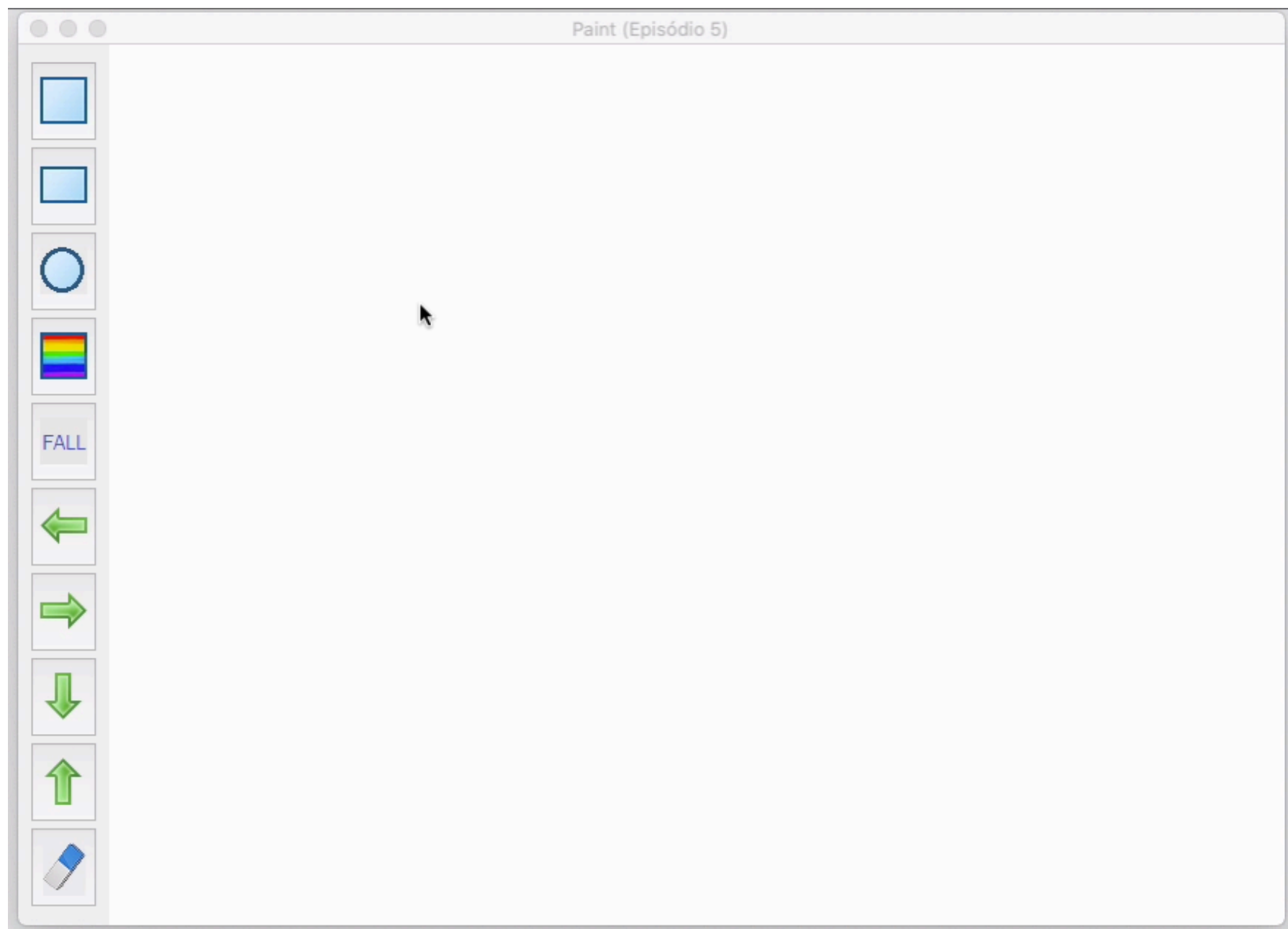


Informações

Mini-teste prático 2 a final vai ser na semana 14-18 Dezembro

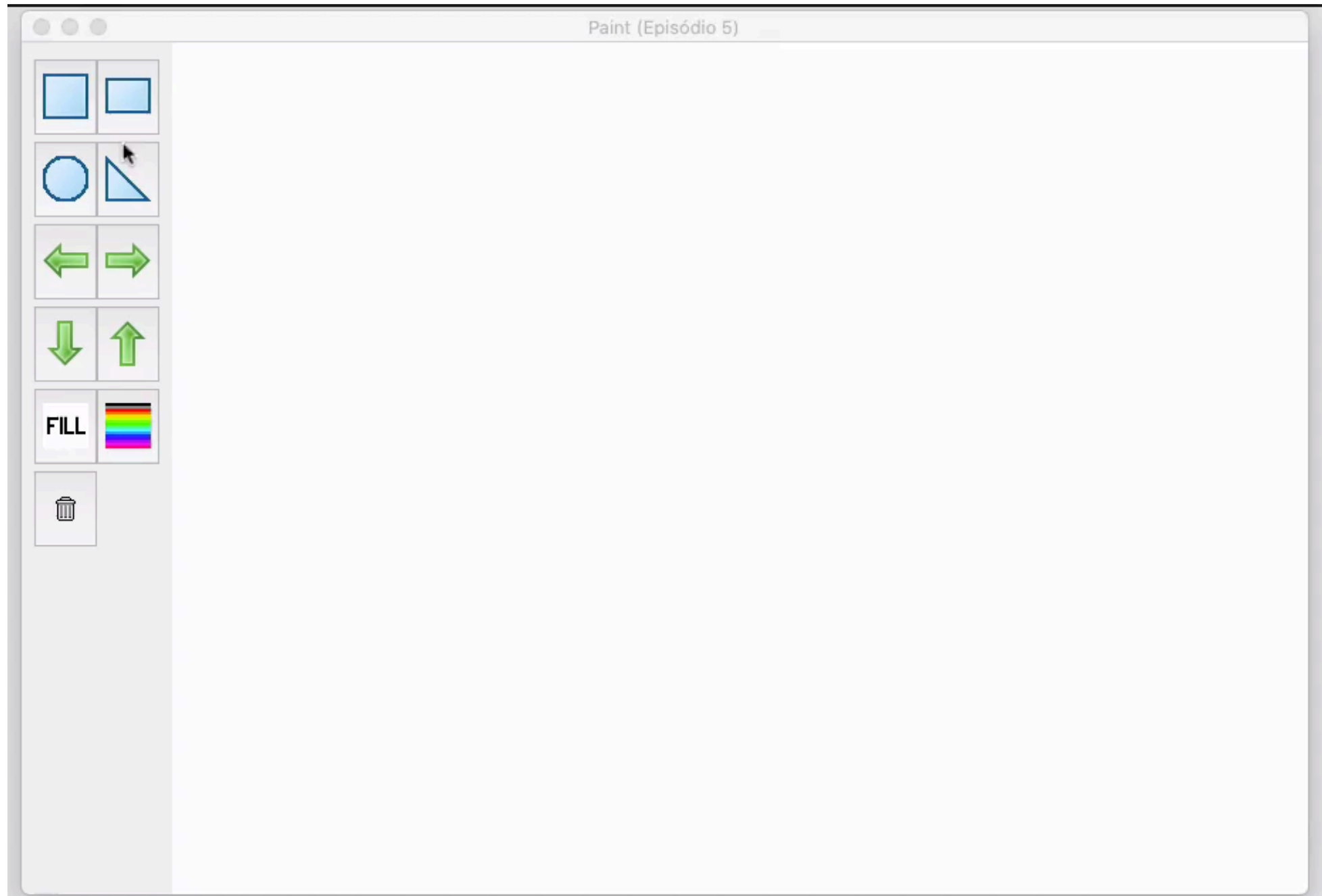
Paint - Episódio 4

Hall of fame



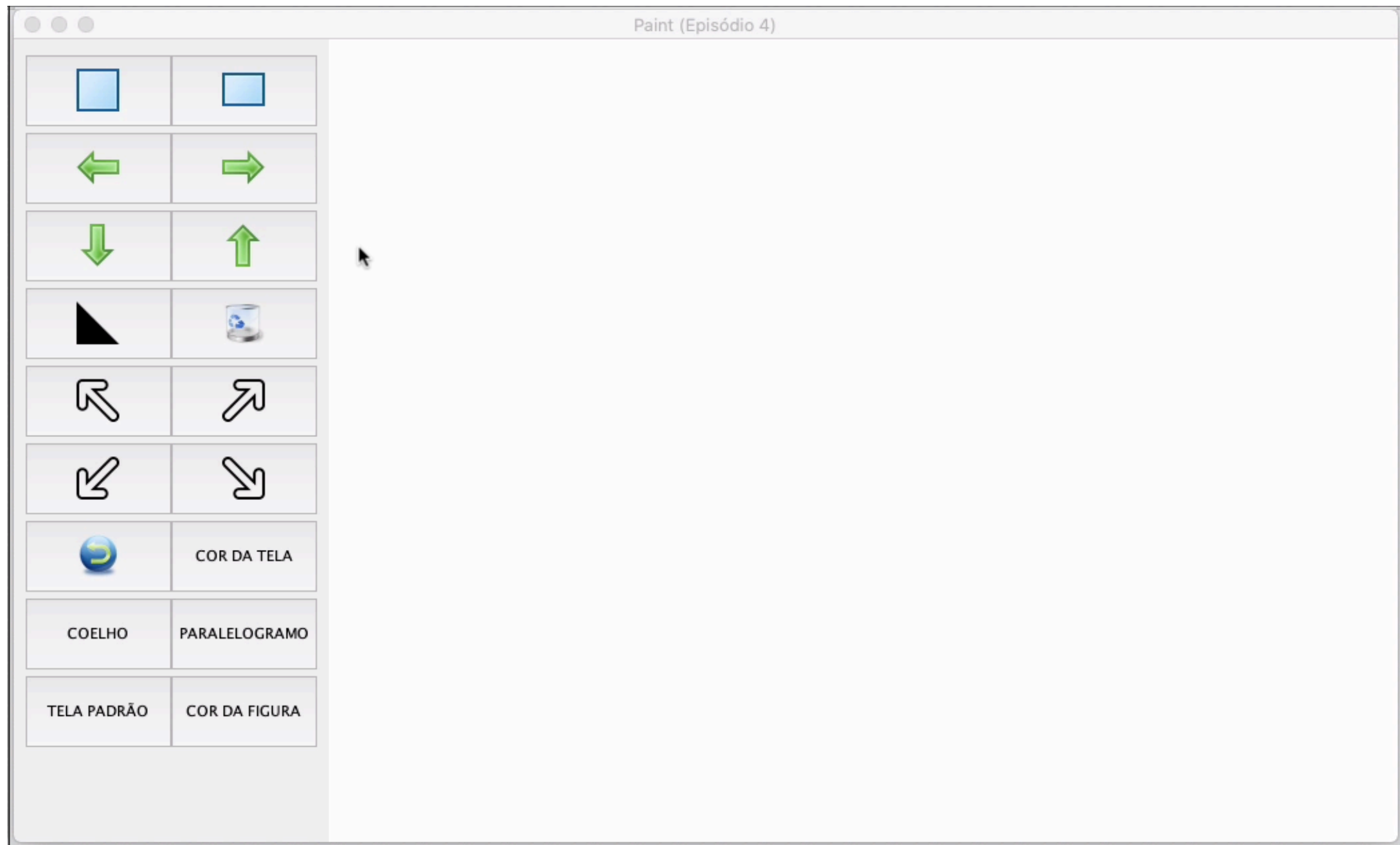
Paint - Episódio 4

Hall of fame

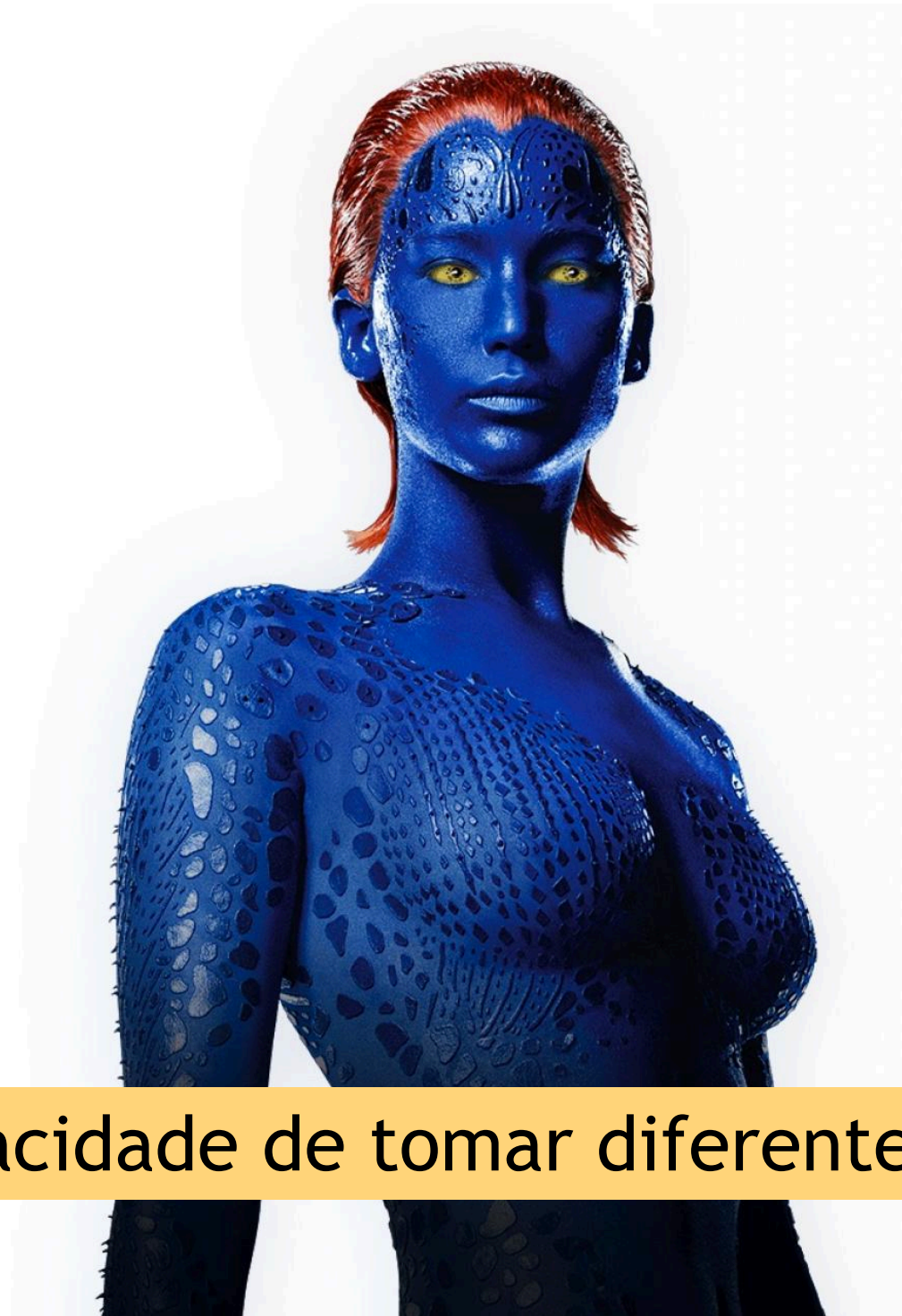


Paint - Episódio 4

Hall of fame



Polimorfismo



Capacidade de tomar diferentes formas

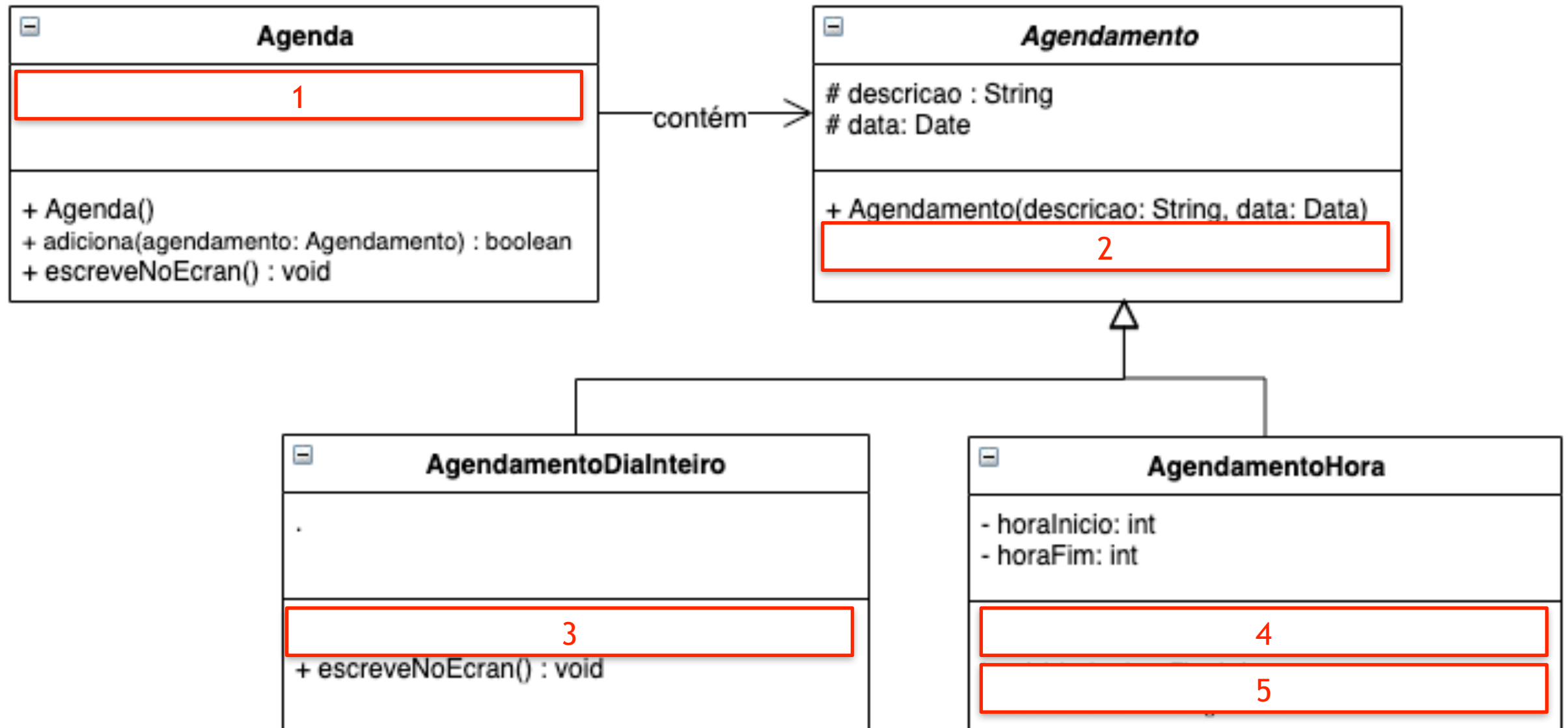
Agenda

Pretende-se desenvolver uma agenda que permita registar dois tipos de agendamento:

- Agendamento de dia inteiro
- Agendamento com hora de início e fim

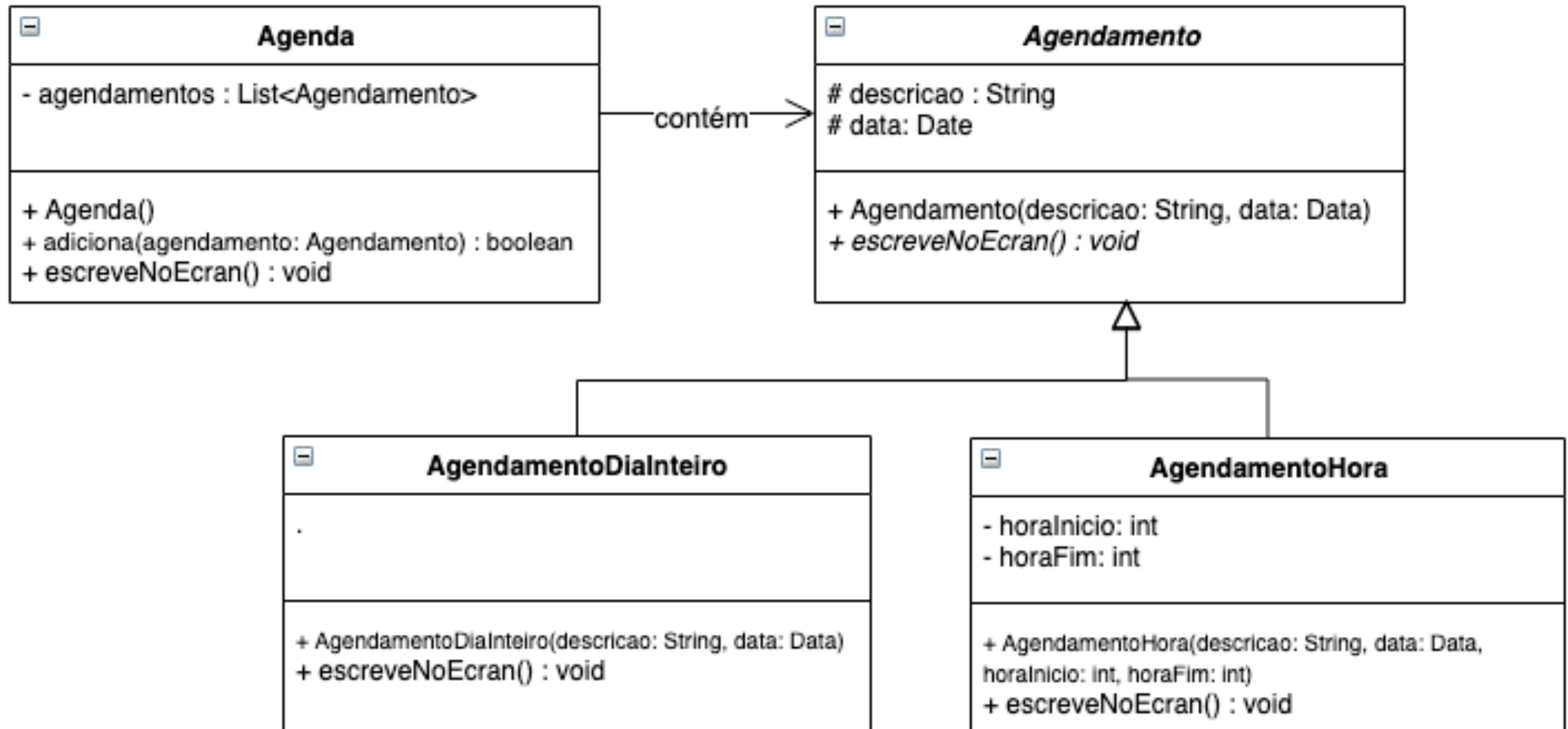
	Fri 24	Sat 25
3		Passeio ao Alentejo
0.10 12:00		
	FP - Compensação 12:00 – 14:00 S.0.10	
.10 16:00		
.10 18:00		
	LP2 - S.0.10 18:30 – 20:30	

Exercício



Completar o diagrama.
Enviar por teams para
p4997

Resolução




```
abstract class Agendamento {  
  
    protected String descricao;  
    protected Date data;  
  
    public Agendamento(String descricao,  
                        Date data) {  
        this.descricao = descricao;  
        this.data = data;  
    }  
  
    public abstract void mostraNoEcran();  
}
```

Agenda

```
class AgendamentoDiaInteiro extends Agendamento {  
  
    public AgendamentoDiaInteiro(String descricao, Date data) {  
        super(descricao, data);  
    }  
  
    @Override  
    public void mostraNoEcran() {  
        System.out.println("Agendamento dia inteiro (dia " +  
                            data + "): " + descricao);  
    }  
}  
  
class AgendamentoHora extends Agendamento {  
  
    private int horaInicio;  
    private int horaFim;  
  
    public AgendamentoHora(String descricao, Date data,  
                           int horaInicio, int horaFim) {  
        super(descricao, data);  
        this.horaInicio = horaInicio;  
        this.horaFim = horaFim;  
    }  
  
    @Override  
    public void mostraNoEcran() {  
        System.out.println("Agendamento (dia " + data + ") das "  
                            + horaInicio + "h às " + horaFim + ": " + descricao);  
    }  
}
```

Classe declarada vs classe instanciada

```
public static void main(String[] args) {  
    AgendamentoDiaInteiro a1 = new AgendamentoDiaInteiro("Férias", new Date());  
    a1.mostraNoEcran();  
  
    AgendamentoHora a2 = new AgendamentoHora("Aulas", new Date(), 16, 18);  
    a2.mostraNoEcran();  
}
```

AgendamentoDiaInteiro a1 = new AgendamentoDiaInteiro(...);



Classe declarada



Classe instanciada

Classe declarada vs classe instanciada

E se a classe declarada puder ser diferente da classe instanciada?

```
Agendamento a1 = new AgendamentoDiaInteiro(...);
```



Classe declarada



Classe instanciada

Classe declarada vs classe instanciada

A classe declarada pode ser diferente da classe instanciada se:

Classe declarada é pai da classe instanciada

```
Agendamento a1 = new AgendamentoDiaInteiro(...);
```



Classe declarada



Classe instanciada

Classe declarada vs classe instanciada

A classe declarada pode ser diferente da classe instanciada se:

Classe declarada é pai da classe instanciada

```
Agendamento a1 = new AgendamentoDiaInteiro(...);
```

```
String a1 = new AgendamentoDiaInteiro(...);
```

Correcto!



Errado!



Polimorfismo

Polimorfismo é a capacidade de um objecto de uma certa classe declarada poder assumir diferentes classes instanciadas

Agendamento a;

```
// 'a' é um agendamento de dia inteiro  
a = new AgendamentoDiaInteiro("Férias", new Date());
```

```
// afinal 'a' é um agendamento para uma certa hora  
a = new AgendamentoHora("Aulas", new Date(), 16, 18);
```

```
// ops, afinal 'a' é um agendamento de dia inteiro  
a = new AgendamentoDiaInteiro("Passeio", new Date());
```

Polimorfismo

Polimorfismo é a capacidade de um objecto de uma certa classe declarada poder assumir diferentes classes instanciadas

```
Agendamento a1 = new  
a1.mostraNoEcran();
```

```
Agendamento a2 = new  
a2.mostraNoEcran();
```

- **a1** pode ser um AgendamentoDiaInteiro ou um AgendamentoHora
- **a2** pode ser um AgendamentoDiaInteiro ou um AgendamentoHora
- **Mas tudo compila e funciona!!!**

Polimorfismo

Porque é que isto é útil???

Polimorfismo

Porque é que isto é útil???



```
List<AgendamentoDiaInteiro> agendamentosDiaInteiro = new ArrayList<>();  
List<AgendamentoHora> agendamentosHora = new ArrayList<>();
```

Polimorfismo

Porque é que isto é útil???



```
List<AgendamentoDiaInteiro> agendamentosDiaInteiro = new ArrayList<>();  
List<AgendamentoHora> agendamentosHora = new ArrayList<>();
```

```
List<Agendamento> agendamentos = new ArrayList<>();
```

A agenda é uma lista de agendamentos, não
interessa saber qual o seu tipo

Polimorfismo

```
List<Agendamento> agendamentos = new ArrayList<>();
```

```
agendamentos.add(new AgendamentoDiaInteiro(...))
```

```
agendamentos.add(new AgendamentoHora(...))
```

```
agendamentos.add(new AgendamentoDiaInteiro(...))
```

— Classe declarada

— Classe instanciada

```
for (Agendamento agendamento : agendamentos) {  
    agendamento.mostraNoEcran();  
}
```

Podemos percorrer os agendamentos da agenda e chamar métodos que estejam na superclasse (eventualmente redefinidos nas subclasses)

Polimorfismo

```
List<Agendamento> agendamentos = new ArrayList<>();

agendamentos.add(new AgendamentoDiaInteiro(...))
agendamentos.add(new AgendamentoHora(...))
agendamentos.add(new AgendamentoDiaInteiro(...))

for (Agendamento agendamento : agendamentos) {
    → if (agendamento instanceof AgendamentoDiaInteiro) {
        ...
    }
    → if (agendamento instanceof AgendamentoHora) {
        ...
    }
}
```

Funciona mas não é a maneira correcta
(orientada a objetos)!!

Se estamos a usar if's para perceber qual é a subclasse
então estamos a fazer mal

Polimorfismo - Sumário

Caso mais comum: Percorrer uma lista de objetos de classes diferentes para fazer “algo” com esses objetos

1. Todos os objetos têm que herdar da mesma classe
2. Essa classe tem que declarar o método que permite fazer “algo” (pode ser abstrato ou ter uma implementação por omissão)
3. Cada classe filha redefine esse “algo” como bem entender

Tratamento de erros



RECENTLY IN THE OPERATING ROOM

Tratamento de erros

Tratamento de erros

```
public class Aplicacao {  
    public static void main(String[] args) {  
  
        ContaBancaria conta = new ContaBancaria();  
        boolean sucesso;  
  
        // depositar 100 euros  
        conta.deposita(100);  
  
        // retirar 50 euros  
        sucesso = conta.levanta(50);  
  
        // tenta retirar 100 euros mas não consegue  
        sucesso = conta.levanta(100);  
        if (!sucesso) {  
            System.out.println("Não tem dinheiro suficiente para esta operação");  
        }  
    }  
}
```

Tratamento de erros

O método `levanta()` pode falhar em duas situações:

```
public boolean levanta(int valor) {  
    if (valor > saldo) {  
        return false;  
    }  
  
    if (valor < 0) {  
        return false;  
    }  
  
    saldo -= valor;  
  
    return true;  
}
```

← Saldo insuficiente

← Valor inválido

Tratamento de erros

```
public boolean levanta(int valor) {  
    if (valor > saldo) {  
        return false;  
    }  
  
    if (valor < 0) {  
        return false;  
    }  
  
    saldo -= valor;  
  
    return true;  
}
```

Ambas as situações retornam false

```
public static void main (...) {  
    Conta c1 = new Conta();  
  
    int quantidade = ... // pede quantidade ao utilizador  
  
    boolean success = conta.levanta(quantidade);  
    if (!success) {  
        System.out.println("..."); // << que erro mostro aqui?  
    }  
}
```

Mas como é que distingo
um erro do outro?

Hipótese 1 - Códigos de erro

```
public int levanta(int valor) {  
    if (valor > saldo) {  
        return 1; // saldo insuficiente  
    }  
  
    if (valor < 0) {  
        return 2; // quantidade negativa  
    }  
  
    saldo -= valor;  
  
    return 0;  
}
```

A função passa a retornar um inteiro com um código de erro:

- 0 - sucesso
- 1 - saldo insuficiente
- 2 - quantidade negativa

```
public static void main (...) {  
    Conta c1 = new Conta();  
  
    int quantidade = ... // pede quantidade ao utilizador  
  
    int resultado = conta.levanta(quantidade);  
    if (resultado == 1) {  
        System.out.println("Saldo insuficiente...");  
    } else if (resultado == 2) {  
        System.out.println("Quantidade negativa...");  
    }  
}
```

Hipótese 1 - Códigos de erro

```
public int getJuros() {  
    if (saldo < 100) {  
        // não está em condições de receber juros  
        return ????  
    } else {  
        return saldo * taxaJuro;  
    }  
}
```

Problema: E se o método precisa de retornar alguma coisa além do código de erro?

Hipótese 2 - Exceptions

```
public int getJuros() {  
    if (saldo < 100) {  
        throw new JurosException("Não está em condições de receber juros");  
    } else {  
        return saldo * taxaJuro;  
    }  
}
```

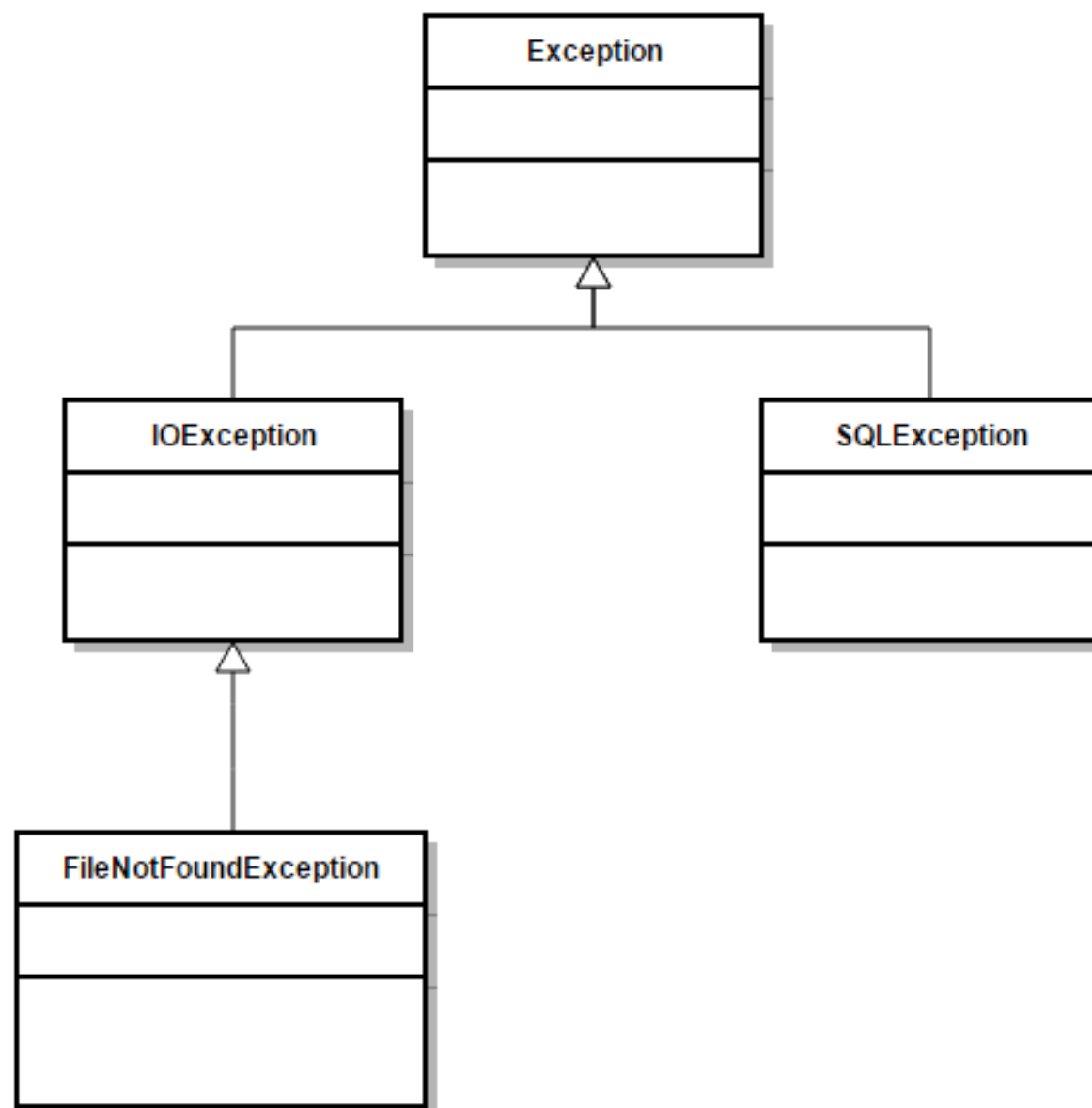
Exceptions

Uma exception é um objeto Java “especial” que é criado no momento em que ocorre um erro

Para o Java perceber que este objecto é especial, ele tem que ser uma instância da classe Exception ou de uma subclasse de Exception

```
Exception erro1 = new Exception();  
FileNotFoundException erro2 = new FileNotFoundException();
```

Exceptions



Podemos criar as nossas próprias classes `Exception` ou usar as já existentes.

Em qualquer caso, serão sempre subclasses (diretas ou indiretas) da classe `Exception`

Exceptions

Um método pode então retornar valores (em caso de sucesso) ou Exceptions (em caso de insucesso).

```
public int getJuros() throws NaoPodeReceberJurosException {  
    if (saldo < 100) {  
        throw new NaoPodeReceberJurosException();  
    } else {  
        return saldo * taxaJuro;  
    }  
}
```

Nota: Isto assume que criei a classe NaoPodeReceberJurosException (a herdar de Exception)

Exceptions

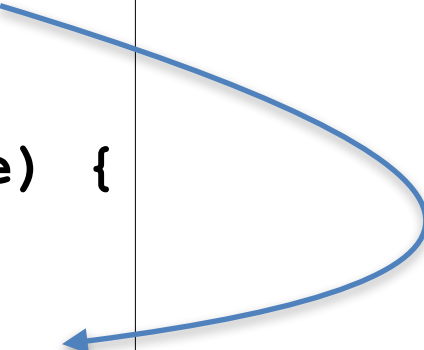
Como é que valido se um certo método teve um erro (isto é lançou (*throws*) uma Exception)?

Invoco/executo o método dentro de um bloco try/catch

```
try {  
    ...  
    int juro = conta.getJuro();  
    System.out.println("Recebeu " + juro);  
} catch (NaoPodeReceberJuroException e) {  
    System.out.println("Não está em condições de receber juro");  
}
```


Exceptions

```
try {  
    ...  
} catch (XXXException e) {  
    ...  
}
```



Qualquer método que lance uma exception da classe XXXException salta para fora do bloco do try e entra no catch respectivo

Se nunca fôr lançada nenhuma exception, o bloco do try é executado até ao fim e o bloco do catch não é executado

Exceptions

Tratamento de erros tradicional

```
int resultado = leFicheiro();

if (resultado == OK) {
    System.out.println("ficheiro ok");
} else if (resultado == FICHEIRO_NAO_ENCONTRADO) {
    System.out.println("ficheiro não encontrado");
}
```

Tratamento de erros com Exceptions

```
try {
    leFicheiro();
    System.out.println("ficheiro ok");
} catch (FileNotFoundException e) {
    System.out.println("ficheiro não encontrado");
}
```

Exceptions

Sou obrigado a apanhar todas as exception?

Não! Posso decidir não apanhar, mas nesse caso tenho que declarar que poderei lançar essa Exception

```
public void obterInformacao() throws FileNotFoundException {  
    FileReader file = new FileReader("ficheiroQueNaoExiste.txt");  
    // ... só chega aqui se o ficheiro existir  
}
```

← Pode lançar uma `FileNotFoundException`, mas decidi não a apanhar, por isso tenho que declarar que vou lançá-la

Exceptions

try/catch/finally

```
try {  
    FileReader file = new FileReader("ficheiroQueNaoExiste.txt");  
    System.out.println("O ficheiro existe");  
} catch (Exception e) {  
    System.out.println("O ficheiro não existe");  
} finally {  
    System.out.println("Isto é escrito em qualquer caso no final");  
}
```

Exceptions

try/catch/finally

```
try {  
    // obtém ligação à Base de Dados (BD)  
    // faz coisas na BD  
} catch (Exception e) {  
    // mostra erro ao utilizador  
} finally {  
    // aconteça o que acontecer, liberta a ligação à BD  
}
```

finally serve para limpar coisas, libertar recursos, etc.

```

class Exemplo {

    public void doSomething(int x) {
        try {
            if (x < 5) {
                System.out.println("1");
                throw new Exception("x menor que 5");
            } else {
                doSomething2();
                System.out.println("2");
            }
        } catch (Exception e) {
            System.out.println("Apanhei exception " + e.getMessage());
        } finally {
            System.out.println("3");
        }
    }

    public void doSomething2() throws FileNotFoundException {
        try {
            throw new Exception("ocorreu um erro");
        } catch (Exception e) {
            throw new FileNotFoundException(e.getMessage());
        }
    }
}

public class App {
    public static void main(String[] args) throws Exception {
        Exemplo exemplo = new Exemplo();
        exemplo.doSomething(3);
        exemplo.doSomething(7);
    }
}

```

Exercício

Qual o output deste programa?

Enviar por teams para
p4997

```

class Exemplo {

    public void doSomething(int x) {
        try {
            2 if (x < 5) {
                3 System.out.println("1");
                4 throw new Exception("x menor que 5");
            } else {
                doSomething2();
                System.out.println("2");
            }
        } catch (Exception e) {
            5 System.out.println("Apanhei exception " + e.getMessage());
        } finally {
            6 System.out.println("3");
        }
    }

    public void doSomething2() throws FileNotFoundException {
        try {
            throw new Exception("ocorreu um erro");
        } catch (Exception e) {
            throw new FileNotFoundException(e.getMessage());
        }
    }

}

public class App {
    public static void main(String[] args) throws Exception {
        Exemplo exemplo = new Exemplo();
        exemplo.doSomething(3); 1
        exemplo.doSomething(7);
    }
}

```

Output:

1
Apanhei exception x menor que 5
3

```

class Exemplo {

    public void doSomething(int x) {
        try {
            2 if (x < 5) {
                System.out.println("1");
                throw new Exception("x menor que 5");
            } else {
                3 doSomething2();
                System.out.println("2");
            }
        } catch (Exception e) {
            6 System.out.println("Apanhei exception " + e.getMessage());
        } finally {
            7 System.out.println("3");
        }
    }

    public void doSomething2() throws FileNotFoundException {
        try {
            4 throw new Exception("ocorreu um erro");
        } catch (Exception e) {
            5 throw new FileNotFoundException(e.getMessage());
        }
    }

}

public class App {
    public static void main(String[] args) throws Exception {
        Exemplo exemplo = new Exemplo();
        exemplo.doSomething(3);
        exemplo.doSomething(7); 1
    }
}

```

Output:

```

1
Apanhei exception x menor que 5
3
Apanhei exception ocorreu um erro
3

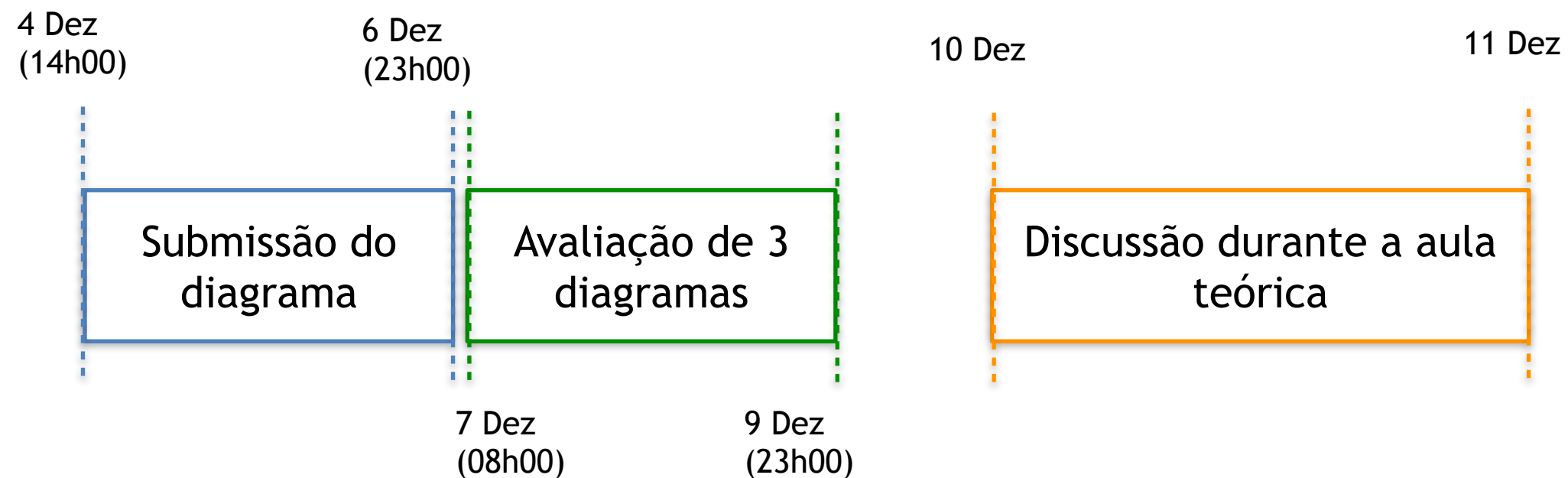
```


Testes unitários e cobertura

(vídeo no Moodle)

TPC teórico 3

Submissão de um diagrama de classes
e avaliações de 3 diagramas



TPC teórico 3

Desenhe o diagrama de classes de uma Aparelhagem constituída por módulos independentes
(deverá usar herança, pelo menos uma classe deverá ser abstracta e usar os mecanismos de visibilidade, ver cábula)



Legenda para quem não conhece aparelhagens



Também pode incluir: Equalizador, Gira-discos, ...
(investiguem outros, perguntem aos vossos pais)