

# Programação Orientada a Objetos

Prof. Paulo Henrique Pisani

<http://professor.ufabc.edu.br/~paulo.pisani/>

# Tópicos

- Exceções (hierarquia)
- Checked vs Unchecked
- Finally
- Try with resources

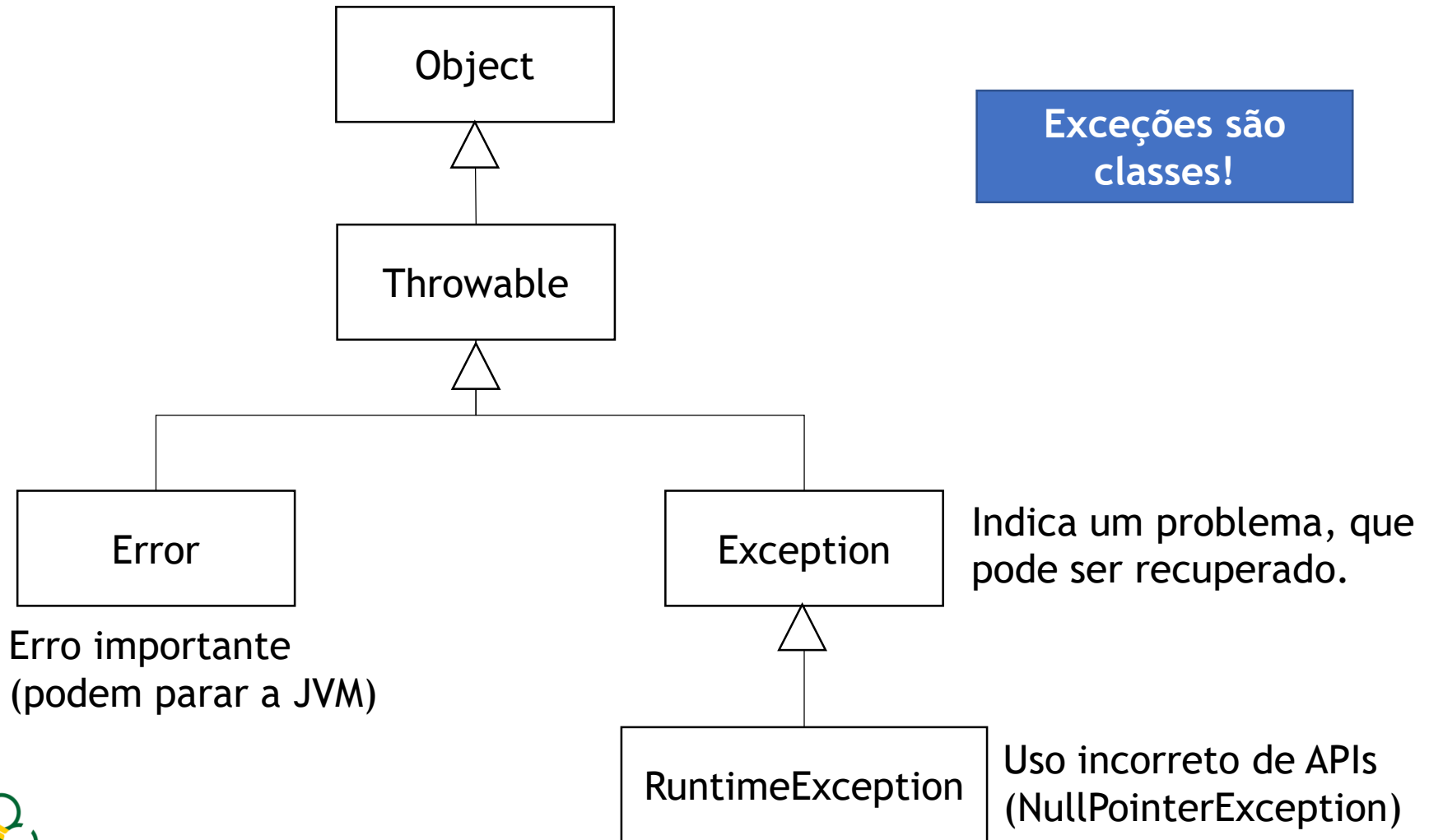
# Exceções (hierarquia)

# Exceções, lembrando...

- **Exceção** (evento excepcional): evento que **quebra o fluxo normal** do programa;
- **Rever final da Aula 5 (Exceções)**

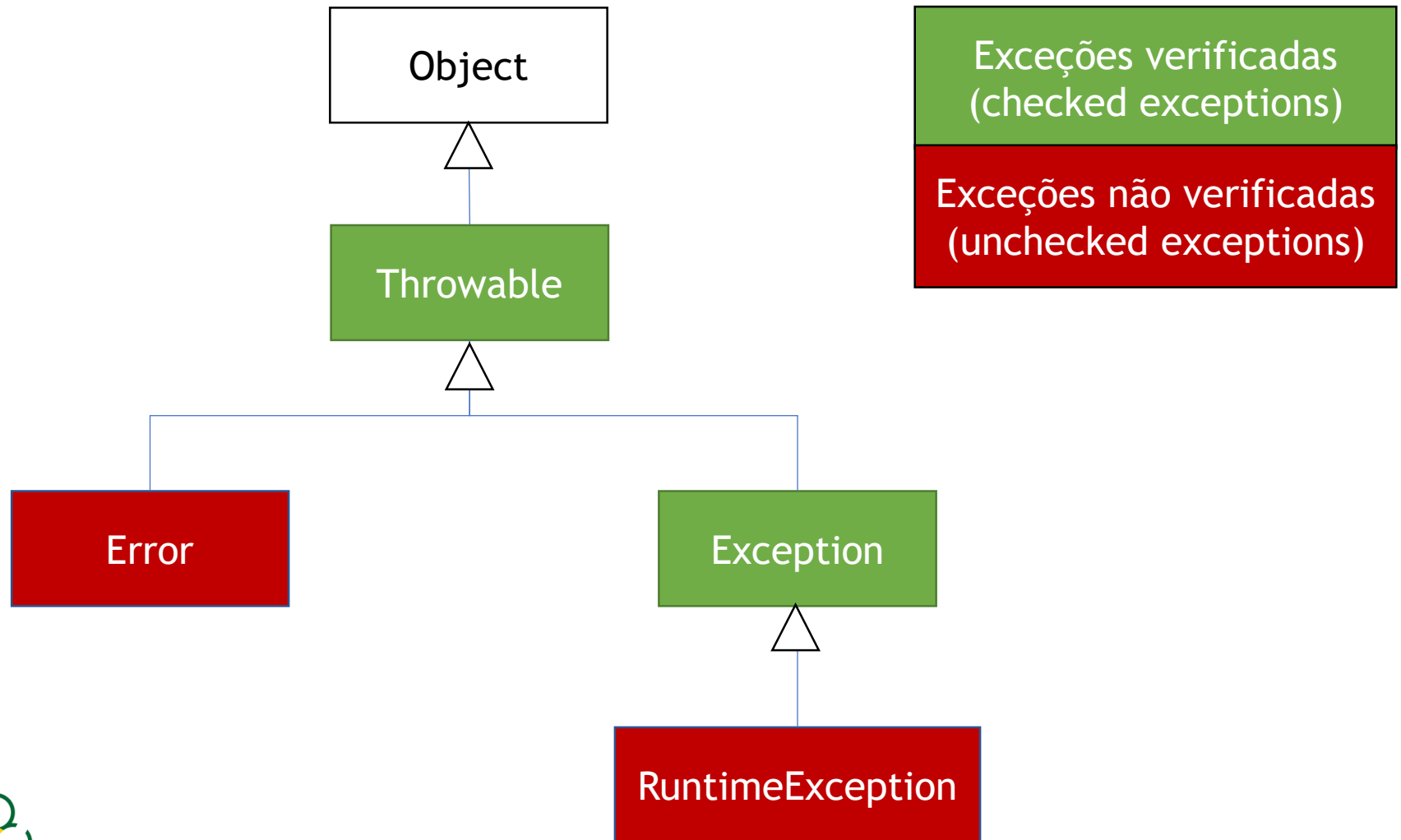
```
public void adicionarPergunta(Pergunta novaPergunta) throws Exception {  
    if (indiceAtual + 1 >= perguntas.length) {  
        //System.out.println("Limite de perguntas atingido!");  
        //return;  
        throw new Exception("Limite de perguntas atingido!");  
    }  
    indiceAtual++;  
    perguntas[indiceAtual] = novaPergunta;  
}
```

# Hierarquia de exceções



Exceções são  
classes!

# Hierarquia de exceções



# Exceções checked vs unchecked

- **Checked** são exceções que devem ser capturadas e tratadas de alguma forma;
- E caso não sejam, isso deve ser informado pelo método.

```
acesso tipoRetorno nomeMetodo(<parametros>) throws TipoExcecao {  
  
}
```

Informa que o método  
pode lançar a exceção  
especificada.

# Exceções checked vs unchecked

- **Unchecked** são exceções que devem ser capturadas e tratadas de alguma forma;
- E caso não sejam, isso deve ser informado pelo método.



```
acesso tipoRetorno nomeMetodo(<parametros>) {
```

Não precisa de **throws**

```
}
```

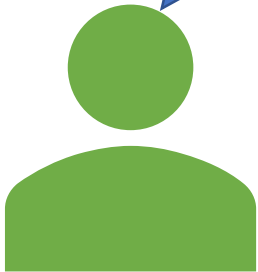


# Exceções checked vs unchecked

- As exceções são uma parte fundamental de um método:
  - Quem utiliza um método deve saber o tipo de retorno, seus parâmetros e quais exceções ele pode lançar;
  - Assim, o usuário do método pode realizar o tratamento das exceções.
- Devido a isso, recomenda-se o uso de exceções checked.

# Exceções checked vs unchecked

Tá, mas pra que fizeram Error e RuntimeException?



# Exceções checked vs unchecked

Tá, mas pra que fizeram Error e RuntimeException?

- Error representa um erro importante que normalmente não pode ser tratado;
- RuntimeException representa um erro de programação, não é papel de quem usa o método tratar um erro de programação de um método (e.g. divisão por zero).



# Exceções checked

# Exceções checked

- São exceções que devem ser capturadas (**catch**) e informadas (**throws**);
- É o tipo de exceção que usamos nas últimas aulas.

# Vamos criar uma exceção

## Checked

- Nos códigos da Aula 9, tínhamos um método que lançava exceção:

```
public void adicionarAula(Aula novaAula) throws Exception {  
    for (int i = 0; i < listaDeAulas.length; i++)  
        if (listaDeAulas[i] == null) {  
            listaDeAulas[i] = novaAula;  
            return;  
        }  
  
    throw new Exception("Tem muita aula!");  
}
```

# Vamos criar uma exceção

## Checked

- Nos códigos da Aula 9, tínhamos um método que lançava exceção:

```
public void adicionarAula(Aula novaAula) throws Exception {  
    for (int i = 0; i < listaDeAulas.length; i++)  
        if (listaDeAulas[i] == null) {  
            listaDeAulas[i] = novaAula;  
            return;  
        }  
  
    throw new Exception("Tem muita aula!");  
}
```

Vamos criar uma exceção  
para esse tipo de erro.

# Vamos criar uma exceção

## Checked

- Para criar um tipo de Exceção basta estender a classe Exception:

```
package plano;  
  
public class TemMuitaAula extends Exception {  
  
}
```



# Vamos criar uma exceção

## Checked

- Para criar um tipo de Exceção basta estender a classe Exception:

```
package plano;  
  
public class TemMuitaAula extends Exception {  
    public TemMuitaAula() {  
        super("Tem muita aula!");  
    }  
}
```

Podemos melhorar um pouco e passar a String do erro para o constructor de Exception -> usando `super(String)`

# Vamos criar uma exceção

## Checked

- Agora Podemos atualizar o método para lançar nossa nova exceção: **TemMuitaAula**

```
public void adicionarAula(Aula novaAula) throws Exception {  
    for (int i = 0; i < listaDeAulas.length; i++)  
        if (listaDeAulas[i] == null) {  
            listaDeAulas[i] = novaAula;  
            return;  
        }  
  
    throw new TemMuitaAula() ;  
}
```

# Throws e catch pode capturar a própria classe ou superclasses da exceção

- Agora Podemos atualizar o método para lançar nossa nova exceção: **TemMuitaAula**

```
public void adicionarAula(Aula novaAula) throws Exception {  
    for (int i = 0; i < listaDeAulas.length; i++)  
        if (listaDeAulas[i] == null) {  
            listaDeAulas[i] = novaAula;  
            return;  
        }  
  
    throw new TemMuitaAula() ;  
}
```

Aqui podemos deixar superclasses das exceções lançadas.

# Throws e catch pode capturar a própria classe ou superclasses da exceção

- Agora Podemos atualizar o método para lançar nossa nova exceção: **TemMuitaAula**

```
public void adicionarAula(Aula novaAula) throws TemMuitaAula {  
    for (int i = 0; i < listaDeAulas.length; i++)  
        if (listaDeAulas[i] == null) {  
            listaDeAulas[i] = novaAula;  
            return;  
        }  
  
    throw new TemMuitaAula();  
}
```

Ou especificamos a classe exata.

# Vamos colocar outra exceção

```
package plano;  
  
public class RevisaoNaoPermitida extends Exception {  
  
    public RevisaoNaoPermitida() {  
        super("Nao pode ter aula de revisao!");  
    }  
  
}
```

```
public void adicionarAula(Aula novaAula)  
    throws TemMuitaAula, RevisaoNaoPermitida {  
    if (novaAula instanceof AulaRevisao)  
        throw new RevisaoNaoPermitida();  
  
    for (int i = 0; i < listaDeAulas.length; i++)  
        if (listaDeAulas[i] == null) {  
            listaDeAulas[i] = novaAula;  
            return;  
        }  
  
    throw new TemMuitaAula();  
}
```

# Podemos tratar cada tipo de exceção separadamente

- Para isso, colocamos um bloco `catch` para cada exceção.

```
try {
    PlanoDeAulas plano = new PlanoDeAulas(1);

    Aula a1 = new Teorica("Introducao");
    plano.adicionarAula(a1);

    Prova p1 = new Prova(2, "Prova Facil");
    plano.adicionarAula(p1);

    plano.imprimir();
} catch (RevisaoNaoPermitida e) {
    System.out.println("Nao pode ter revisao: " + e);
} catch (TemMuitaAula e) {
    System.out.println("Aulas demais! " + e);
}
```

# Ordem de captura

- As exceções mais específicas (subclasses) devem ser capturadas primeiro!

```
try {  
    PlanoDeAulas plano = new PlanoDeAulas(1);  
  
    Aula a1 = new Teorica("Introducao");  
    plano.adicionarAula(a1);  
  
    Prova p1 = new Prova(2, "Prova Facil");  
    plano.adicionarAula(p1);  
  
    plano.imprimir();  
} catch (Exception e) {  
    System.out.println("Erro: " + e);  
} catch (RevisaoNaoPermitida e) {  
    System.out.println("Nao pode ter revisao: " + e);  
} catch (TemMuitaAula e) {  
    System.out.println("Erro aulas demais: " + e);  
}
```

# Ordem de captura

- As exceções mais específicas (subclasses) devem ser capturadas primeiro!

```
Principal.java:21: error: exception
RevisaoNaoPermitida has already been caught
                } catch (RevisaoNaoPermitida e) {
                    ^
Principal.java:23: error: exception TemMuitaAula
has already been caught
                } catch (TemMuitaAula e) {
                    ^
2 errors
```



# Ordem de captura

- As exceções mais específicas (subclasses) devem ser capturadas primeiro!

```
try {  
    PlanoDeAulas plano = new PlanoDeAulas(1);  
  
    Aula a1 = new Teorica("Introducao");  
    plano.adicionarAula(a1);  
  
    Prova p1 = new Prova(2, "Prova Facil");  
    plano.adicionarAula(p1);  
  
    plano.imprimir();  
} catch (RevisaoNaoPermitida e) {  
    System.out.println("Nao pode ter revisao: " + e);  
} catch (TemMuitaAula e) {  
    System.out.println("Erro aulas demais: " + e);  
} catch (Exception e) {  
    System.out.println("Erro: " + e);  
}
```

Ok, agora  
Exception (a  
superclasse)  
é capturada  
depois das  
subclasses.

# Exceções unchecked

# Exceções unchecked

- São exceções que **não** precisam ser capturadas (**catch**) e informadas (**throws**);
- Veremos exemplos a seguir...

# Vamos criar uma exceção **Unchecked**

- Para criar uma exceção unchecked, basta estender **Error** ou **RuntimeException**:
  - Como será um erro recuperável, vamos estender **RuntimeException**.

```
package plano;  
  
public class AulaNula extends RuntimeException {  
    public AulaNula() {  
        super("Foi passada uma aula nula!");  
    }  
}
```

# Veja que não somos obrigados a informar exceções unchecked

- O Código a seguir compila, mesmo sem a exceção **AulaNula** estar no **throws**

```
public void adicionarAula(Aula novaAula)
    throws TemMuitaAula, RevisaoNaoPermitida {
    if (novaAula == null)
        throw new AulaNula();

    if (novaAula instanceof AulaRevisao)
        throw new RevisaoNaoPermitida();

    for (int i = 0; i < listaDeAulas.length; i++)
        if (listaDeAulas[i] == null) {
            listaDeAulas[i] = novaAula;
            return;
        }

    throw new TemMuitaAula();
}
```

Exceção unchecked  
(estende RuntimeException)

# Veja que também não somos obrigados a tratar exceções unchecked

- O Código a seguir compila, mesmo sem capturar a exceção **RevisaoNaoPermitida**

```
try {  
    PlanoDeAulas plano = new PlanoDeAulas(2);  
  
    Aula a1 = new Teorica("Revisao");  
    plano.adicionarAula(a1);  
  
    Prova p1 = new Prova(2, "Prova Facil");  
    plano.adicionarAula(p1);  
  
    plano.imprimir();  
} catch (RevisaoNaoPermitida e) {  
    System.out.println("Nao pode ter revisao: " + e);  
} catch (TemMuitaAula e) {  
    System.out.println("Aulas demais! " + e);  
}
```

Pode lançar exceção AulaNula

# Mas podemos tratar essas exceções mesmo assim...

```
try {  
    PlanoDeAulas plano = new PlanoDeAulas(1);  
  
    Aula a1 = new Teorica("Revisao");  
    plano.adicionarAula(a1);  
  
    Prova p1 = new Prova(2, "Prova Facil");  
    plano.adicionarAula(p1);  
  
    plano.imprimir();  
} catch (RevisaoNaoPermitida e) {  
    System.out.println("Nao pode ter revisao: " + e);  
} catch (TemMuitaAula e) {  
    System.out.println("Aulas demais! " + e);  
} catch (AulaNula e) {  
    System.out.println("Aula nula! " + e);  
}
```

# Estendendo exceções (ainda mais)



# Estendendo exceções...

- Como vimos, **exceções são classes**;
- Portanto, podemos adicionar atributos e métodos!
  - Assim, quem capturar a exceção pode tratar melhor as exceções lançadas.

# Vamos adicionar informações adicionais na exceção TemMuitaAula

```
package plano;

public class TemMuitaAula extends Exception {
    private int limite;
    private Aula aula;

    public TemMuitaAula(int limite, Aula aula) {
        super("Tem muita aula!");
        this.limite = limite;
        this.aula = aula;
    }

    public int getLimite() {
        return this.limite;
    }
    public Aula getAula() {
        return this.aula;
    }
}
```

# Vamos adicionar informações adicionais na exceção TemMuitaAula

```
public void adicionarAula(Aula novaAula) throws TemMuitaAula, RevisaoNaoPermitida {  
    if (novaAula == null)  
        throw new AulaNula();  
  
    if (novaAula instanceof AulaRevisao)  
        throw new RevisaoNaoPermitida();  
  
    for (int i = 0; i < listaDeAulas.length; i++)  
        if (listaDeAulas[i] == null) {  
            listaDeAulas[i] = novaAula;  
            return;  
        }  
  
    throw new TemMuitaAula(listaDeAulas.length, novaAula);  
}
```

Agora passamos os parâmetros adicionais ao construtor de TemMuitaAula.

# Vamos adicionar informações adicionais na exceção **TemMuitaAula**

```
public void adicionarAula(Aula novaAula) throws TemMuitaAula, RevisaoNaoPermitida {  
    if (novaAula == null)  
        throw new AulaNula();  
  
    if (novaAula instanceof AulaRevisao)  
        throw new RevisaoNaoPermitida();  
  
    for (int i = 0; i < listaDeAulas.length; i++)  
        if (listaDeAulas[i] == null) {  
            listaDeAulas[i] = novaAula;  
            return;  
        }  
}
```

```
    TemMuitaAula excecao = new TemMuitaAula(listaDeAulas.length, novaAula);  
    throw excecao;
```

Exceção é uma classe. Portanto, podemos instanciá-la e guardar em uma variável. Depois lançamos com **throw**.

# Vamos adicionar informações adicionais na exceção TemMuitaAula

```
try {
    PlanoDeAulas plano = new PlanoDeAulas(1);

    Aula a1 = new Teorica("Introducao");
    plano.adicionarAula(a1);

    Prova p1 = new Prova(2, "Prova Facil");
    plano.adicionarAula(p1);

    plano.imprimir();
} catch (RevisaoNaoPermitida e) {
    System.out.println("Revisao nao permitida! " + e);
} catch (TemMuitaAula e) {
    System.out.println("Aulas demais! " + e);
    System.out.println("O limite de aulas eh " + e.getLimite());
    System.out.println("A aula que passou do limite foi "
        + e.getAula().getDescricao());
} catch (AulaNula e) {
    System.out.println("Aula nula! " + e);
}
```

# Finally

# Finally

- É um bloco executado quando o bloco **try** termina.

```
try {  
    System.out.println("Inicio do programa!");  
    int[] vetor = new int[5];  
    System.out.println(vetor[5]);  
  
    System.out.println("Apos indice invalido!");  
} finally {  
    System.out.println("Cheguei no final!");  
}
```

# Finally

- O bloco **finally** não captura exceções, mas é sempre executado, independentemente de ocorrer uma exceção no bloco **try**;
  - Pode ser usado garantir que um recurso será finalizado (e.g. ao abrir uma conexão TCP, o bloco **finally** pode conter uma chamada para encerrar a conexão mesmo em caso de erros).



# Finally

- Finally pode ser usado para garantir a execução de parte do código mesmo quando **return** ou **break** é usado.

```
public static void verificaDadosNoBD(int numero) {  
    System.out.println("Abriu a conexão.");  
    try {  
        System.out.println("Testa número.");  
        if (numero == 8)  
            return;  
        else  
            System.out.println("Número não é 8.");  
  
    } finally {  
        System.out.println("Fechou a conexão.");  
    }  
}
```

# Try with resources

# Try with resources

- Muitas vezes usamos recursos que precisam realizar algum procedimento para encerramento;
  - E precisamos garantir que esse procedimento seja executado SEMPRE.
  - Uma forma de garantir isso é usando **finally** (como vimos hoje)

```
// Abre recurso  
try {  
    // Faz alguma coisa  
} finally {  
    // Fecha recurso  
}
```

Mesmo se ocorrer uma exceção no bloco try, o recurso será fechado

# Try with resources

- Mas existe uma outra forma de lidar com esse problema em Java: **try with resources**
- Para isso, usamos a seguinte estrutura:

```
// Abre recurso  
try {  
    // Faz alguma coisa  
} finally {  
    // Fecha recurso  
}
```



```
try (/*abre recurso*/) {  
}
```

Recurso será automaticamente fechado ao final do bloco try (mesmo se ocorrer exceção)

# Exemplo: leitura de arquivos

- Para ler arquivos, podemos usar algumas classes do pacote `java.io`:

```
import java.io.BufferedReader;
import java.io.FileReader;

public class Principal {
    public static void main(String[] args) {
        try {
            BufferedReader br = new BufferedReader(new FileReader("arquivo.txt")); Abre arquivo
            System.out.println(br.readLine());
            System.out.println(br.readLine());
            br.close(); Fecha arquivo
        } catch (Exception e) {
            System.out.println("Erro: " + e);
        }
    }
}
```

# Exemplo: leitura de arquivos

- Para ler arquivos, podemos usar algumas classes do pacote `java.io`:

```
import java.io.BufferedReader;
import java.io.FileReader;

public class Principal {
    public static void main(String[] args) {
        try {
            BufferedReader br = new BufferedReader(new FileReader("arquivo.txt"));
            System.out.println(br.readLine());
            System.out.println(br.readLine());
            br.close();
        } catch (Exception e) {
            System.out.println("Erro: " + e);
        }
    }
}
```

Abre arquivo

Fecha arquivo

Se for lançada alguma exceção entre a abertura e o fechamento do arquivo, ele não é fechado!

# Exemplo: leitura de arquivos

- Para ler arquivos, podemos usar algumas classes do pacote `java.io`:

```
import java.io.BufferedReader;
import java.io.FileReader;

public class Principal {
    public static void main(String[] args) {
        try {
            BufferedReader br = new BufferedReader(new FileReader("arquivo.txt")); Abre arquivo
            System.out.println(br.readLine());
            System.out.println(br.readLine());
            int a = 3 / 0;
            br.close(); Fecha arquivo
        } catch (Exception e) {
            System.out.println("Erro: " + e);
        }
    }
}
```

# Exemplo: leitura de arquivos

- Para ler arquivos, podemos usar algumas classes do pacote `java.io`:

```
import java.io.BufferedReader;
import java.io.FileReader;
```

```
public class Principal {
    public static void main(String[] args) {
        try {
            BufferedReader br = new BufferedReader(new FileReader("arquivo.txt"));
            System.out.println(br.readLine());
            System.out.println(br.readLine());
            int a = 3 / 0;
            br.close();
        } catch (Exception e) {
            System.out.println("Erro: " + e);
        }
    }
}
```

Abre arquivo

Exception in thread "main"  
`java.lang.ArithmeticException: / by zero`

Fecha arquivo

Programa não  
fecha o arquivo,  
devido à exceção  
lançada!



# Exemplo: leitura de arquivos

- Agora usando try...finally

```
import java.io.BufferedReader;  
import java.io.FileReader;
```

```
public class Principal {
```

```
    public static void main(String[] args) {
```

```
        try {
```

```
            BufferedReader br = new BufferedReader(new FileReader("arquivo.txt"));
```

```
            try {
```

```
                System.out.println(br.readLine());
```

```
                System.out.println(br.readLine());
```

```
                int a = 3 / 0;
```

```
            } finally {
```

```
                br.close();
```

```
            }
```

```
        } catch (Exception e) {
```

```
            System.out.println("Erro: " + e);
```

```
        }
```

```
    }
```

Abre arquivo

Exception in thread "main"  
java.lang.ArithmeticException: / by zero

Fecha arquivo

Agora o programa  
sempre fecha o  
arquivo!

# Exemplo: leitura de arquivos

- Agora usando try with resources

```
import java.io.BufferedReader;
import java.io.FileReader;

public class Principal {

    public static void main(String[] args) {
        try {
            try (BufferedReader br = new BufferedReader(new FileReader("arquivo.txt"))) {
                System.out.println(br.readLine());
                System.out.println(br.readLine());
                int a = 3 / 0;
            }
        } catch (Exception e) {
            System.out.println("Erro: " + e);
        }
    }
}
```

Abre arquivo

Exception in thread "main"  
java.lang.ArithmeticException: / by zero

Agora o programa sempre fecha o arquivo! O método close() é chamado ao final do bloco try.

Isso ocorre porque abrimos o arquivo dentro do bloco entre parênteses.

# O try with resources funciona para qualquer coisa?

- Quando essa estrutura é usada, o método **close()** é chamado automaticamente ao final do bloco try;
- Try with resources requer que a classe instanciada **implemente a interface **AutoCloseable!****

# Exemplos de classes que implementam AutoCloseable

- `java.io.BufferedReader`
- `java.util.zip.ZipFile`
- `java.sql.Statement`
- `java.net.Socket`
- `java.beans.XMLEncoder`
- etc

# Vamos fazer a nossa classe implementar AutoCloseable!

```
public class Prova implements AutoCloseable {  
    public Prova() {  
        System.out.println("Construtor da prova");  
        System.out.println("Colocar nome em todas as folhas");  
    }  
  
    public void responderQuestao(int numero) throws RuntimeException {  
        System.out.println("Respondendo questao " + numero);  
        if (numero == 2) throw new RuntimeException("Nao sei!");  
    }  
  
    @Override  
    public void close() {  
        System.out.println("Entregar prova e assinar lista de chamada");  
    }  
}
```

# E agora vamos testá-la...

- Primeiro do jeito errado:

```
Prova p1 = new Prova();  
p1.responderQuestao(1);  
p1.responderQuestao(2);  
p1.responderQuestao(3);  
p1.close();
```

O que será impresso?

# E agora vamos testá-la...

- Primeiro do jeito errado:

```
Prova p1 = new Prova();  
p1.responderQuestao(1);  
p1.responderQuestao(2);  
p1.responderQuestao(3);  
p1.close();
```

Veja que close() não foi chamado devido à exceção lançada!

## Saída

```
Construtor da prova  
Colocar nome em todas as folhas  
Respondendo questao 1  
Respondendo questao 2  
Exception in thread "main" java.lang.RuntimeException: Nao sei!  
    at Prova.responderQuestao(Prova.java:10)  
    at Principal.main(Principal.java:35)
```

# E agora vamos testá-la...

- Agora do jeito certo (com finally):

```
Prova p1 = new Prova();  
try {  
    p1.responderQuestao(1);  
    p1.responderQuestao(2);  
    p1.responderQuestao(3);  
} finally {  
    p1.close();  
}
```

O que será impresso?

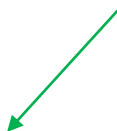


# E agora vamos testá-la...

- Agora do jeito certo (com finally):

```
Prova p1 = new Prova();  
try {  
    p1.responderQuestao(1);  
    p1.responderQuestao(2);  
    p1.responderQuestao(3);  
} finally {  
    p1.close();  
}
```

Agora close() foi  
chamado!



Saída

```
Construtor da prova  
Colocar nome em todas as folhas  
Respondendo questao 1  
Respondendo questao 2  
Entregar prova e assinar lista de chamada  
Exception in thread "main" java.lang.RuntimeException: Nao sei!  
    at Prova.responderQuestao(Prova.java:10)  
    at Principal.main(Principal.java:35)
```

# E agora vamos testá-la...

- Agora do jeito certo (com try with resources):

```
try (Prova p1 = new Prova()) {  
    p1.responderQuestao(1);  
    p1.responderQuestao(2);  
    p1.responderQuestao(3);  
}
```


O que será impresso?

# E agora vamos testá-la...

- Agora do jeito certo (com try with resources):

```
try (Prova p1 = new Prova()) {  
    p1.responderQuestao(1);  
    p1.responderQuestao(2);  
    p1.responderQuestao(3);  
}
```

Agora close() foi  
chamado!



Saída

```
Construtor da prova  
Colocar nome em todas as folhas  
Respondendo questao 1  
Respondendo questao 2  
Entregar prova e assinar lista de chamada  
Exception in thread "main" java.lang.RuntimeException: Nao sei!  
    at Prova.responderQuestao(Prova.java:10)  
    at Principal.main(Principal.java:35)
```

E se eu quiser abrir  
vários recursos?



Pode colocar todos no try with  
resources. Mas a ordem de  
fechamento é inversa à de  
abertura!



# Vamos criar outra classe para testar isso...

```
public class Cola implements AutoCloseable {
    private String texto;
    public Cola(String texto) {
        this.texto = texto;
        System.out.println("Cola criada: " + texto);
    }

    public void usar() {
        if (Math.random() < 0.5)
            throw new RuntimeException("Nota = F");
        System.out.println("Professor nao viu...");
    }

    @Override
    public void close() {
        System.out.println("Jogar cola fora: " + texto);
    }
}
```

Abre vários recursos.

```
try (  
    Cola c1 = new Cola("Classe abstrata nao pode ser instanciada");  
    Cola c2 = new Cola("Usar @Override na sobrescrita de metodos");  
    Prova p1 = new Prova();  
) {  
    p1.responderQuestao(1);  
    p1.responderQuestao(2);  
    p1.responderQuestao(3);  
}
```

O que será impresso?

Abre vários recursos.

```
try (  
    Cola c1 = new Cola("Classe abstrata nao pode ser instanciada");  
    Cola c2 = new Cola("Usar @Override na sobrescrita de metodos");  
    Prova p1 = new Prova();  
) {  
    p1.responderQuestao(1);  
    p1.responderQuestao(2);  
    p1.responderQuestao(3);  
}
```

Observe que os recursos são fechados em ordem inversa!

### Saída

```
Cola criada: Classe abstrata nao pode ser instanciada  
Cola criada: Usar @Override na sobrescrita de metodos  
Construtor da prova  
Colocar nome em todas as folhas  
Respondendo questao 1  
Respondendo questao 2  
Entregar prova e assinar lista de chamada  
Jogar cola fora: Usar @Override na sobrescrita de metodos  
Jogar cola fora: Classe abstrata nao pode ser instanciada  
Exception in thread "main" java.lang.RuntimeException: Nao sei!  
    at Prova.responderQuestao(Prova.java:9)  
    at Principal.main(Principal.java:48)
```

catch + finally  
+ try with resources



# catch + finally

```
Cola c1 = new Cola("Classe abstrata nao pode ser instanciada");
Cola c2 = new Cola("Usar @Override na sobrescrita de metodos");
Prova p1 = new Prova();
try {
    p1.responderQuestao(1);
    p1.responderQuestao(2);
    p1.responderQuestao(3);
} catch (Exception e) {
    System.out.println("Excecao capturada");
} finally {
    p1.close();
}
```

O que será impresso?

# catch + finally

```
Cola c1 = new Cola("Classe abstrata nao pode ser instanciada");
Cola c2 = new Cola("Usar @Override na sobrescrita de metodos");
Prova p1 = new Prova();
try {
    p1.responderQuestao(1);
    p1.responderQuestao(2);
    p1.responderQuestao(3);
} catch (Exception e) {
    System.out.println("Excecao capturada");
} finally {
    p1.close();
}
```

O bloco catch é executado antes do finally!

Saída

```
Cola criada: Classe abstrata nao pode ser instanciada
Cola criada: Usar @Override na sobrescrita de metodos
Construtor da prova
Colocar nome em todas as folhas
Respondendo questao 1
Respondendo questao 2
Excecao capturada
Entregar prova e assinar lista de chamada
```

# catch + finally

```
Cola c1 = new Cola("Classe abstrata nao pode ser instanciada");
Cola c2 = new Cola("Usar @Override na sobrescrita de metodos");
Prova p1 = new Prova();
try {
    p1.responderQuestao(1);
    p1.responderQuestao(2);
    p1.responderQuestao(3);
} catch (Exception e) {
    c1.usar();
    c2.usar();
} finally {
    p1.close();
}
```

Neste caso, a saída vai depender do número randômico. Mesmo assim, a ordem permanece a mesma: primeiro catch e depois finally.

# Try with resources + catch

```
try (  
    Cola c1 = new Cola("Classe abstrata nao pode ser instanciada");  
    Cola c2 = new Cola("Usar @Override na sobrescrita de metodos");  
    Prova p1 = new Prova();  
) {  
    p1.responderQuestao(1);  
    p1.responderQuestao(2);  
    p1.responderQuestao(3);  
} catch (Exception e) {  
    c1.usar();  
    c2.usar();  
}
```

# Try with resources + catch

```
try (  
    Cola c1 = new Cola("Classe abstrata nao pode ser instanciada");  
    Cola c2 = new Cola("Usar @Override na sobrescrita de metodos");  
    Prova p1 = new Prova();  
) {  
    p1.responderQuestao(1);  
    p1.responderQuestao(2);  
    p1.responderQuestao(3);  
} catch (Exception e) {  
    c1.usar();  
    c2.usar();  
}
```

```
Principal.java:56: error: cannot find symbol  
                c1.usar();  
                ^  
symbol:   variable c1  
location: class Principal  
Principal.java:57: error: cannot find symbol  
                c2.usar();  
                ^  
symbol:   variable c2  
location: class Principal  
2 errors
```

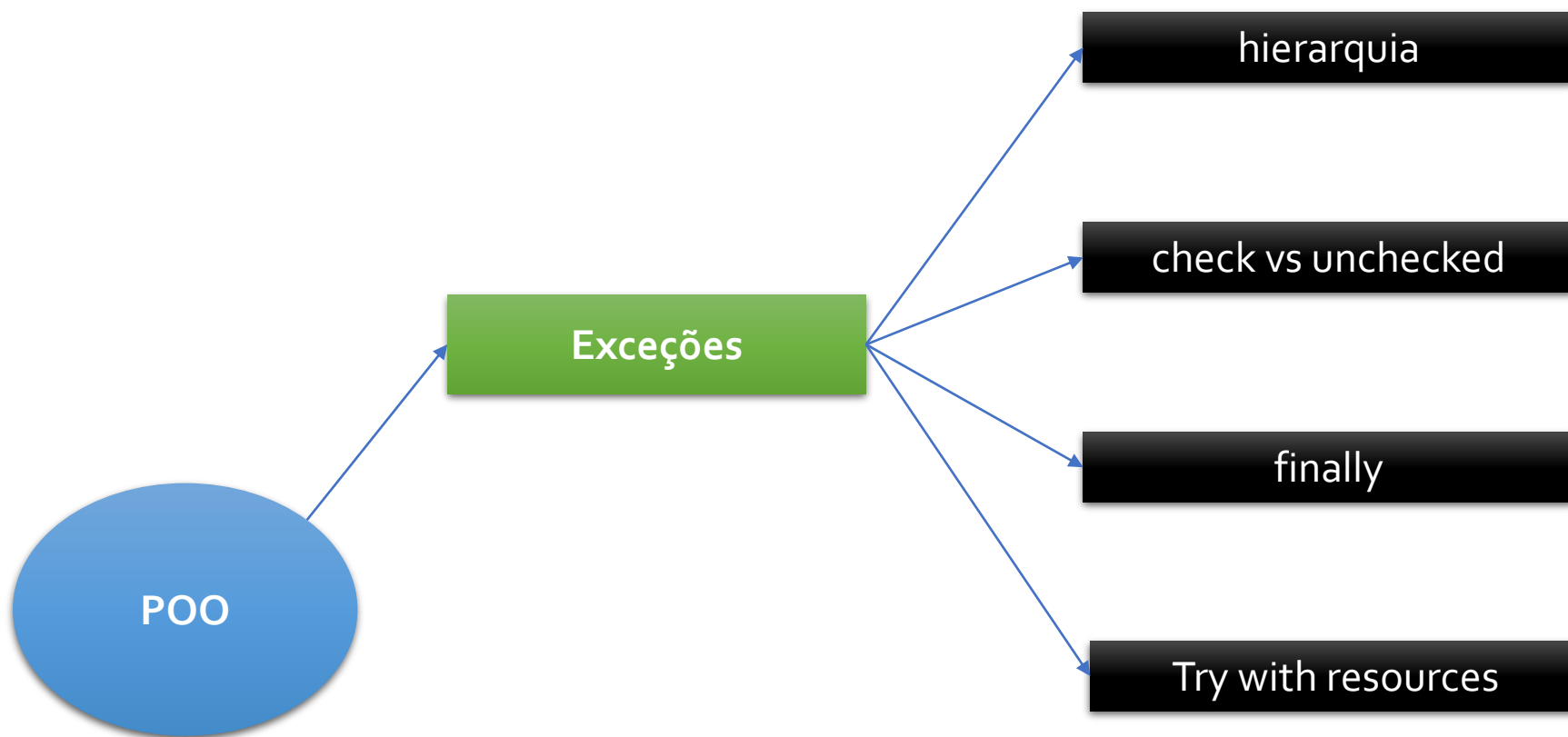
# Try with resources + catch + finally

```
Cola c1 = new Cola("Classe abstrata nao pode ser instanciada");
Cola c2 = new Cola("Usar @Override na sobrescrita de metodos");
try (
    Prova p1 = new Prova();
) {
    p1.responderQuestao(1);
    p1.responderQuestao(2);
    p1.responderQuestao(3);
} catch (Exception e) {
    c1.usar();
    c2.usar();
} finally {
    c2.close();
    c1.close();
}
```

# Exercício 1

- No exercício 1 da Aula 8 (figuras planas), crie classes para exceções que podem ser geradas na validação de figuras:
  - Por exemplo, verificar se é um triângulo válido pode ser uma exceção checked. Informar valores negativos para os lados em qualquer figura pode ser uma exceção unchecked.

# Resumo da aula





# Referências

- Documentação Java:  
<https://docs.oracle.com/javase/8/docs/>

# Referências (projeto pedagógico)

- BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. UML: guia do usuário. Rio de Janeiro, RJ: Campus, 2005.
- GUEDES, G. T. A. UML 2: uma abordagem prática. São Paulo, SP: Novatec, 2009.
- DEITEL, H. M.; DEITEL, P. J. Java: como programar. 6a edição. Porto Alegre, RS: Bookman, 2005.
- BARNES, D. J.; KOLLING, M. Programação orientada a objetos com Java. 4ª edição. São Paulo, SP: Editora Pearson Prentice Hall, 2009.

# Referências (projeto pedagógico)

- FLANAGAN, D. Java: o guia essencial. 5ª edição. Porto Alegre, RS: Bookman, 2006.
- BRUEGGE, B.; DUTOIT, A. H. Object-oriented software engineering: using UML, patterns, and Java. 2ª edição. Upper Saddle River, NJ: Prentice Hall, 2003.
- LARMAN, C. Utilizando UML e padrões: uma introdução à análise e ao projeto orientados a objetos e ao desenvolvimento iterativo. 3ª edição. Porto Alegre, RS: Bookman, 2007.
- FOWLER, M. UML essencial: um breve guia para a linguagem-padrão de modelagem de objetos. 3ª edição. Porto Alegre, RS: Bookman, 2005.