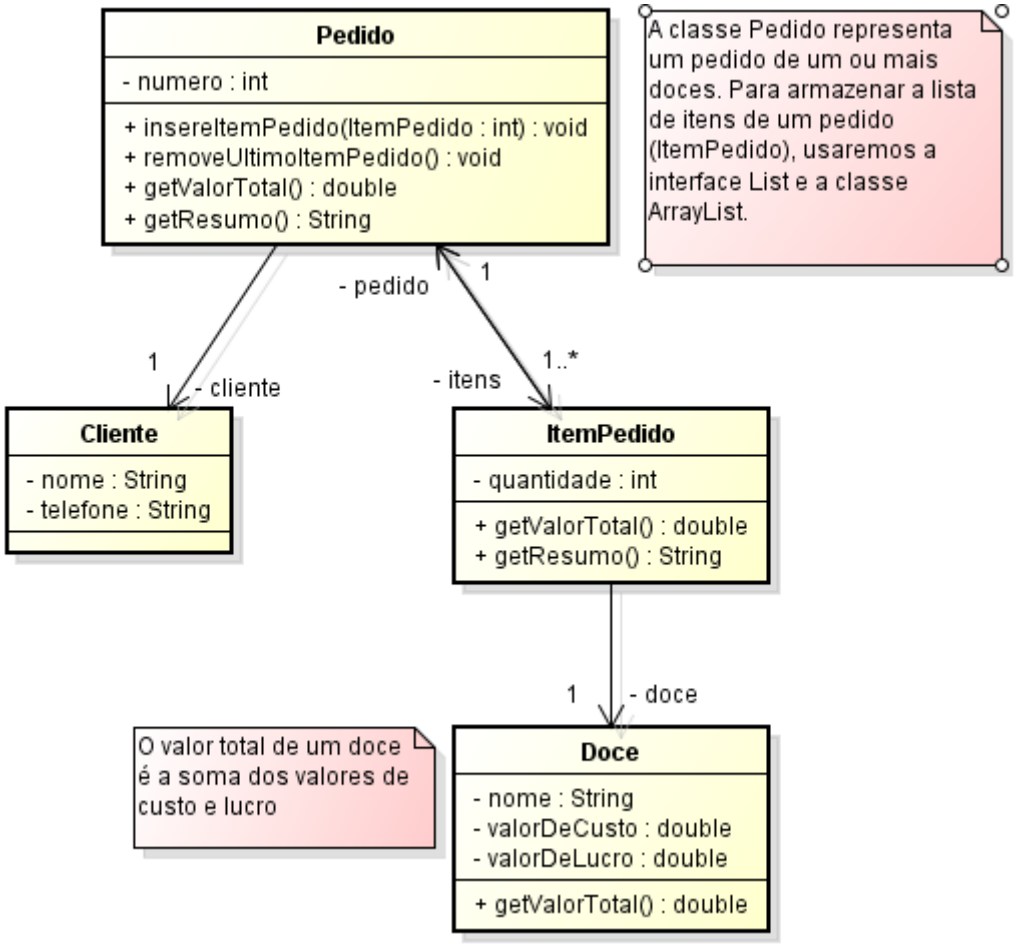
- Java possui um conjunto de classes e interfaces conhecido como *Collections Framework*, que reside no pacote **java.util**
- Implementa vários tipos de estruturas de dados
  - Listas (*List*): *ArrayList*, *LinkedList*
  - Conjuntos (*Set*): *HashSet*, *LinkedHashSet*, *TreeSet*
  - Mapas (*Maps*): *HashMap*, *TreeMap*, *Hashtable*
- As estruturas de dados são genéricas e têm seu tipo instanciado na declaração
  - Por ex., a interface *List* e a classe *ArrayList* servem tanto para declarar uma lista de Strings quanto uma lista de Gatos

```
public class ControleGatos{
   public static void main(String[] args) {
      List<Gato> meusGatos = new ArrayList<>();
      List<String> nomes = new ArrayList<>();
   }
}
```

ArrayList

- Alguns métodos da classe ***ArrayList***
  - boolean *add* (Object element): Adiciona o elemento especificado no final da lista
  - Object *get* (int index): Retorna o i-ésimo elemento da lista
  - int *size*(): Retorna o número de elementos da lista
  - boolean *isEmpty* (): Retorna verdadeiro se a lista estiver vazia e falso caso contrário
  - Object *remove* (int index): Remove o i-ésimo elemento da lista.
  - int *indexOf* (Object element): Retorna a posição da primeira ocorrência do elemento especificado na lista
  - void *clear* (): Remove todos os elementos da lista
  - boolean *contains* (Object element): Retorna verdadeiro se a lista contém o elemento especificado e falso caso contrário

Exercício


Pedido
- numero : int
+ insereItemPedido(ItemPedido : int) : void
+ removeUltimoItemPedido() : void
+ getValorTotal() : double
+ getResumo() : String
A classe Pedido representa um pedido de um ou mais doces. Para armazenar a lista de itens de um pedido (ItemPedido), usaremos a interface List e a classe ArrayList.
- pedido
1
1
- cliente
1..*
- itens
Cliente
- nome : String
- telefone : String
ItemPedido
- quantidade : int
+ getValorTotal() : double
+ getResumo() : String
1
- doce
O valor total de um doce é a soma dos valores de custo e lucro
Doce
- nome : String
- valorDeCusto : double
- valorDeLucro : double
+ getValorTotal() : double

3ª AULA PRÁTICA

Membros estáticos
Enumerações

Membros estáticos

- Geralmente, os **atributos** e **métodos** pertencem ao objeto
  - Os atributos são chamados **variáveis de instância**
  - Cada objeto tem sua cópia das variáveis de instância
  - Nesse caso, as variáveis e os métodos só podem ser acessados por meio de um objeto da classe (a classe precisa ser instanciada)
- Opcionalmente, atributos e métodos podem pertencer a classe em vez de a um objeto
  - Chamadas **variáveis de classe ou membros estáticos**
  - Nesse caso, todos os objetos compartilharão a mesma cópia da variável estática
  - As variáveis e métodos estáticos podem ser acessados sem que um objeto da classe precise ser criado (eles existem mesmo que a classe não tenha sido instanciada)
  - Membros estáticos são criados anexando-se o modificador **`static`** à declaração

Ver arquivos de código Employee.java e EmployeeTest.java

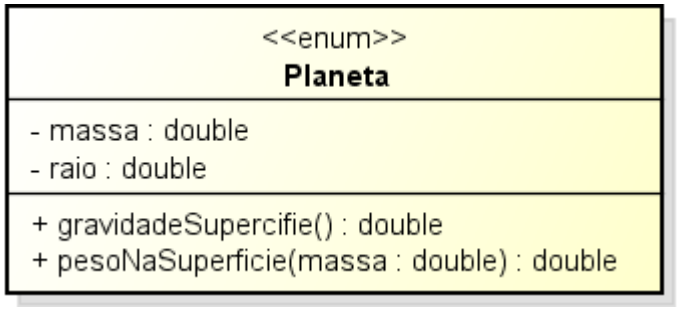
Enumeração

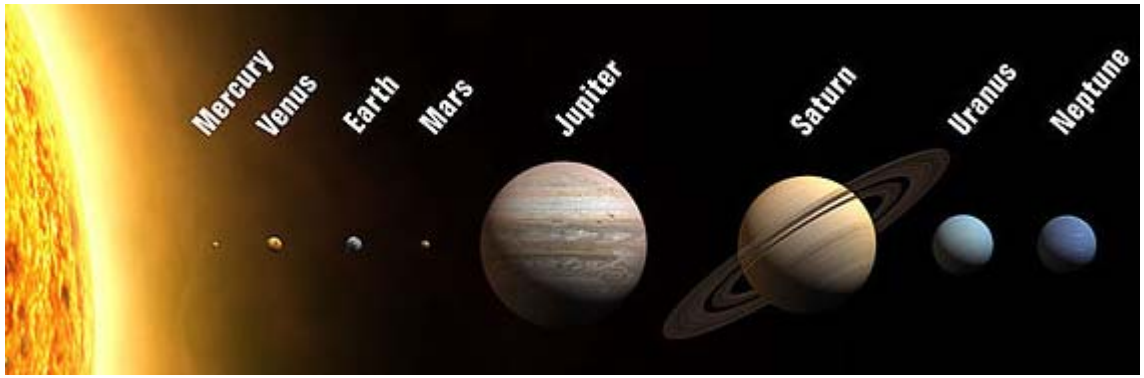
- Já vimos em aulas anteriores que um **tipo enumeração** define uma lista de constantes nomeadas
- Em Java, um tipo enumeração é declarado com uma declaração **`enum`**
  - Pode ser só uma lista de constantes ou ser tão complexa quanto uma classe
  - Pense na enumeração como uma classe em que todos os objetos possíveis são conhecidos
- Cada declaração **`enum`** declara uma **classe `enum`** com as seguintes restrições
  - Constantes **`enum`** são implicitamente **`final`** → não podem ser modificadas
  - Constantes **`enum`** são implicitamente **`static`**
  - **Uma classe `enum` não pode ser instanciada**
  - O construtor de uma **`enum`** não pode ser público
- Toda **`enum`** estende implicitamente a classe **`java.lang.Enum`**
  - Herda métodos como values()
- Por convenção, os nomes das constantes **`enum`** são escritos em letras maiúsculas

Ver arquivo de código TestWeekDay.java

Enumeração

- **Exemplo**: Imagine uma classe Planeta que descreve os planetas do sistema solar em termos de suas massas e raios
  - Só poderão existir 8 objetos dessa classe
  - Sempre que todos os objetos de uma classe são conhecidos, podemos modelar a classe como uma enumeração → classe **`enum`**





Planeta	Massa (Kg)	Raio (m)
Mercúrio	3,303e+23	2,4397e6
Vênus	4,869e+24	6,0518e6
Terra	5,976e+24	6,37814e6
Marte	6,421e+23	3,3972e6
Júpiter	1,9e+27	7,1492e7
Saturno	5,688e+26	6,0268e7
Urano	8,686e+25	2,5559e7
Netuno	1,024e+26	2,4746e7

Ver arquivos de código Planeta.java e TestPlaneta.java

Exercício

- Faça um programa que exibe um cardápio de uma lanchonete, com três tipos de comida e três tipos de bebida. A partir do cardápio o cliente escolhe o que quer comer e/ou beber e, a cada escolha, o programa retorna o valor da conta
- Para isso crie duas enumerações: **'Bebida'** e **'Comida'**
  - Cada enumeração possui três atributos privados: **nome** (*String*), **preçoDeCusto** (*double*) e **preçoDeLucro** (*double*)
  - Além disso, as enumerações terão um método construtor e métodos públicos que retornam o nome (**getNome**()) e o preço final (**getPrecoFinal**()) de cada produto
- Na classe principal:
  - Além do método **main**(), crie um método **menu()**, que exibe na tela as bebidas e comidas disponíveis no cardápio com os respectivos preços
  - Crie também um método **preco**(), que recebe como parâmetro o número correspondente ao item do cardápio escolhido pelo cliente e retorna o preço desse item.
  - No método **main**(), crie um objeto **entrada** da classe **Scanner** e o use para ler a **opção** do cliente por meio do método **nextInt**()
  - A opção escolhida pelo cliente será passada ao método **preco**(), que retornará o valor do item pedido, de modo que podemos acumular o valor do item ao **valor da conta**, que deve ser mostrado sempre que o cliente faz um pedido
  - Quando o cliente terminar de pedir, ele deve digitar '0' e programa encerrará apresentando o valor final da conta

**Exercício extraído de*
http://www.javaprogressivo.net/2012/10/Como-usar-enumA-melhor-maneira-para-manusear-constantes-em-Java.html

4ª AULA PRÁTICA

- Escreva um programa completo para jogar o jogo da velha. Para isso crie uma classe **JogoDaVelha**:
  - A classe deve conter como dados privados um array bidimensional 3 x 3 para representar a grade do jogo
  - Crie uma enumeração Celula para representar as possibilidades de ocupação de uma célula na grade (vazia, jogador 1 ou jogador 2)
  - O construtor deve inicializar a grade como vazia
  - Forneça um método para exibir a grade (imprimir())
  - Permita dois jogadores humanos
  - Crie um método privado que verifica se houve uma vitória ou empate
  - Forneça um método para jogar o jogo (jogar()):
    - Toda jogada deve ocorrer em uma célula vazia
    - Depois de cada jogada, determine se houve uma vitória ou um empate

**Exercício proposto pelo Prof. José Romildo Malaquias (UFOP). Disponível em http://www.decom.ufop.br/romildo/bcc221.2011-1/poo-1-classes.pdf*