

# Linguagem de Programação Java

Universidade Federal do Pará

Instituto de Ciências Exatas e Naturais

Programa de Pós-Graduação em Ciência da Computação

Estágio Docência

Professor Dr. Claudomiro Sales

Discente Gustavo Lobato

Contato: gustavomaues (gtalk, skype, @ufpa.br)

Belém - 2016

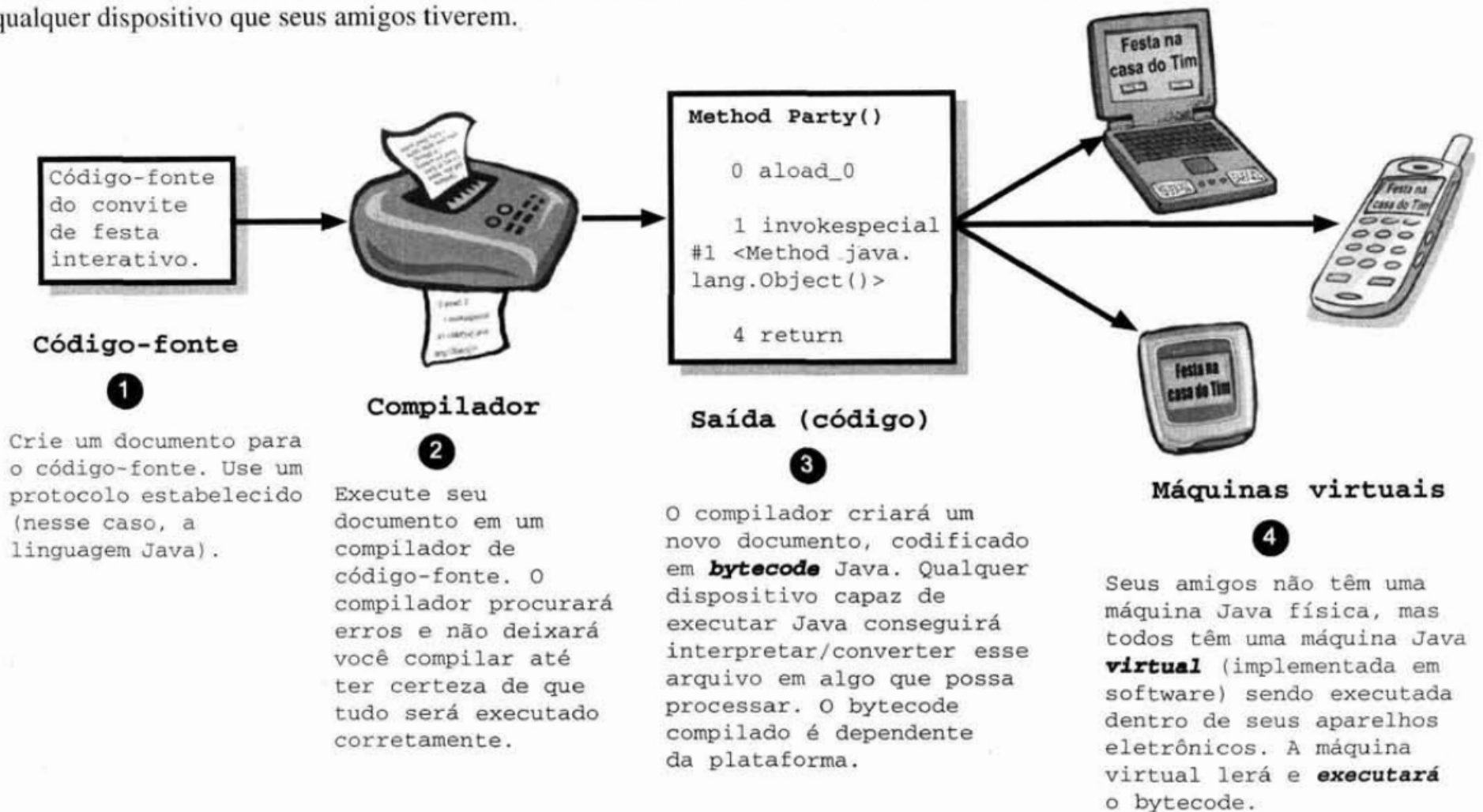
# Introdução a Linguagem Java

- Como qualquer linguagem de programação, a linguagem Java tem sua própria estrutura, regras de sintaxe e paradigma de programação. O paradigma de programação da linguagem Java baseia-se no conceito de OOP, que os recursos da linguagem suportam.
- A linguagem Java deriva da linguagem C, portanto suas regras de sintaxe assemelham-se às regras de C. Por exemplo, os blocos de códigos são modularizados em métodos e delimitados por chaves ({ e }) e variáveis são declaradas antes que sejam usadas.
- Estruturalmente, a linguagem Java começa com pacotes. Um pacote é o mecanismo de namespace da linguagem Java. Dentro dos pacotes estão as classes e dentro das classes estão métodos, variáveis, constantes e mais.

# Como o Java funciona

## Como o Java funciona

O objetivo é escrever um aplicativo (neste exemplo, um convite de festa interativo) e fazê-lo funcionar em qualquer dispositivo que seus amigos tiverem.



# Como o Java funciona

## O que você fará em Java

Você criará um arquivo de código-fonte, compilará usando o compilador javac e, em seguida, executará o bytecode compilado em uma máquina virtual Java.

```
import java.awt.*;
import java.awt.event.*;
class Party {
    public void buildInvite() {
        Frame f = new Frame();
        Label l = new
Label("Party at Tim's");
        Button b = new
Button("You bet");
        Button c = new
Button("Shoot me");
        Panel p = new Panel();
        p.add(l);
    } // mais código aqui...
}
```

### Código-fonte

1

Digite seu código-fonte.  
Salve como: **Party.java**

```
File Edit Window Help Plead
%javac Party.java
```

### Compilador

2

Compile o arquivo **Party.java** executando o javac (o aplicativo do compilador). Se não houver erros, você terá um segundo documento chamado **Party.class**. O arquivo **Party.class** gerado pelo compilador é composto de **bytecodes**.

```
Method Party()
  0 aload_0
  1 invokespecial #1 <Method
java.lang.Object()>
  4 return
Method void buildInvite()
  0 new #2 <Class java.awt.Frame>
  3 dup
  4 invokespecial #3 <Method
java.awt.Frame()>
```

### Saída (código)

3

Código compilado:  
**Party.class**

```
File Edit Window Help Swear
%java Party
Party at Tim's!
You bet Shoot Me
```

### Máquinas virtuais

4

Execute o programa iniciando a Java Virtual Machine (JVM) com o arquivo **Party.class**. A JVM converterá o **bytecode** em algo que a plataforma subjacente entenda e executará seu programa.

# 1. Declarações e Controle de Acesso

## Objetivos

Declarar Classes e Interfaces

Desenvolver Interfaces e Classes *Abstract*

Usar Primitivos, *Enums* e Identificadores Legais

Usar Métodos *Static*, Nomeação *JavaBeans* e *Var-Args*

# Declarações e Controle de Acesso

```
package br.ufpa.sigaa;
```

```
import br.ufpa.sipac.*;
```

```
/*
```

```
* Mostrando o esqueleto de uma classe
```

```
*/
```

```
[default,public] [final,abstract,strictfp] class NomeClasse extends NomeClasseMae implements NomeInterface {
```

```
//Constantes
```

```
[private, protected, public] final static tipo NOME_DA_CONSTANTE = Valor da Constante;
```

```
//Variáveis de instância
```

```
[private, protected, public] [final,transient,volatile] tipo nomeVariavelDeInstancia;
```

```
//Construtores
```

```
[private, protected, public] NomeClasse([argumentos]){ }
```

```
//Métodos
```

```
[private, protected, public] [final,abstract,synchronized,native,strictfp] tipoRetorno nomeMetodo([argumentos]){
```

```
    [final] tipo variavelLocal = valor;
```

```
}
```

```
• }
```

# 1.2 Identificadores, Convenções de Código e JavaBeans

- **Identificadores Legais**

*regras que o compilador usa para determinar se um dado nome é legal;*

- **Convenções de Código Java (Sun, 1999)**

*padrões de codificação que facilitam os testes, manutenção e evolução de códigos;*

- **Padrões de Nomeação Javabens**

*convenções básicas de nomeação;*

## 1.2 Identificadores, Convenções de Código e JavaBeans

### 1.2.1 Identificadores Legais

- Caractere inicial válido: a-Z, \$, \_
- Caracteres seguintes válidos: caracteres unicode (letras e dígitos), números, símbolos de moedas e caracteres de conexão (*underscores*)
- Os identificadores em Java são *case-sensitive*;
- Não se pode usar uma palavra-chave como identificador



# 1.2 Identificadores, Convenções de Código e JavaBeans

## 1.2.1 Identificadores Legais

```
1
2 public class IdentificadoresLegais {
3
4     //Início Válido
5     int letra, let1, l_etra, l$etra; //pode iniciar com qualquer letra
6     int $cifrao, $$$$; //pode iniciar com cifrão
7     int _underscore, _underscore$, _l$$$underscore, ____; //pode iniciar com underscore
8     int nao_ha_limite_de_caracteres_para_se_estabelecer_o_nome_de_um_identificador_em_java;
9
10    //Início inválido: qualquer outro caso;
11    int -traco, .ponto, lnumero, #jogodavelha;
12
13    //Case sensitive
14    int idade, Idade; //variáveis diferentes
15    int nome, nome; //variáveis idênticas
16
17    //palavras-chave não podem ser usadas como identificadores
18    int class = 0;
19    int boolean = 0;
20 }
```

## 1.2 Identificadores, Convenções de Código e JavaBeans

### 1.2.2 Convenções de Código

- <http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html>

- **Classes e interfaces**

primeira letra de cada palavra maiúscula. Ex: Pessoa, PessoaFisica, Auditavel, etc;

- **Métodos**

primeira letra minúscula, palavras seguintes iniciadas com letra maiúscula. Ex.: getPessoa, listarVendedoresExternos;

- **Variáveis**

mesma regra dos métodos: Ex.: saldo, alturaMaxima, valorVenda;

- **Constantes**

somente letras maiúsculas com palavras separadas por underscore. Ex.: MENSAGEM\_ERRO, LIMITE\_ARQUIVO, TAMANHO\_MAXIMO;

# 1.2 Identificadores, Convenções de Código e JavaBeans

## 1.2.3 Padrões JavaBeans

- **O que são JavaBeans**

Uma classe Java que expõe propriedades, seguindo uma convenção de nomenclatura simples para os métodos getter e setter. O JavaBean é um Objeto Java que é serializável, possui um construtor sem argumentos e permite acesso às suas propriedades através de métodos getter e setter.

```
public class JavaBeans implements java.io.Serializable {

    public JavaBeans() {
        //Construtor sem argumentos
    }

    /*
     * Abaixo, temos as propriedades nome e vaiAprenderJava
     * observem que são variáveis private, portanto, só poderão ser acessadas ou modificadas
     * externamente através dos métodos declarados a seguir.
     */
    private String nome;

    private boolean vaiAprenderJava = true;

    /*
     * Observe abaixo os métodos públicos
     * setter: modificam o valor de uma propriedade (→ sem retorno, com argumentos)
     * getter: retornam o valor de uma propriedade (→ com retorno, sem argumentos)
     */

    public String getNome() {
        return nome;
    }

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

    public void setVaiAprenderJava(boolean vaiAprenderJava) {
        this.vaiAprenderJava = vaiAprenderJava;
    }

    //o método getter de uma propriedade boolean pode começar com is ou get
    public boolean isVaiAprenderJava() {
        return vaiAprenderJava;
    }

}
```

## 1.3 Declaração de Classes

### 1.3.1 Regras de Declaração Para Arquivos

- Em cada arquivo só pode existir uma classe *public*; é possível ter mais de uma classe *não-public*;
- Se um arquivo tem uma classe *public*, o nome do arquivo deve ter o mesmo nome da classe, com a extensão *.java*; se não houver classe *public*, pode ter qualquer nome;
- Se a classe fizer parte de um pacote, a declaração *package* deve a primeira linha, antes de qualquer declaração;
- Se houver declarações *import*, elas devem ficar entre a declaração *package*, se houver, e a declaração da classe;
- As declarações *package* e *import* aplicam-se a todas as classes dentro de um arquivo;

# 1.3 Declaração de Classes

## 1.3.2 Declarações e Modificadores de Classes

### Modificadores de Acesso

- Declaração mais simples: **class MinhaClasse { }**
- Uma classe **A** acessa uma classe **B** quando:
  - cria uma instância de B;
  - estende/herda/torna-se subclasse de B;
  - acessa métodos e variáveis dentro da classe B;
- Acesso (Visibilidade): Modificadores vs Controladores
- Controladores: *default*, *public*, ~~*private*~~, ~~*protected*~~
- Modificadores: *public*, ~~*private*~~, ~~*protected*~~

# 1.3 Declaração de Classes

## 1.3.2 Declarações e Modificadores de Classes

### Modificadores de Acesso

MODIFICADOR	CLASSE	MESMO PACOTE	PACOTE DIFERENTE (SUBCLASSE)	PACOTE DIFERENTE(GLOBAL)
Public				
Protected				
Default				
Private				

# 1.3 Declaração de Classes

## 1.3.2 Declarações e Modificadores de Classes

### Acesso *default* vs *public*

- **Default**: somente as classes do mesmo pacote podem acessar (não necessita importar)
- **Public**: todas as classes podem acessar (necessita importar, se em pacote diferente)

The screenshot displays an IDE with a Package Explorer on the left and five code editors showing Java files. The Package Explorer shows a project named 'Aula1' with a 'src' folder containing a package 'acesso'. Inside 'acesso', there are three sub-packages: 'pacoteA', 'pacoteB', and 'pacoteC'. Each package contains a class file: 'ClasseA1.java' in 'pacoteA', 'ClasseB1.java' in 'pacoteB', and 'ClasseC1.java' in 'pacoteC'. The code editors show the following code:

```
ClasseA1.java
1 package acesso.pacoteA;
2
3 import acesso.pacoteB.ClasseB1;
4 import acesso.pacoteC.ClasseC1;
5
6 class ClasseA1 {
7     ClasseA1 classeA1;
8     ClasseA2 classeA2;
9     ClasseB1 classeB1;
10    ClasseC1 classeC1;
11 }
12
```

```
ClasseA2.java
1 package acesso.pacoteA;
2
3 class ClasseA2 extends ClasseA1 {
4
5 }
6
```

```
ClasseB1.java
1 package acesso.pacoteB;
2
3 import acesso.pacoteA.ClasseA1;
4
5 class ClasseB1 extends ClasseA1 {
6
7 }
8
9 default class ClasseB2 {
10
11 }
```

```
ClasseC1.java
1 package acesso.pacoteC;
2
3 public class ClasseC1 {
4
5     ClasseA1 classeA1;
6     ClasseA2 classeA2;
7     ClasseB2 classeB2;
8     ClasseC1 classeC1;
9     ClasseC2 classeC2;
10
11 }
```

```
ClasseC2.java
1 package acesso.pacoteC;
2
3 public class ClasseC2 extends ClasseC1 {
4
5 }
6
7 private class ClasseC3{
8
9 }
10
11 protected class ClasseC4{
12
13 }
```

## 1.3 Declaração de Classes

### 1.3.2 Declarações e Modificadores de Classes

#### **Modificadores Não-Referentes a Acesso**

- ***Final***

registra que a classe não pode ser subclassificada/herdada;

- ***Abstract***

registra que uma classe não pode ser instanciada;

- ***Strictfp***

identifica que o código na classe/método se conformará com o padrão IEEE 754 para pontos flutuantes, ou seja, não irão variar conforme a plataforma;



# 1.3 Declaração de Classes

## 1.3.2 Declarações e Modificadores de Classes

### Modificadores Não-Referentes a Acesso

## ***abstract*** **VS** ***final***



```
ClasseAbstrata.java
1 package naoAcesso;
2
3 public abstract class ClasseAbstrata {
4
5 }

ClasseFinal.java
1 package naoAcesso;
2
3 public final class ClasseFinal {
4
5 }

HerdoDaAbstrata.java
1 package naoAcesso;
2
3 public class HerdoDaAbstrata extends ClasseAbstrata {
4
5     ClasseAbstrata classeAbstrata = new ClasseAbstrata();
6
7 }

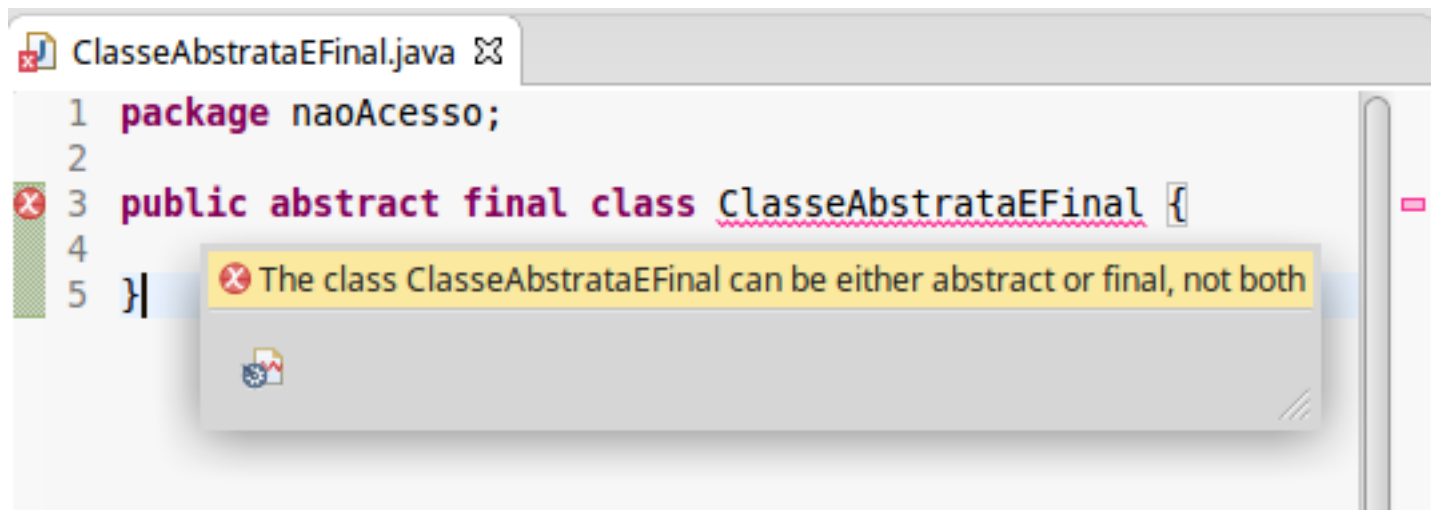
HerdoDaFinal.java
1 package naoAcesso;
2
3 public class HerdoDaFinal extends ClasseFinal {
4
5     ClasseFinal classeFinal = new ClasseFinal();
6
7 }
```

## 1.3 Declaração de Classes

### 1.3.2 Declarações e Modificadores de Classes

#### Modificadores Não-Referentes a Acesso

***abstract e final***



# 1.3 Declaração de Classes

## 1.3.2 Declarações e Modificadores de Classes

### Modificadores Não-Referentes a Acesso

## Um pouco mais de *abstract*

```
Pessoa.java
1 package exemploAbstrato;
2
3 public abstract class Pessoa {
4
5     private String nome;
6     private String endereco;
7
8     //Método abstrato
9     public abstract void pagarImposto();
10
11     //Método não abstrato
12     public String getNome() { return nome; }
13
14     public void setNome(String nome) { this.nome = nome; }
15
16     public String getEndereco() { return endereco; }
17
18     public void setEndereco(String endereco) {
19         this.endereco = endereco;
20     }
21 }

InstanciandoPessoas.java
1 package exemploAbstrato;
2
3 public class InstanciandoPessoas {
4
5     //Pessoa pessoa = new Pessoa(); ERRO!!!
6     Pessoa pf = new PessoaFisica();
7     Pessoa pj = new PessoaJuridica();
8
9     public static void main(String[] args) {
10         InstanciandoPessoas executar = new InstanciandoPessoas();
11         executar.pf.pagarImposto();
12         executar.pj.pagarImposto();
13     }
14 }
15
16

*PessoaFisica.java
1 package exemploAbstrato;
2
3 public class PessoaFisica extends Pessoa {
4
5     @Override
6     public void pagarImposto() {
7         System.out.println("PF deve pagar imposto mais barato");
8     }
9 }

*PessoaJuridica.java
1 package exemploAbstrato;
2
3 public class PessoaJuridica extends Pessoa {
4
5     @Override
6     public void pagarImposto() {
7         System.out.println("PJ deve pagar imposto mais caro");
8     }
9 }
```

@ Javadoc Declaration Console

<terminated> InstanciandoPessoas [Java Application] /opt/java/jdk1.8.0\_45/bin/java (30 de mar de 2016 05:07:49)

PF deve pagar imposto mais barato

PJ deve pagar imposto mais caro

## 1.3 Declaração de Classes

### 1.3.2 Declarações e Modificadores de Classes

#### Modificadores Não-Referentes a Acesso

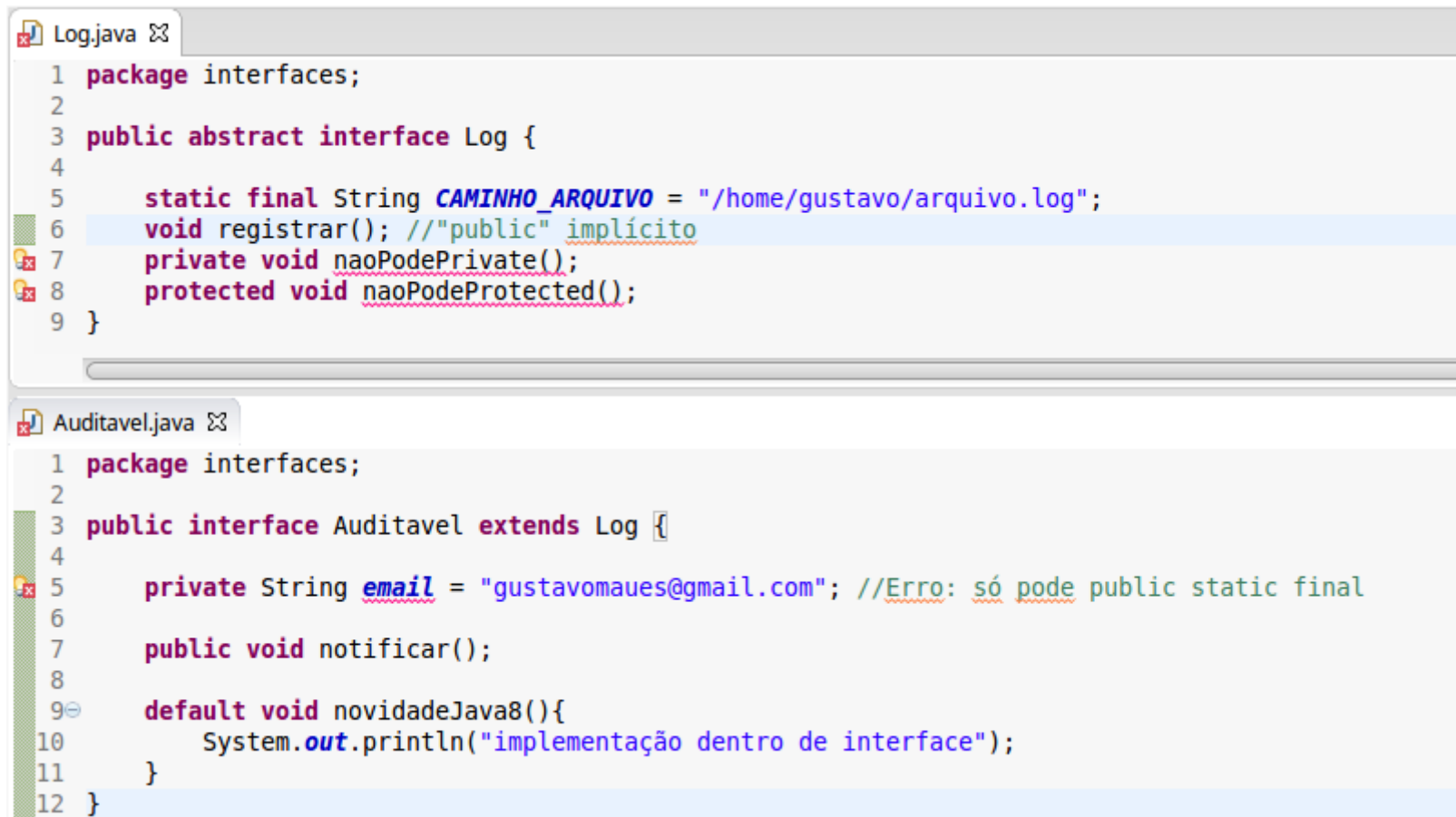
#### **Um pouco mais de *abstract* 2**

- \* Uma classe abstrata pode possuir métodos abstratos ou concretos;*
- \*\* Uma classe que contenha pelo menos um método abstrato deve, obrigatoriamente, ser marcada como abstrata;*
- \*\*\* Uma classe concreta que herde de uma abstrata deve implementar todos os métodos abstratos “de sua mãe”;*

# 1.4 Declaração de Interfaces

- Uma interface descreve um conjunto de métodos que podem ser chamados em um objeto, mas não fornece implementações concretas para todos os métodos;
  - ~~Considere uma interface como uma classe 100% abstract\*~~;
  - Todos\* os métodos são `public abstract`;
  - Todas as variáveis devem ser `public static final`, ou seja, **somente declara-se constantes e não variáveis de instância**;
  - Os métodos de interfaces **não** podem ser `static`, `final`, `native`, `strictfp` ou `synchronized`;
  - Uma interface (só) pode estender uma ou mais interfaces;
  - Uma interface não pode implementar nada;
- \* Java 8 trouxe o conceito de *default method*, permitindo a criação de métodos concretos em interfaces.

# 1.4 Declaração de Interfaces



```
Log.java
1 package interfaces;
2
3 public abstract interface Log {
4
5     static final String CAMINHO_ARQUIVO = "/home/gustavo/arquivo.log";
6     void registrar(); // "public" implícito
7     private void naoPodePrivate();
8     protected void naoPodeProtected();
9 }

Auditavel.java
1 package interfaces;
2
3 public interface Auditavel extends Log {
4
5     private String email = "gustavomaues@gmail.com"; // Erro: só pode public static final
6
7     public void notificar();
8
9     default void novidadeJava8(){
10         System.out.println("implementação dentro de interface");
11     }
12 }
```

# 1.5 Declarar Membros de Classes

## 1.5.1 Modificadores de Acesso

- Membros de classes? atributos e métodos;
- Controles de acesso: default, private, protected, public;
- Modificadores de acesso **não podem ser** aplicados a variáveis locais;
- Entenda questões relativas ao acesso: **acessar** vs **herdar**:
  - Se o código de uma classe pode acessar um membro de outra classe;
  - Se uma subclasse pode herdar um membro de sua superclasse;

# 1.5 Declarar Membros de Classes

## 1.5.1 Modificadores de Acesso

- Ao **acessar**: quando um método de uma classe tenta acessar um membro de outra classe através do operador ponto (.);

```
Animal.java
1 package modificadoresMembros;
2
3 public class Animal {
4
5     char sexo;
6     private int idade = 8;
7     protected String nome;
8     public String cor;
9
10    void nascer(){
11        System.out.println("Animal nascendo...");
12    }
13
14    public void morrer(){
15        System.out.println("Animal morrendo com "+idade+" anos");
16    }
17 }
18
```

```
Cachorro.java
1 package modificadoresMembros;
2
3 public class Cachorro {
4
5     public void testandoAcesso(){
6         Animal animal = new Animal();
7         animal.nascer();
8         animal.sexo = 'M';
9         animal.idade = 5;
10        animal.nome = "Robb";
11        animal.cor = "Cinza";
12        animal.morrer();
13    }
14 }
15
```

```
Cachorro.java
1 package modificadoresMembroOutroPacote;
2
3 import modificadoresMembros.Animal;
4
5 public class Cachorro {
6
7     public void testandoAcesso(){
8         Animal animal = new Animal();
9         animal.nascer();
10        animal.sexo = 'M';
11        animal.idade = 5;
12        animal.nome = "Robb";
13        animal.cor = "Cinza";
14        animal.morrer();
15    }
16 }
```



# 1.5 Declarar Membros de Classes

## 1.5.1 Modificadores de Acesso

- Ao **herdar**: quais membros de uma superclasse uma subclasse pode acessar através da herança;

```
Animal.java
1 package modificadoresMembros;
2
3 public class Animal {
4
5     char sexo;
6     private int idade = 8;
7     protected String nome;
8     public String cor;
9
10    void nascer(){
11        System.out.println("Animal nascendo...");
12    }
13
14    public void morrer(){
15        System.out.println("Animal morrendo com "+idade+" anos");
16    }
17 }
18
```

```
CachorroHeranca.java
1 package modificadoresMembros;
2
3 public class CachorroHeranca extends Animal {
4
5    public void testandoAcesso(){
6        nascer();
7        sexo = 'M';
8        idade = 5;
9        nome = "Robb";
10       cor = "Cinza";
11       morrer();
12    }
13 }
14
```

```
CachorroHeranca.java
1 package modificadoresMembroOutroPacote;
2
3 import modificadoresMembros.Animal;
4
5 public class CachorroHeranca extends Animal {
6
7    public void testandoAcesso(){
8        nascer();
9        sexo = 'M';
10       idade = 5;
11       nome = "Robb";
12       cor = "Cinza";
13       morrer();
14    }
15 }
```

# 1.5 Declarar Membros de Classes

## 1.5.1 Modificadores de Acesso

- ***Formalizando***... mas antes, lembre-se: antes de verificar se um membro é acessível ou não, verifique se a própria classe pode ser acessada;
- ***public***  
quando um membro é declarado *public*, todas as outras classes, independentemente do pacote, podem acessar o membro em questão.
- ***private***  
quando um membro é declarado *private*, **apenas** a própria classe pode acessá-lo;
- ***default***  
quando um membro não recebe modificador, ou seja, é *default*, pode ser acessado se a classe estiver no mesmo pacote;
- ***protected***  
quando um membro é declarado *protected*, possui o mesmo comportamento *default* para classes no mesmo pacote, porém, pode ser acessado por herança independentemente do pacote;

# Pergunta

- Compila ou não compila?

```
package modificadoresMembros;
```

```
public class Animal {
```

```
    char sexo;
```

```
    private int idade = 8;
```

```
    protected String nome;
```

```
    public String cor;
```

```
    void nascer(){
```

```
        System.out.println("Animal nascendo...");
```

```
    }
```

```
    public void morrer(){
```

```
        System.out.println("Animal morrendo com "+idade+" anos");
```

```
    }
```

```
}
```

```
package modificadoresMembrosOutroPacote;
```

```
import modificadoresMembros.Animal;
```

```
public class CachorroPergunta extends Animal {
```

```
    Animal animal = new Animal();
```

```
    public void testandoAcesso(){
```

```
        animal.nome = "Laila";
```

```
    }
```

```
}
```

# Resposta

- **Compila ou não compila?**
- Não, pois o acesso ao membro protegido “nome” da classe Animal só poderia ser acessado:
  - pela própria classe
  - por outra classe no mesmo pacote
  - por outra classe em pacote diferente através da herança
- \* **Não se engane com a herança, pois o acesso foi a uma instância de Animal...**

```
package modificadoresMembros;

public class Animal {

    char sexo;
    private int idade = 8;
    protected String nome;
    public String cor;

    void nascer(){
        System.out.println("Animal nascendo...");
    }

    public void morrer(){
        System.out.println("Animal morrendo com "+idade+" anos");
    }
}
```

```
package modificadoresMembrosOutroPacote;

import modificadoresMembros.Animal;

public class CachorroPergunta extends Animal {

    Animal animal = new Animal();

    public void testandoAcesso(){
        animal.nome = "Laila";
    }
}
```

# Pergunta

- **Compila ou não compila?**

```
package modificadoresMembros;

public class Pai {

    protected String nome = "Raimundo";

}

package modificadoresMembros;

public class Irmao {

    Filho irmao = new Filho();

    void mostrarNome(){
        System.out.println("Nome: "+irmao.nome);
    }

}

package modificadoresMembros;

public class Filho extends Pai {

    void mostrarNome(){
        System.out.println("Nome: "+nome);
    }

}
```

```
package modificadoresMembrosOutroPacote;

import modificadoresMembros.Filho;
import modificadoresMembros.Pai;

class Irma extends Pai {

    void mostrarNome(){
        System.out.println("Nome: "+nome);
    }

}

class Irmao {

    Pai pai = new Pai();

    void mostrarNome(){
        System.out.println("Nome: "+pai.nome);
    }

}
```

# Resposta

- **Compila ou não compila?**
- Não! Quando uma classe fora do pacote herda um membro ***protected***, ele torna-se ***private***.
- Todas as classes são válidas, exceto a classe **Irmão**. Por quê?

# 1.5 Declarar Membros de Classes

## 1.5.2 Modificadores Não-Referentes a Acesso em Métodos

- **Métodos *Final***
  - impede que um método possa ser substituído em uma subclasse;
- **Argumentos *Final***
  - impede que o argumento (variável local) seja alterado no método;
- **Métodos *Abstract***
  - é um método sem corpo que força a primeira subclasse concreta a implementá-lo. Termina com ponto e vírgula em vez de chaves;
- Não podemos marcar um método como ***abstract*** e ***final***, ***abstract*** e ***private*** ou ***abstract*** e ***static***;
- **Métodos *Static***
  - métodos que existem independentemente da criação de instâncias para a classe. Existe apenas uma cópia do método, que é compartilhada entre instâncias;
- **Métodos *Synchronized*, *Nativos*, *Strictfp***
  - pesquisem!

# 1.5 Declarar Membros de Classes

## 1.5.2 Modificadores Não-Referentes a Acesso em Métodos **var-args**

- **Métodos com Lista de Argumentos Variáveis (var-args)**
  - impede que um método possa ser substituído em uma subclasse;
- **Sintaxe**
  - basta inserir um sinal de reticências (...) após a declaração do tipo;
- **Restrições**
  - só podemos ter um var-args;
  - o var-args deve ser o último parâmetro da assinatura do método;

```
public void veSePode(int... numeros){ }
```

```
public void veSePode(int... numeros, String nome){ }
```

```
public void veSePode(int... numeros, String... nomes){ }
```

```
public void veSePode(String nome, int... numeros){ }
```

```
public void veSePode(int numeros...){ }
```



# 1.5 Declarar Membros de Classes

## 1.5.3 Modificadores Não-Referentes a Acesso em Variáveis

- **Tipos de Variáveis**

- Primitivos: *char*, *boolean*, *byte* (8bits), *short* (16), *int* (32), *long* (64), *double* (64) ou *float* (32);
- De Referência: variáveis que referenciam (acessam) objetos;

- **Variáveis de Instância**

- são definidas dentro classe, mas fora dos métodos; podem ter qualquer modificador de acesso, além de *final* e *transient*; não podem ser *abstract*, *synchronized*, *strictfp*, *native*;
- Atenção: se a variável for declarada *static*, torna-se uma **variável de classe** e não de instância;

- **Variáveis Locais**

- são declaradas dentro de um método; são iniciadas e destruídas no escopo do método; não podem receber **nenhum** modificador, exceto *final*;

- **Sombreamento**

- ocorre quando declaramos uma variável local com o mesmo nome de uma variável de instância (*this*).

```
public class Pessoa {  
  
    private String nome;  
  
    public void setNome(String nome){  
        this.nome = nome;  
    }  
}
```

# 1.5 Declarar Membros de Classes

## 1.5.3 Modificadores Não-Referentes a Acesso em Variáveis

- **Variáveis *Final***
  - impossibilita a alteração do valor da variável;
- **Variáveis *Transient***
  - indica a JVM que a variável deve ser ignorada em caso de serialização;
- **Variáveis *Volatile***
  - indica a JVM que cada thread que acessar a variável deve acessar a cópia presente na memória, evitando valores diferentes entre threads;
- **Variáveis *Static***
  - existem independentemente da criação de instâncias para a classe. Existe apenas uma cópia da variável, que é compartilhada entre instâncias;

# 1.5 Declarar Membros de Classes

## 1.5.4 Declarações de Arrays

- **Arrays**
  - Em Java, arrays são objetos que armazenam múltiplas variáveis do mesmo tipo, ou que sejam subclasses do mesmo tipo;
- **O que vamos aprender?**
  - criar uma referência (declarar), criar um objeto (construir) e preencher (inicializar);
- **Declarar**
  - tipo + [ ] + nome **OU** tipo + nome + [ ];
  - Exemplo: `int[ ] ids; String emails[ ];`
  - Multidimensionais: `int[ ][ ] ids; String[ ] emails[ ];`
- **Não confunda com outras linguagens!**
  - **NÃO** se inclui o tamanho do array em sua declaração.
  - `String[15] e-mails;` NÃO COMPILA!

# 1.5 Declarar Membros de Classes

## 1.5.5 Declaração de Construtores

- Em java os objetos são criados através de um construtor;
- Toda classe tem um construtor. Se não declarado explicitamente, o compilador irá criar um padrão;
- **Não confunda um construtor com um método;**
- Qual a diferença?

```
class Pessoa {  
    Pessoa(){ }  
    void Pessoa(){ }  
}
```

- O construtor ***nunca terá um tipo de retorno***, mas pode ter qualquer modificador de acesso e argumentos; deve ter o mesmo nome da classe;
- Não podem ser *static*, *final*, nem *abstract*;

# 1.5 Declarar Membros de Classes

## 1.5.6 Declaração de Enums

- Java permite definir um tipo de variável com valores pré-definidos;
- Ou seja, com uma lista de valores **enumerados**;
- Enums podem ser declarados como classes próprias ou como membros de classes. Só não pode ser declarada dentro de métodos;

```
class Pessoa {  
    enum Sexo { FEMININO, MASCULINO};  
  
    private String nome;  
    private Sexo sexo = Sexo.MASCULINO;  
}  
  
public enum Sexo {  
    FEMININO, MASCULINO;  
}
```

```
public enum Sexo {  
    FEMININO("Menina"), MASCULINO("Menino");  
  
    private String outroNome;  
  
    private Sexo(String outroNome) {  
        this.outroNome = outroNome;  
    }  
  
    public String getOutroNome() {  
        return outroNome;  
    }  
}
```

- **Atenção:** a **ordem** dos valores de um *enum* são **importantes**;
- Todo enum possui um método *static values()* que retorna os valores do enum na ordem em que foram declarados;
- Enums são um tipo especial de classe, possuem construtores, variáveis de instância, métodos, etc. Com diferenças que conheceremos durante o curso.

## **2. Orientação a Objetos em Java**

- Próximo Tópico

## 2. Orientação a Objetos em Java

- **Conceitos importantes:** herança, polimorfismo, coesão, acoplamento, composição, encapsulamento, etc.
- Quais os benefícios do encapsulamento?
- **Básico para obter encapsulamento:**
  - Proteger variáveis de instância;
  - Criar método públicos de acesso às variáveis de instância;
- \* *Mesmo que em um primeiro momento você não implemente lógica em um método de acesso, o dia que precisar fazê-lo não irá afetar quem já estava o utilizando;*

## 2. Orientação a Objetos em Java

### 2.1 Herança, É-UM, TEM-UM

- Herança para que? **Reutilização de código, polimorfismo.**
- **Object**: toda classe java é subclasse de **Object** (exceto a própria)
- **Extends**: **class Filha extends Pai { }**

```
public class Pessoa {  
  
    public static void main(String[] args) {  
  
        Pessoa gustavo = new Pessoa();  
        Pessoa thamiris = new Pessoa();  
  
        System.out.println(gustavo.equals(thamiris) ? "Iguais!" : "Diferentes!");  
  
        if (gustavo instanceof Object){  
            System.out.println("É do tipo Object!");  
        } else {  
            System.out.println("Não do tipo Object!");  
        }  
    }  
}
```



## 2. Orientação a Objetos em Java

### Herança - Reutilização de código

```
public class Conta {  
  
    private BigDecimal valor;  
    private Date dataPrevisao;  
    private Date dataEfetivacao;  
    private String observacao;  
  
    public void imprimirComprovante(){  
        System.out.println("-----COMPROVANTE-----\n");  
        System.out.println("Data: "+getDataEfetivacao()+" - Valor: R$"+getValor());  
    }  
    //getters e setters omitidos  
}  
  
public class ContaReceber extends Conta {  
  
    public void transferirParaPoupanca(){  
        //lógica da transferência...  
    }  
}  
  
public class ContaPagar extends Conta {  
  
    public void notificarProximidadeVencimento(){  
        //lógica da notificação...  
    }  
}
```

## 2. Orientação a Objetos em Java

### Herança - Polimorfismo

- Observe o método *imprimir*, ele pode receber qualquer subtipo de Conta;
- Atente que como o tipo declarado é Conta, somente os métodos de Conta podem ser chamados;

```
public class ExemploPolimorfismo {  
  
    public static void main(String[] args) {  
        ContaReceber contaReceber = new ContaReceber();  
        contaReceber.setDataEfetivacao(new Date());  
        contaReceber.setValor(BigDecimal.valueOf(30.00));  
  
        ContaPagar contaPagar = new ContaPagar();  
        contaPagar.setDataEfetivacao(new Date());  
        contaPagar.setValor(BigDecimal.valueOf(20.00));  
  
        imprimir(contaReceber);  
        imprimir(contaPagar);  
    }  
  
    static void imprimir(Conta conta) {  
        conta.imprimirComprovante();  
    }  
}
```

## 2. Orientação a Objetos em Java

### Relacionamentos É-UM e TEM-UM

- É-UM: baseia-se na herança de classe ou implementação de interfaces;

```
class TimeGrande extends Time { } //TimeGrande é um Time
```

```
class Remo implements Vencedor { } //Remo é um Vencedor
```

```
class Remo extends TimeGrande implements Vencedor { }  
//Remo é um TimeGrande, Remo é um Time, Remo é um Vencedor  
//E não esqueça: Remo é um Object
```

- É-UM: baseia-se no uso, na referência a uma instância;

```
public class Remo {  
    private Estadio baenao; //Remo tem um estádio  
    private Torcida fenomenoAzul; //Remo tem uma torcida  
    private Fregues paysandu; //Remo tem um freguês  
}
```

- *Os exemplos acima demonstram que a decisão entre usar um relacionamento É-UM ou TEM-UM não deve basear-se apenas no aproveitamento de código, mas apresentar uma lógica e semântica coerente. Por exemplo, seria uma péssima decisão Remo herdar de Endereco como justificativa de aproveitamento dos atributos de Endereco. Remo é um Endereco? Definitivamente, não.*

## 2. Orientação a Objetos em Java

### Polimorfismo

- Todo objeto Java que tiver mais de um caso É-UM é considerado polimórfico;
- Então, se todas as classes são, também, Object, todas são polimórficas;
- Quando criamos uma variável de referência, ela pode se referir ou próprio tipo ou a qualquer subtipo do tipo declarado;
- Uma variável de referência pode ser de um tipo de classe ou de interface; Se for de uma interface, pode se referir a qualquer classe que implemente a interface;
- Java não permite herança múltipla, ou seja, uma classe pode estender diretamente apenas uma outra classe, mas pode-se criar uma hierarquia;
- Mas Java possui **interfaces**! Vejamos...

## 2. Orientação a Objetos em Java

# Polimorfismo

```
public class Animal {  
    protected int idade;  
    public void viver(){ }  
}
```

```
public interface Mamifero {  
    void mamar();  
}
```

```
public interface Caçador {  
    void caçar();  
}
```

```
public class Boi extends Animal implements Mamifero {  
    public void mugir(){ System.out.println("Muuuuh..."); }  
    public void mamar() { }  
}
```

```
public class Cachorro extends Animal implements Caçador, Mamifero {  
    public void latir(){  
        System.out.println("Au au");  
    }  
  
    public void caçar(){ }  
    public void mamar() { }  
}
```

