

### Aula 06

# Implementação Paradigma de objetos II

## Professores:

Anarosa Alves Franco Brandão (PCS 2302)  
Marcos A. Simplício Junior (PCS 2302/2024)  
Ricardo Luis de Azevedo da Rocha

Monitores: Felipe Leno, Michel Bieleveld, Diego Queiroz



# Agenda

- Pacotes
- Visibilidade
- Classes abstratas
- Interfaces
- Exceções
- Modelagem da dinâmica

# Java Packages

- Uma coleção de classes relacionadas pode ser agrupada em “Packages”. Por exemplo:
  - java.net contém as classes relacionadas a rede
  - java.awt as classes relacionadas a AWT (GUI)
  - java.io as classes relacionadas a manipulação de entradas e saídas
  - É possível ainda criar seus próprios pacotes de classes...
- Para usar uma classe de um package diferente do seu, utilize a palavra reservada **import**:

```
import java.math;
```

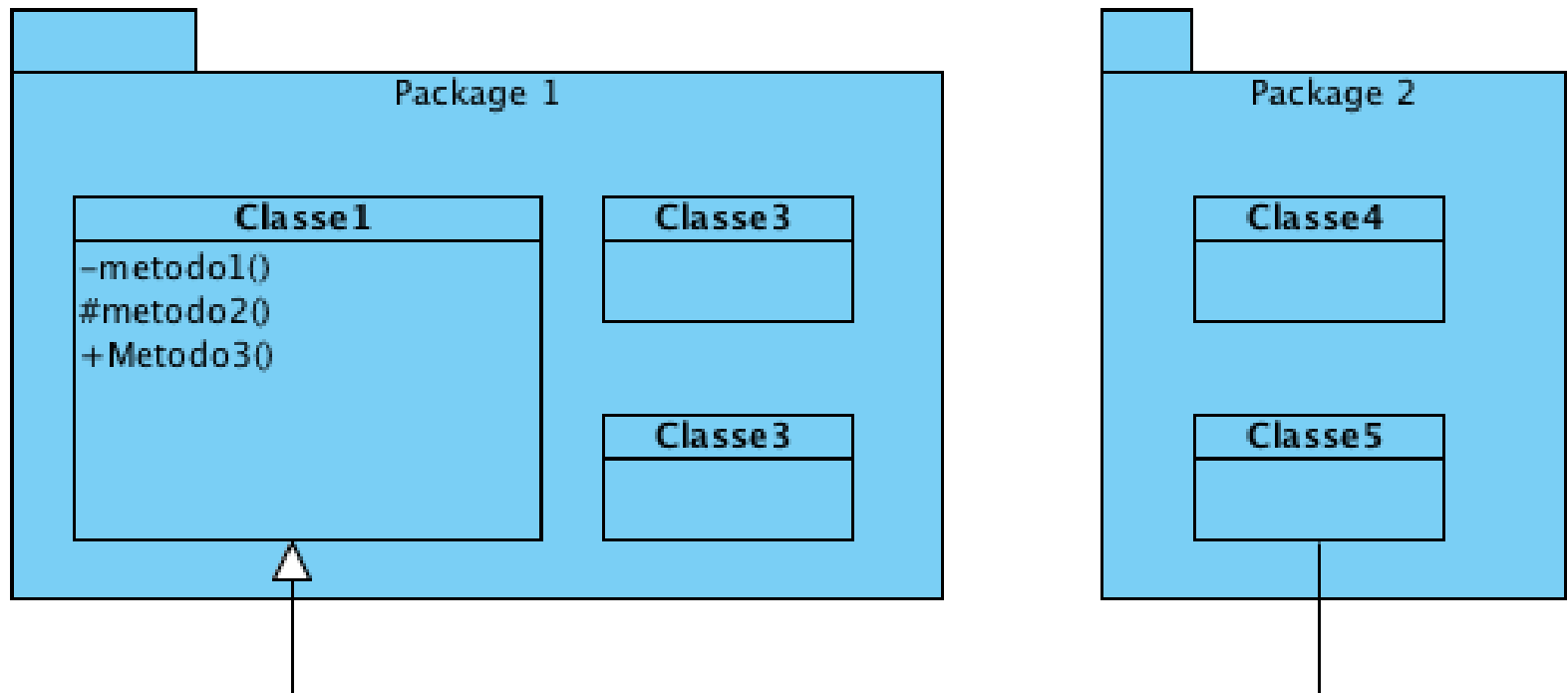
# Visibilidade das Informações em Java

- Java provê três níveis de visibilidade de informação. São eles:
  - Public
  - Protected
  - Private
- Classes, variáveis e métodos podem ser precedidos por uma das palavras-chave acima.
  - Public:
    - Visível a todas as entidades
  - Protected:
    - Visível somente para subclasses e classes no mesmo package
  - Private:
    - Visível somente na classe

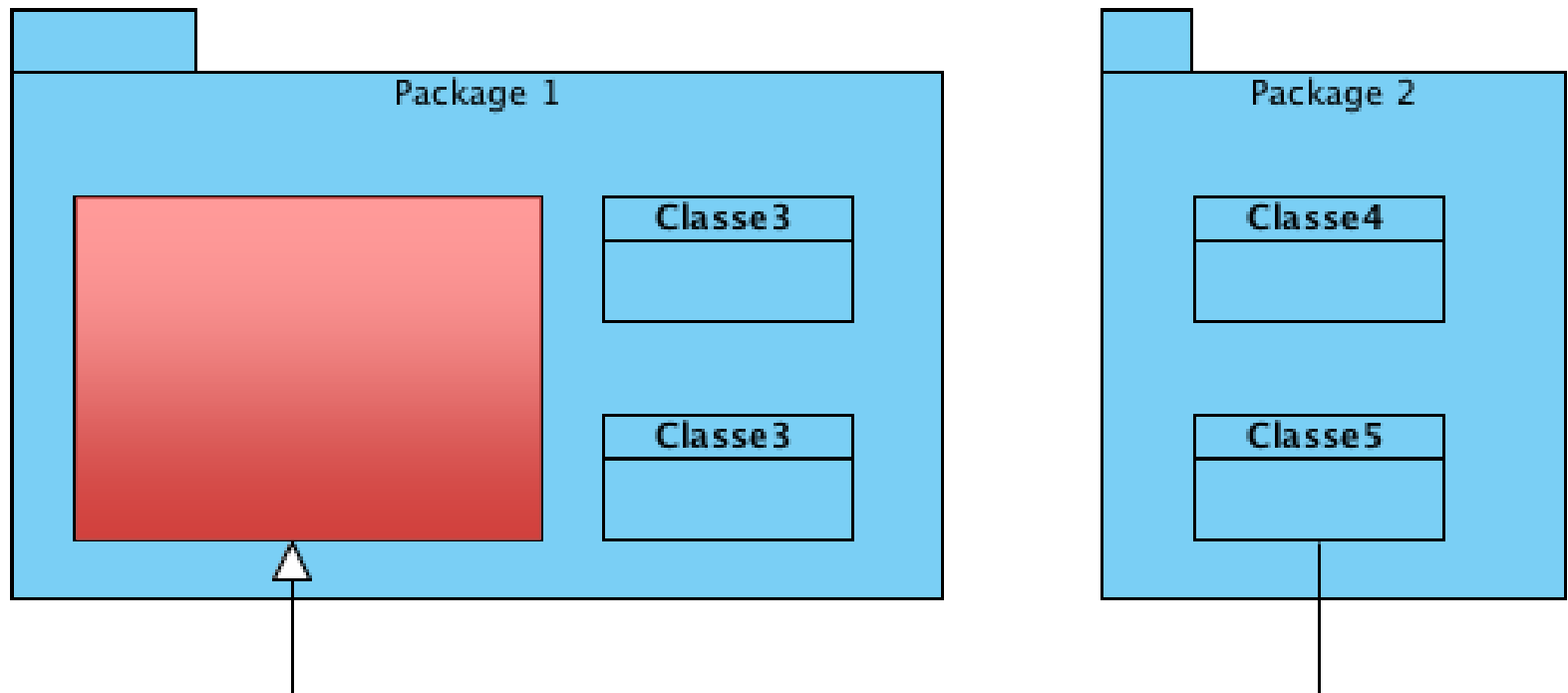
Repare no  
identificador de  
visibilidade

ClasseExemplo
-atributoPrivado
+metodoPublico()
#metodoProtected()

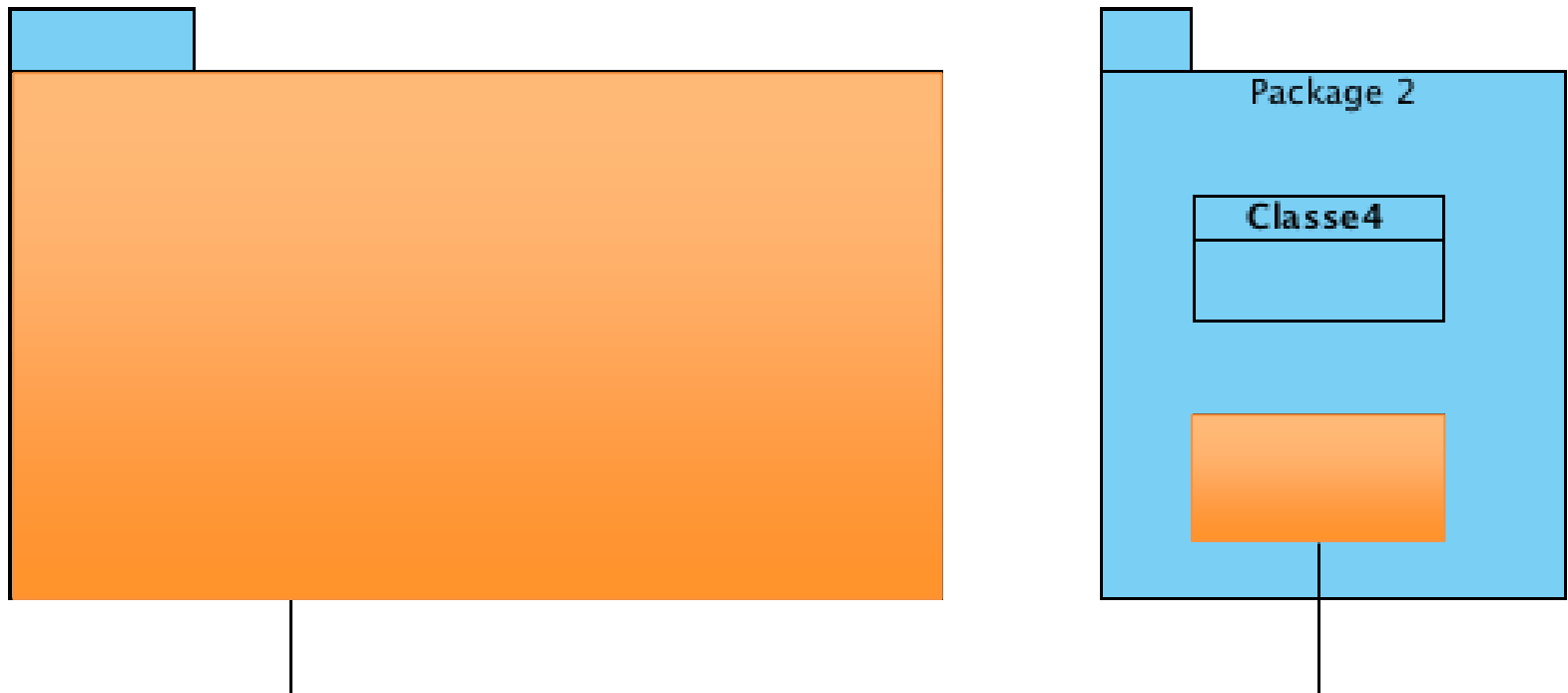
# Visibilidade das Informações em Java



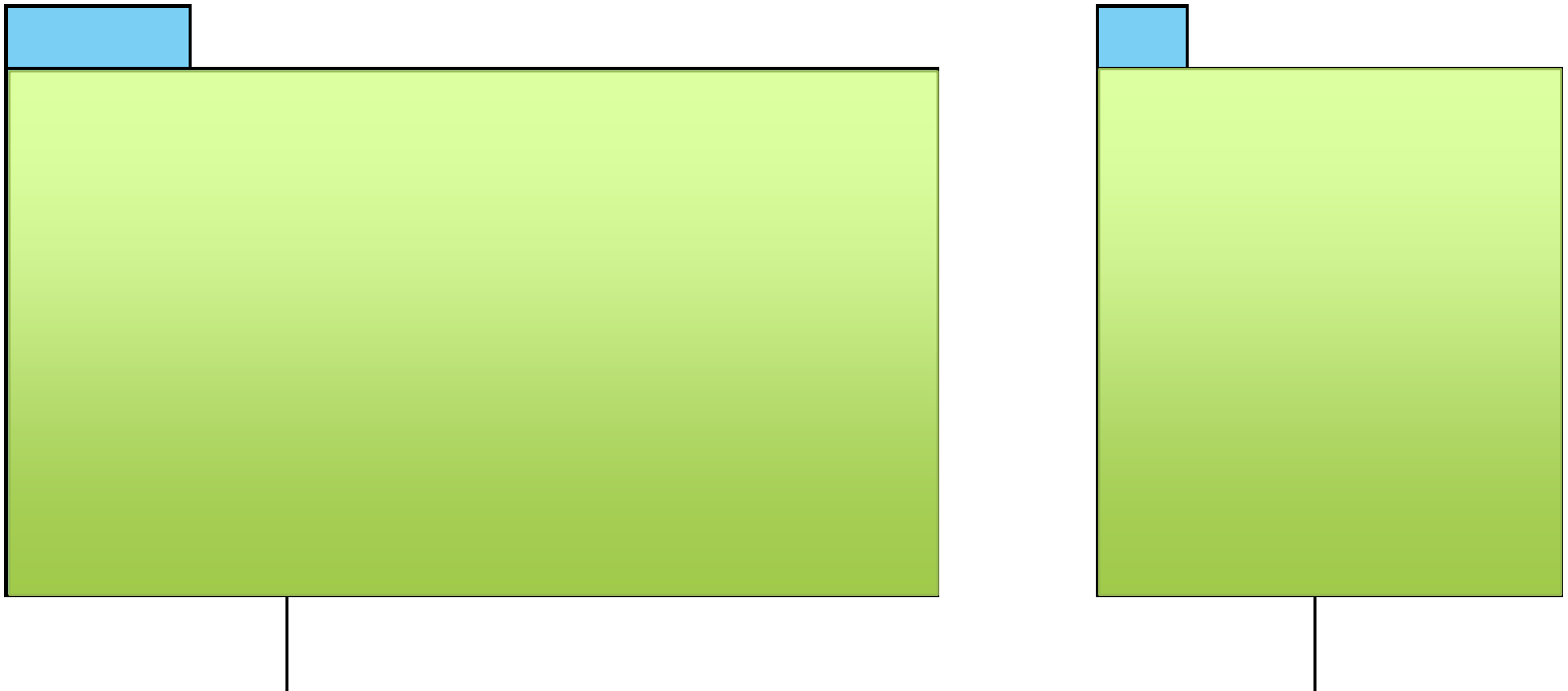
# Visibilidade das Informações em Java



# Visibilidade das Informações em Java



# Visibilidade das Informações em Java





# Exemplos em Java

```
package mvn;

import java.util.BitSet;

public class Bits8 extends BitSet implements Comparable<Bits8>{

    public static final int BYTE_SIZE= 8;
    public static final int NIBBLE_SIZE= 4;
    public static final int HEXBYTE_SIZE= BYTE_SIZE / NIBBLE_SIZE;

    public Bits8(){
        super(BYTE_SIZE);
    }

    public Bits8(byte initialValue){
        this();
        setValue(initialValue);
    }

    private void setValue(byte value){
        for(int i = 0; i < BYTE_SIZE; i++){
            this.set(i, (1 & (value >> i)) != 0);}
        }
    }
```

# Exemplos em Java

```
package mvn; ←  
import java.util.BitSet; ←  
  
public class Bits8 extends BitSet implements Comparable<Bits8>{  
    public static final int BYTE_SIZE= 8;  
    public static final int NIBBLE_SIZE= 4;  
    public static final int HEXBYTE_SIZE= BYTE_SIZE / NIBBLE_SIZE;  
  
    public Bits8(){  
        super(BYTE_SIZE);  
    }  
    public Bits8(byte initialValue){  
        this();  
        setValue(initialValue);  
    }  
    private void setValue(byte value){  
        for(int i = 0; i < BYTE_SIZE; i++){  
            this.set(i, (1 & (value >> i)) != 0);  
        }  
    }  
}
```

# Exemplos em Java

```
package mvn;

import java.util.BitSet;

public class Bits8 extends BitSet implements Comparable<Bits8>{

    public static final int BYTE_SIZE= 8;
    public static final int NIBBLE_SIZE= 4;
    public static final int HEXBYTE_SIZE= BYTE_SIZE / NIBBLE_SIZE;

    public Bits8(){
        super(BYTE_SIZE);
    }

    → public Bits8(byte initialValue){
        this();
        setValue(initialValue);
    }

    → private void setValue(byte value){
        for(int i = 0; i < BYTE_SIZE; i++){
            this.set(i, (1 & (value >> i)) != 0);}
        }
    }
```

# Métodos Estáticos

- Em algumas situações faz sentido ter um método em uma “Classe” ao invés de um “Objeto”
  - Estes métodos são chamados de “Estáticos” (**static**)
  - Para chamar um método estático a sintaxe é a seguinte:
    - `className.methodName(parameter1, parameter2... )`
    - Métodos Estáticos são usados para funcionalidades que se aplicam ao tipo de objeto e não ao objeto em si (instância da classe)
    - Métodos Estáticos são úteis pois podem ser chamados sem instanciar a classe

# Métodos estáticos – exemplo Java

```
package mvn;
```

```
public class MvnControle{
    private static final int MIN_ADDRESS= 0x0000;
    private static final int MAX_ADDRESS= 0x0FFF;
    private static final String DEV_NAME_TECLADO= "Teclado";
    // Este nome deve ser igual ao da classe
    private static final String DEV_NAME_MONITOR= "Monitor";
    private static final String DEV_NAME_IMPRESSORA= "Impressora";
    private static final String DEV_NAME_DISCO= "Disco";
```

```
    public MvnControle(){
        this.memoria = new Memoria(MIN_ADDRESS, MAX_ADDRESS);
        this.io = new GerenciadorDispositivos();
        this.cpu = new UnidadeControle(io, memoria);}

    ...
```

```
    public static String availableDevices(){
        StringBuilder out = new StringBuilder(MSG_HEADER_TIPOSDISPOSITIVOSDISPONIVEIS);
        out.append(System.getProperty("Line.separator"));
        for(int i = 0; i < DEVICES.Length; i++){
            out.append(String.format(" %-10s -> %d", DEVICES[i][0], i));
            out.append(System.getProperty("Line.separator"));}
        return out.toString();
    }
```

# Métodos estáticos – exemplo Java

```
package mvn.controle;
```

```
...
```

```
public class PaineiControle {...
```

```
    public PaineiControle(MvnControle mvn, boolean debug) {  
        this.mvn = mvn;  
        this.terminal = new TerminalPadrao(debug);  
        initialize();  
    }  
    ...
```

```
    public void dispositivos() throws MVNException {  
        terminal.exibeLinha(mvn.listDispositivos());  
        terminal.pulaLinha();  
        char acao = terminal.obtem(MSG_PROMPT_ALTERARDISPOSITIVO, " ").charAt(0);  
        if (acao != ADICIONAR && acao != REMOVER) {return;}  
        terminal.exibe(MvnControle.availableDevices());  
        String strTipo = terminal.obtem(MSG_PROMPT_TIPODISPOSITIVO, "");  
        if (strTipo.isEmpty()) {return;}  
    }  
    ...
```

  
Nome da classe  
Método estático



# Classes abstratas em Java

- Classes abstratas são classes usadas com a finalidade de fornecer uma subclasse apropriada a partir da qual outras classes podem herdar e compartilhar um design comum
  - Classes abstratas não são instanciáveis
  - Codificação incompleta
    - Alguns métodos são implementados apenas pelas subclasses

# Classes abstratas em Java: Exemplo

```
package mvn.controle;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.PrintStream;

public abstract class Terminal{

    private static final String PROMPT_STRING= "> ";
    private static final int TERMINAL_WIDTH= 80;
    private static final String ERR_NAO_ESPECIFICADO= "Erro não especificado.";
    private static final String ERR_ENTRADA_TECLADO= "Erro na entrada do terminal";
    private BufferedReader in;
    private PrintStream out;
    private PrintStream err;
    private boolean ativado;
    private boolean debug;

    ...
}
```



# Classes abstratas em Java: Exemplo

```
package mvn.controle;

...

public abstract class Terminal{

    ...

    public Terminal(boolean debug){
        this.debug = debug;
        this.ativado = true;
        in = setIn();
        out = setOut();
        err = setErr();
    }

    public Terminal(){
        this(false);
    }

    public void desativa(){
        ativado = false;}

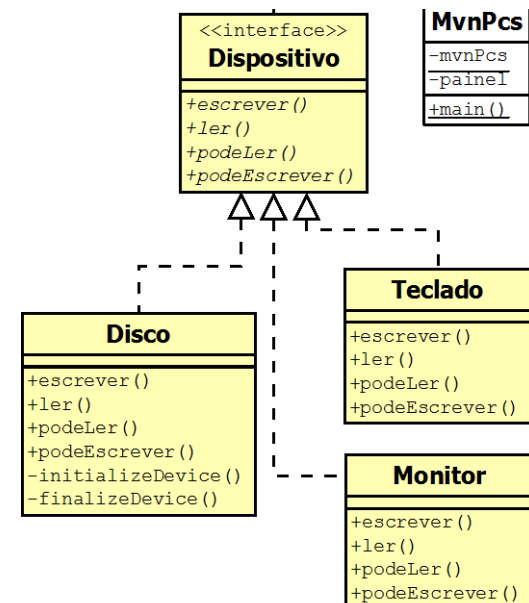
    public void ativa(){
        ativado = true;}

    ...

    protected abstract BufferedReader setIn();
    protected abstract PrintStream setOut();
    protected abstract PrintStream setErr(); ...
}
```

# Interfaces em Java

- Interfaces Java (diferente de interfaces visuais – GUIs) são “Contratos” de código, onde uma classe que adere a este contrato (que implementa a interface) garante que os serviços do contrato serão implementados por ela.



# Vantagens de usar Interfaces

- Seguindo o conceito de polimorfismo, se um método recebe como parâmetro um tipo interface, é possível chamá-lo passando qualquer classe que implemente a interface em questão:

```
private void addDispositivo(int deviceType, int  
logicalUnit, Dispositivo newDevice) throws  
MVNException{}
```

- Embora as classes em java só possam herdar de uma classe, estas podem implementar quantas interfaces forem necessárias.

# Exemplos em Java

```
package mvn;

import mvn.controle.MVNException;

public interface Dispositivo{

    public static final String ERR_WRITEONLYDEVICE= "Dispositivo \"%s\"  
disponível somente para escrita.";

    public static final String ERR_READONLYDEVICE= "Dispositivo \"%s\"  
disponível somente para leitura.";

    public void escrever(Bits8 in) throws MVNException;

    public Bits8 ler() throws MVNException;

    public boolean podeLer();

    public boolean podeEscrever();

    public void reset() throws MVNException;

    public Bits8 skip(Bits8 val) throws MVNException;

    public Bits8 position() throws MVNException;

    public Bits8 size() throws MVNException;
```

# Exemplos em Java

```
package mvn.dispositivo;

public class Disco implements Dispositivo {

    public Disco(String arquivo, char modoOperacao) throws MVNException {
        switch (modoOperacao) {
            case MODO_LEITURA:
                this.modoOperacao = LEITURA;
                break;
            case MODO_ESCRITA:
                this.modoOperacao = ESCRITA;
                break;
            case MODO_LEITURAESCRITA:
                this.modoOperacao = LEITURAESCRITA;
                break;
            default:
                this.modoOperacao = INVALIDO; }
        this.arquivo = new File(arquivo);
        outFile = null;
        inFile = null;
        initializeDevice();
    }
}
```

# Exemplos em Java

```
package mvn.dispositivo;

public class Disco implements Dispositivo {
    ...

    public void escrever(Bits8 in) throws MVNException{
        //código de escrever} ;

    public Bits8 ler() throws MVNException{
        //código de ler} ;

    public boolean podeLer(){
        return modoOperacao == LEITURA || modoOperacao == LEITURAESCRITA;}

    ...
}
```

# Polimorfismo via Classes (Concretas), Classes Abstratas e Interfaces

- Polimorfismo via classes concretas
  - Herança onde todos os métodos a serem reutilizados (ou sobrescritos) pelas classes filhas devem ser implementados na classe mãe.
- Polimorfismo via classes abstratas
  - Herança onde alguns métodos a serem reutilizados (ou sobrescritos) pelas classes filhas devem ser implementados na classe mãe e outros definem apenas os “contratos”, para implementação na classe filha.
- Polimorfismo via interfaces
  - Herança onde são definidos apenas os “contratos” (assinatura dos métodos) na classe mãe, para implementação nas classes filhas.

# Introdução a Exceções

- Em Java, o projetista do sistema pode deixar clara a intenção de que, em um bloco de código, pode ocorrer uma situação de erro a ser tratada. Exemplos:
  - Um programa que abre um arquivo do disco, deve tratar a situação de o arquivo não existir;
  - Uma calculadora deve tratar a situação de o usuário tentar efetuar uma divisão por Zero.
- Mas como são representadas estas situações de erro em Java?
  - Tais situações são conhecidas como exceções





# Representação de Exceções

- A biblioteca Java possui uma série de classes que herdam da classe *Exception* (daí o nome Exceção!) como:
  - `DateFormatException`;
  - `IOException`
  - ...
- Da mesma forma, os programas construídos por você podem lançar exceções. Basta criar uma classe que herda de `Exception`
  - O código da MVN possui uma exceção específica chamada `MVNException`.

# Lançando Exceções (1)

- No caso de um programa Java precisar lançar uma exceção, por conta de uma situação que não pertence ao fluxo natural do sistema, a palavra reservada **throw** deve ser utilizada:

```
private void loadTextFiletoMemory(String args) throws  
MVNException{  
    ...  
  
    if(!args.isEmpty()){  
        ...  
    }else{  
        throw new MVNException(MSG_ERRO_ABRIR_ARQUIVO);  
    }  
}
```

## Lançando Exceções (2)

- Note ainda, que a assinatura do método deve indicar quais exceções podem ser lançadas em sua implementação (estrutura **throws**):

```
private void loadTextFiletoMemory(String args) throws
MVNException{
    ...

    if(!args.isEmpty()){
        ...
    }else{
        throw new MVNException(MSG_ERRO_ABRIR_ARQUIVO);
    }
}
```

# Tratando Exceções (1)

- Para tratar exceções no código de nossos programas, temos a estrutura **try** que deve ser usada da seguinte forma:

```
try{  
    //Código que pode lançar exceções  
}catch(MVNException e){  
    //Tratamento da exceção  
}finally{  
    //Código que roda  
    //independente da exceção ocorrer  
}
```



## Tratando Exceções (2)

```

public void reset() throws MVNException {
    if (podeLer()) {
        try {
            inFile.close();
            inFile = new FileInputStream(arquivo);
        } catch (IOException ex) {
            throw new MVNException(ERR_IOERROR,
                                    arquivo.getName());
        }
    } else {
        // modo de operacao inadequado
        throw new MVNException(ERR_WRITEONLYDEVICE,
                                this);
    }
}

```

## Tratando Exceções (3)

- Note que a estrutura **catch** recebe como parâmetro o tipo de exceção que deve ser tratada. Para tratar qualquer exceção basta usar o tipo mais primitivo de exceção: a **Exception**

```
try{  
    ...  
}catch(MVNException e){  
    terminal.erro(e);  
}catch(Exception e){  
    terminal.erro(ERR_EXECUCAO_TERMINAL, e);  
}
```



## Tratando Exceções (4)

- Exceções não tratadas, causam a interrupção do programa, portanto crie a prática de pensar que exceções podem ocorrer em seu código e tratá-las.
- Se o seu código chama uma biblioteca que pode causar exceções, estas devem obrigatoriamente ser tratadas (estrutura **try**) **OU** o seu código deve indicar que exceções podem ser lançadas novamente para o chamador (estrutura **throws**)

# Modelando a dinâmica do código

- Enquanto o Diagrama de Classes (usado até o momento) tem o objetivo de representar a visão estrutural de um sistema OO, o diagrama de sequências tem como objetivo representar a visão comportamental do sistema.
  - Nele, as classes são dispostas em linhas de tempo visual e a troca de mensagens (chamada de métodos) é representada com setas indicando o sentido da mensagem
  - As barras verticais azuis na linha do tempo representam o tempo de vida da chamada



# Iniciando a MVN

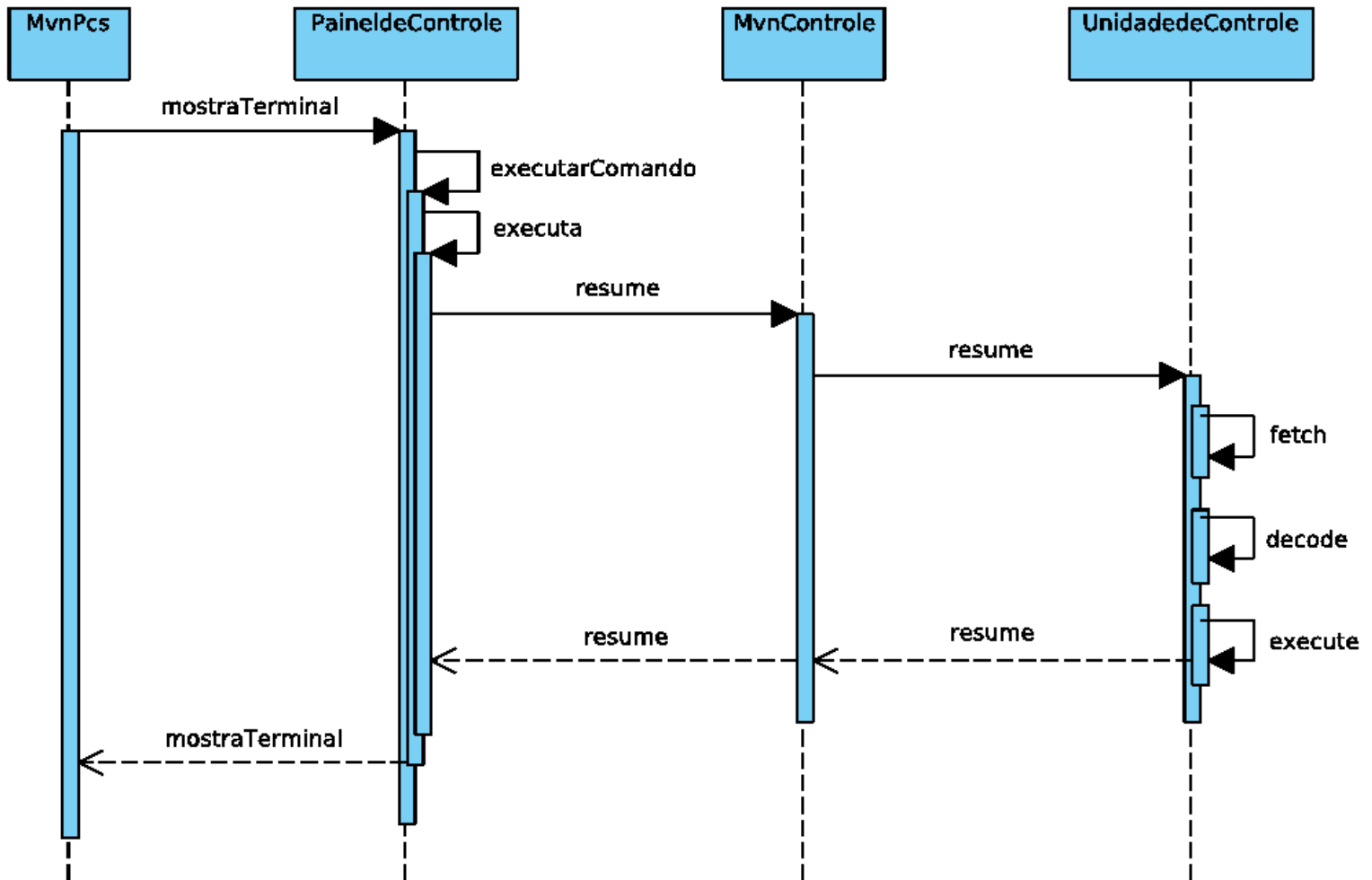
```
package mvn;

import mvn.controle.PainelControle;

/**
 * Classe que inicia a MVN.
 */
public class MvnPcs{
    /** Controlador da MVN */
    private static MvnControle mvnPcs;
    /** Painel de Controle da MVN */
    private static PainelControle painel;

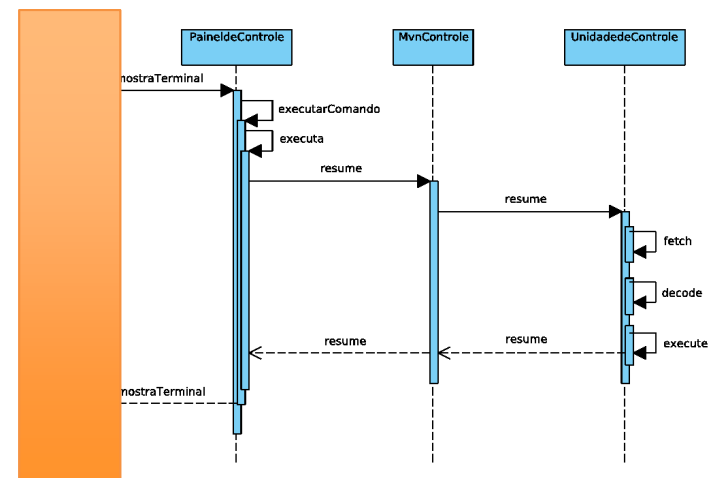
    public static void main(String args[]){
        mvnPcs = new MvnControle();
        painel = new PainelControle(mvnPcs, false);
        painel.mostrarTerminal();
    }
} // Fim da Classe MvnPcs
```

# Dinâmica de Chamada de Métodos



# Exemplo de Código do Diagrama de Sequência

```
public static void main(String args[]) {
    mvnPcs = new MvnControle();
    painel = new PainelControle(mvnPcs, false);
    painel.mostrarTerminal();
}
```

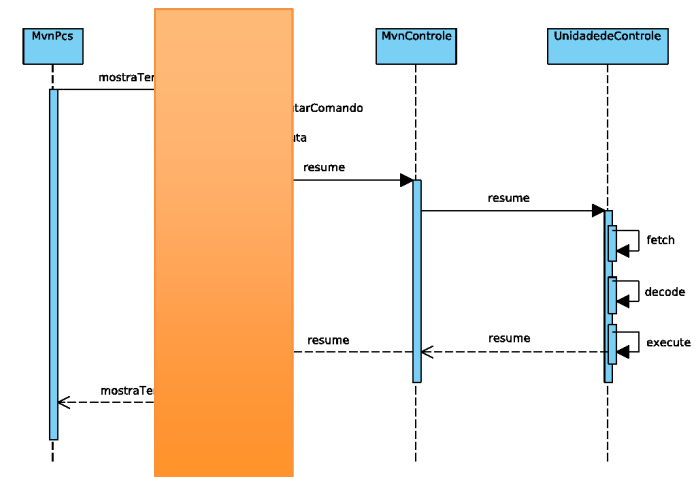


# Exemplo de Código do Diagrama de Sequência

```

public void mostrarTerminal() {
    ...
    try{
        leTerminal = executaComando(comando,
linhaComando.toString().trim());
    }catch (MVNException e) {
        terminal.erro(e);
    }catch (Exception e) {
        terminal.erro(ERR_EXECUCAO_TERMINAL, e);
    }
    ...
}

```

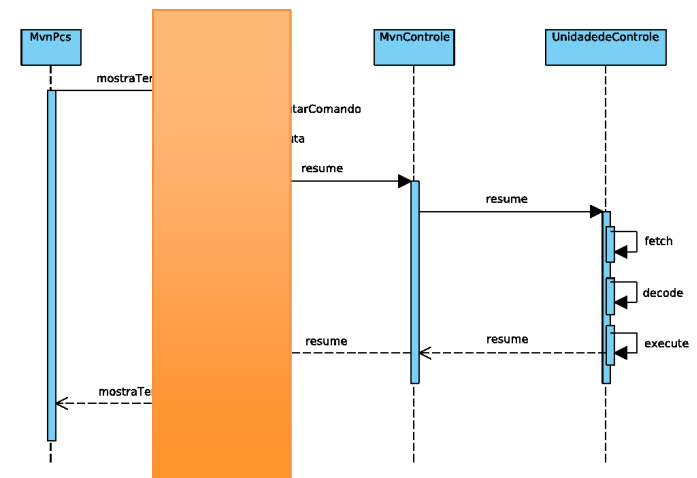


# Exemplo de Código do Diagrama de Sequência

```
private void executa(String[] args) throws MVNException{
    ...
    boolean continueRunning;
    do{
        continueRunning = mvn.resume();

        if(outputBuffer.length() > 0){
            terminal.exibe(outputBuffer.toString());
            outputBuffer.delete(0, outputBuffer.length());
        }

        ...
    }while(continueRunning);
}
```

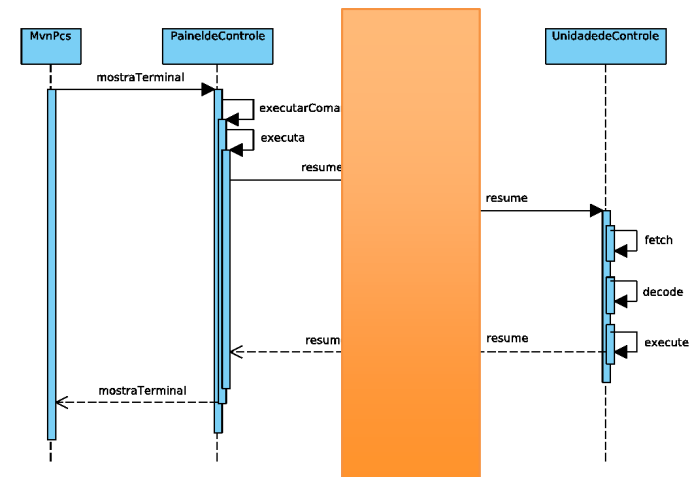


# Exemplo de Código do Diagrama de Sequência

```
public boolean resume() throws MVNException{
    boolean stillrunning = cpu.resume();

    if (showRegs) {
        outputInfo(cpu.obterRegs().toString());
    }

    return stillrunning;
}
```



# Exemplo de Código do Diagrama de Sequência

```
public boolean resume() throws MVNException{
    fetch();
    decode();
    execute();

    return (regs.getRegister(OP).toInt() != HM);
}
```

