



# Tópicos avançados em Programação

Collections + Tratamento de exceções

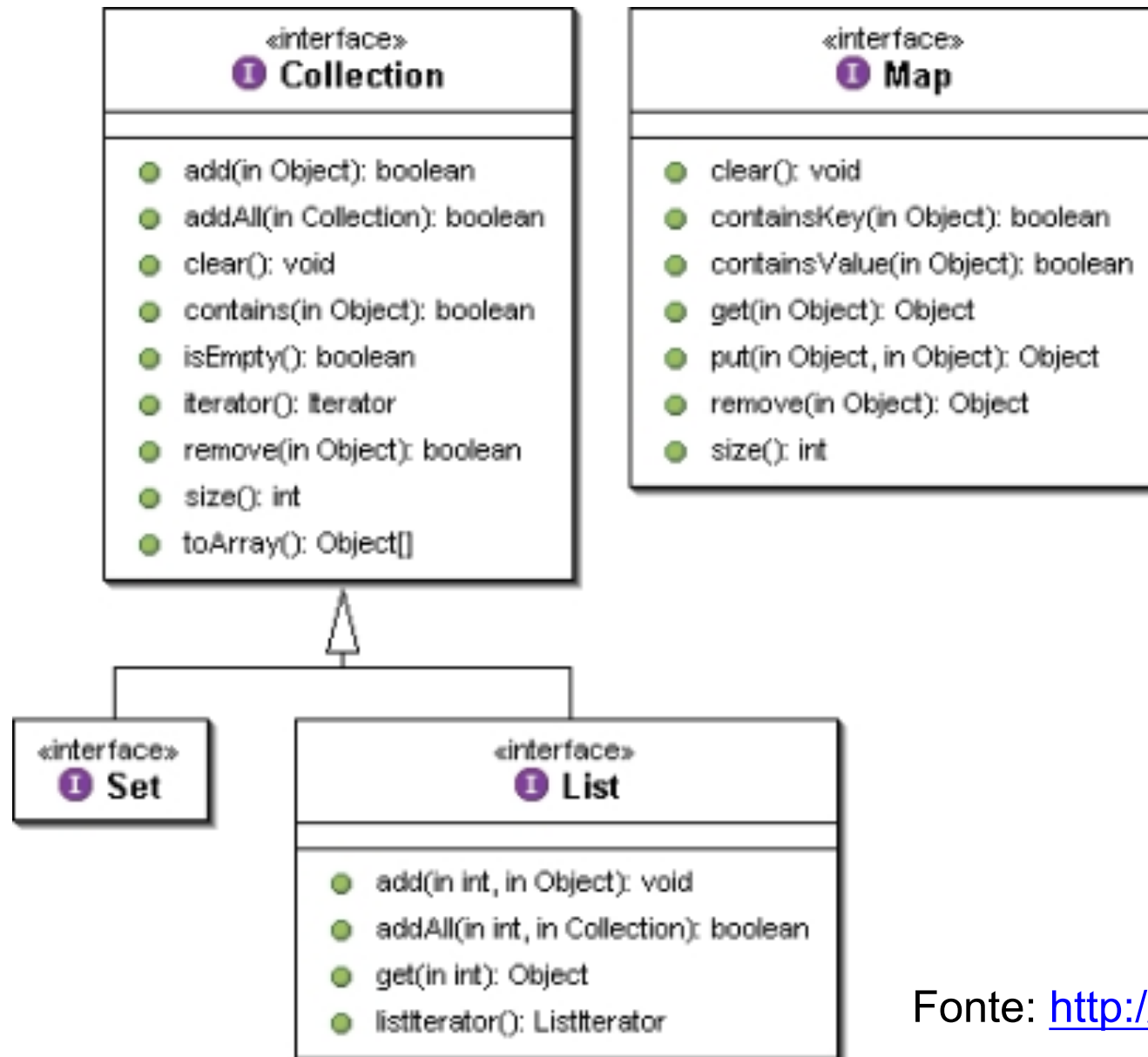
<http://dl.dropbox.com/u/3025380/prog2/aula4.pdf>

[flavio.cecil@unisul.br](mailto:flavio.cecil@unisul.br)

# Agenda

- Java Collection
  - Classe utilitária ***Collections***;
  - Hash e Árvores de objetos complexos;
    - Implementação para conjuntos e mapas.
- Tratamento de exceções
  - O que são exceções;
  - Hierarquia de classes;
  - Exceções checadas e não checadas.

# Relembrando...



Fonte: <http://bit.ly/19QLdeU>

# Classe utilitária para Collection

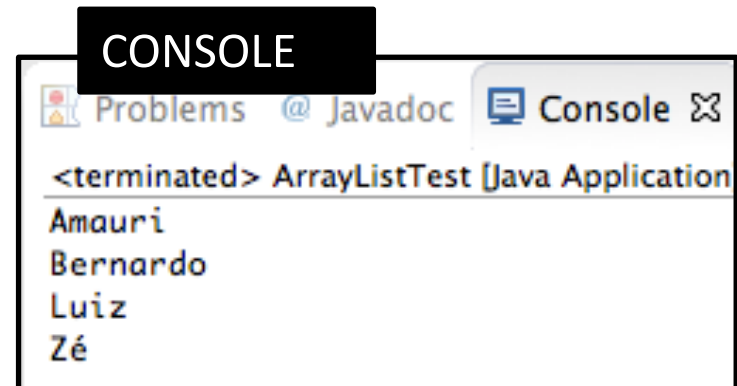
- A classe Collections trás uma série de métodos para auxiliar no uso das estrutura escolhidas.
- Exemplo de ordenação:

```
//Declara e instancia a lista
List<String> lista = new ArrayList<String>();

//Adiciona os valores à lista
lista.add("Zé");
lista.add("Amauri");
lista.add("Luiz");
lista.add("Bernardo");

//Ordena os elementos
Collections.sort(lista);

//Lista todos os valores cadastrados
for(String valor : lista) {
    System.out.println(valor);
}
```



The screenshot shows a console window titled 'CONSOLE' with tabs for 'Problems', '@ Javadoc', and 'Console'. The output text is: '<terminated> ArrayListTest [Java Application]' followed by the names 'Amauri', 'Bernardo', 'Luiz', and 'Zé' on separate lines, indicating the list has been sorted alphabetically.

Utilizando o método estático  
Da classe ***Collections***.

# Collections

- Mais informações sobre os métodos da classe utilitária ***Collections***:

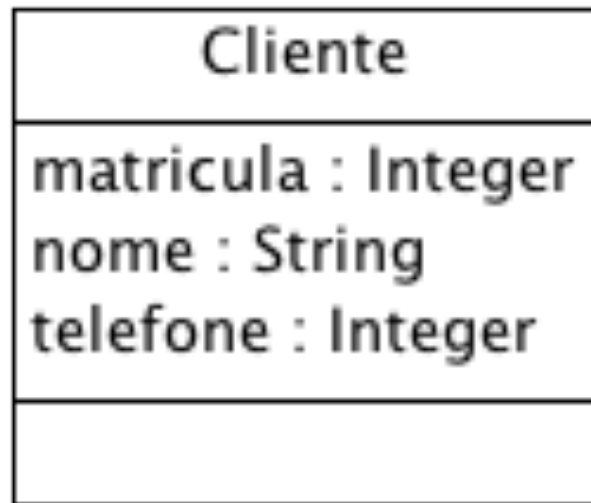
[https://docs.oracle.com/javase/8/docs/api/java/  
util/Collection.html](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html)



Como garantir que um conjunto  
não possua dois objetos complexo  
iguais?

# Igualdade em Hash

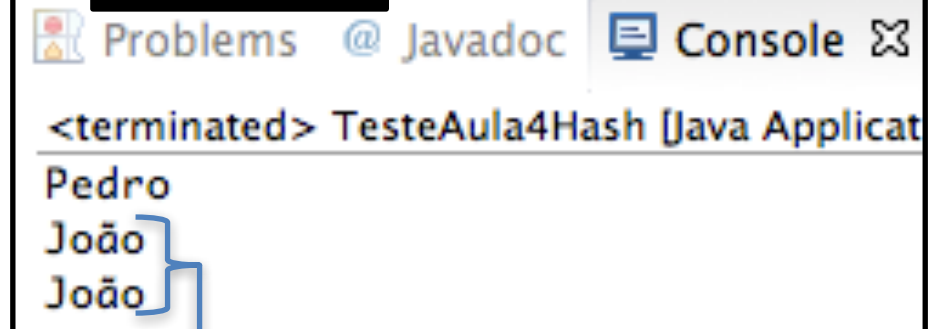
- Para entendermos como é possível identificar se dois elementos (objetos complexos) são iguais e devem ser ou não adicionados em um Hash, formulou-se a seguinte classe:



# Adicionando clientes em um Hash

```
public void adicionaClienteHash() {  
    Cliente cliente = null;  
    Set<Cliente> conjuntoCliente = new HashSet<Cliente>();  
  
    cliente = new Cliente();  
    cliente.setNome("João");  
    conjuntoCliente.add(cliente);  
  
    cliente = new Cliente();  
    cliente.setNome("Pedro");  
    conjuntoCliente.add(cliente);  
  
    cliente = new Cliente();  
    cliente.setNome("João");  
    conjuntoCliente.add(cliente);  
  
    for(Cliente obj : conjuntoCliente) {  
        System.out.println(obj.getNome());  
    }  
}
```

## CONSOLE



<terminated> TesteAula4Hash [Java Applicat]  
Pedro  
João  
João

Para objetos complexos o conjunto não retira duplicidade.



# Adicionando clientes em um Hash

- Para adicionar um cliente (ou qualquer outro tipo de objeto complexo) em um hash, é necessário implementar dois métodos:
  - hashCode; e
  - equals.
- Esses métodos devem ser implementados na classe do objeto complexo, no nosso caso na classe Cliente.

# Adicionando clientes em um Hash

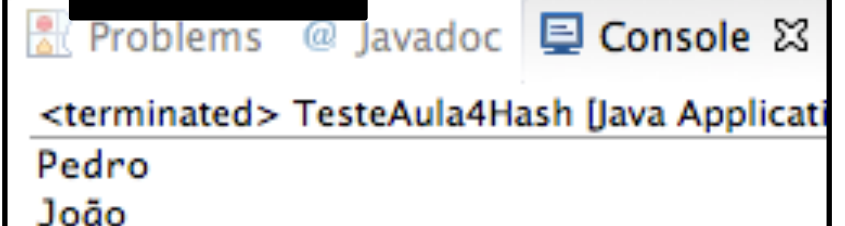
```
@Override
public int hashCode() {
    //Retornar no valor do hashCode do atributo nome
    return nome.hashCode();
}

@Override
public boolean equals(Object obj) {
    //É feito o cast para o tipo da classe
    Cliente cliente = (Cliente) obj;

    //Foi definido que o critério de igualdade do objeto
    //era o atributo nome.
    if(this.nome != null && cliente.getNome() != null) {
        if(this.nome.equals(cliente.getNome())) {
            return true;
        }
    }

    return false;
}
```

## CONSOLE



<terminated> TesteAula4Hash [Java Applicati  
Pedro  
João

# Adicionando clientes em um Hash

- O mesmo comportamento deve ser implementado, quando deseja-se que um objeto complexo, seja chave de um Mapa.
- Da mesma forma que um Mapa não pode possuir duas chaves iguais, também é necessário adicionar os métodos ***equals*** e ***hashCode*** ao objeto chave.

# Tratamento de exceções

# Tratamento de exceções

- O que é uma exceção?
  - **Exceção** é um evento que ocorre fora do padrão de execução esperado;
  - São muito utilizadas pelos Sistemas Operacionais e demais sistemas;
  - Usar tratamento de exceções permite detectar erros e manipular esses erros, ou seja, tratá-los;

# Blocos para tratamento de exceções

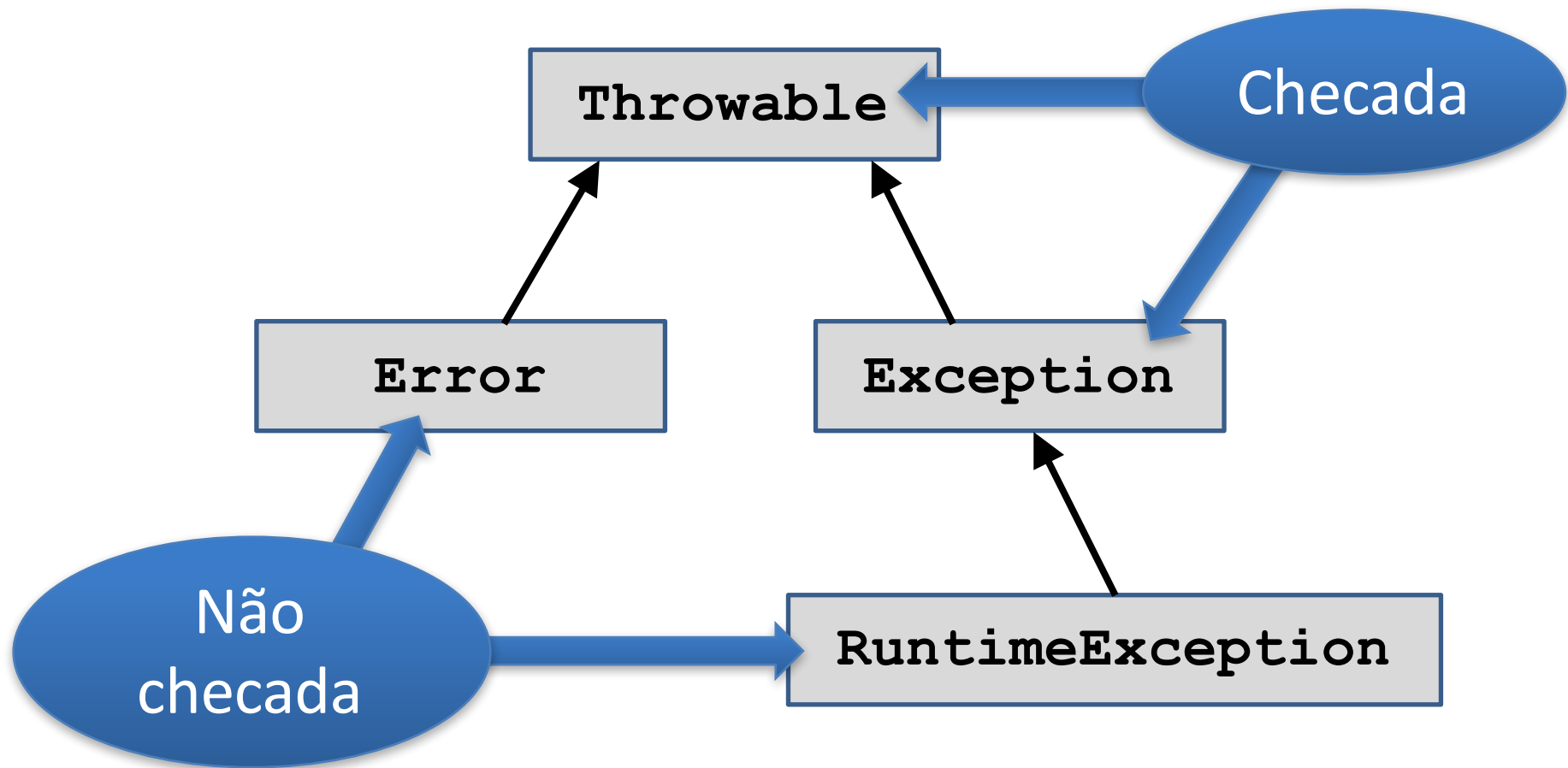
- **try:**
  - É usado para indicar um bloco de código que possa ocorrer uma exceção.
- **catch:**
  - Serve para manipular as exceções, ou seja, tratar o erro.
- **finally:**
  - Sempre será executado depois do bloco try/catch;
  - É importante saber que esse bloco sempre será executado;
  - Sempre que se trabalhar com a abertura de um recurso (arquivo, conexão,...) deve-se fechar a mesma aqui.

# Blocos para tratamento de exceções - Exemplo

```
public void metodoBD() {  
    Connection conn = null;  
  
    try {  
        conn = DatabaseService.getConnPostgres();  
  
        //TODO fazer a operação desejada  
  
    } catch (Exception e) {  
        System.err.println(e);  
  
        } finally {  
            if(conn != null) {  
                try {  
                    conn.close();  
                } catch (Exception e) {}  
            }  
        }  
    }  
}
```

O método abriu recurso?  
Então depois do seu uso  
deve-se fechá-lo

# Visão hierárquica





# Exception

- As classes que deveriam aqui lançar exceções e não erros de programação.
  - Exemplo: tentar abrir um arquivo que não existe. Então, é lançada uma exceção verificada, porque a classe de leitura de arquivos deriva de ***Exception***.

# Runtime Exception

- São exceções que indicam erros de programas (não de lógica, pois senão não passaria pelo compilador).
- Esse tipo de exceção é conhecida como não checada. Sendo assim, não é requisito declarar uma cláusula `try{}` e `catch{}`. Ex.: tentar converter "dois" em "2".

# Error

- Causadas por condições externas à aplicação, e normalmente a aplicação não tem como antecipar-se e tratar.
  - **Exemplo:** a aplicação abre um arquivo para leitura, mas não consegue gravar por uma falha de hardware ou mal funcionamento do sistema. Isso causará o lançamento de `java.io.IOException`.
- Não são exceções, e sim erros que jamais poderiam ter acontecido.
  - **Exemplo:** estouro da memória.

# Exceções checadas

- Há situações excepcionais que uma aplicação bem escrita devem antecipar e tratar.
- Exceções checadas estão sujeitas ao *Catch or Specify Requirement*.
- Todas exceções são checadas, exceto aquelas indicadas por **Error** e **RuntimeException** e suas especializações.

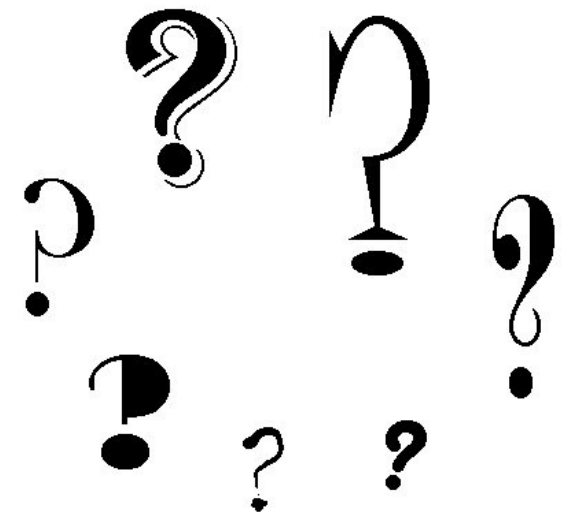
# Exceções não checadas

- Causada por condições internas à aplicação e normalmente a aplicação não tem como recuperar-se ou tratar.
  - Geralmente indica problema no desenvolvimento: se um erro de lógica permite, por exemplo, a chegada de uma referência nula ao construtor de ***FileReader***, ***NullPointerException*** será lançada.
- O desenvolvedor pode capturar a exceção (catch), mas faz mais sentido corrigir o erro.

Vish...



Mas quando utilizar exceções  
checadas e não checadas?



# Exceções checada X não checada

- Exceções não checadas representam o resultado de um problema de programação do qual não há como lidar ou recuperar de maneira alguma.
  - Exemplos: exceções aritméticas, referência nula e erro de indexação (tentativa de acessar um índice que não existe).

# Exceções checada X não checada

- Não lance ***RuntimeException*** ou uma especialização dela só porque não quer se incomodar com a especificação de exceções que podem ser lançadas.
- Se é razoável que um cliente receba a exceção para tratá-la, use exceção checada.
  - Se, no entanto, não tiver o que fazer a respeito do erro, faça-a não checada.



# Erros comuns no uso da Exceptions

- `catch (Exception e) {}`
  - Nunca deixe o bloco catch vazio!
- Continuação do fluxo de execução após erro
  - Analise sempre até que ponto o fluxo deve ser interrompido quando ocorrer uma exceção.
- Falha relatada incorretamente
  - Verifique com cautela a exceção capturada, tenha certeza do que será feito log.

# Boas práticas (onjava.com)

- Sempre feche recursos após usá-los
- Não use exceções para controlar fluxo
- Não ignore ou “mate” a exceção
- Não capture exceções de primeiro nível:

```
try{  
  ..  
} catch (Exception ex) {  
}
```

Fazendo isso, capturam-se todas as exceções, inclusive ***RuntimeExceptions***.



# Trabalho Integrador

Biblioteca Universitária



# Trabalho Integrador

- Deve ser feito em dupla;
- O trabalho contempla:
  - Trabalho escrito com modelagem e demais artefatos solicitados na documentação; Scripts de criação da base de dados;
  - Desenvolvimento de uma aplicação Java conforme os requisitos passados;
- Data limite para entrega: 29/11/2016;
  - Pode ser entregue a partir de: 22/11/2016;

# Trabalho Integrador

- O trabalho deve ser apresentado para o professor;
  - A apresentação vale 2 pontos;
- Deve-se utilizar boas práticas de programa (e da orientação à objeto);
- Para cada dia de atraso na entrega será decrementado 2 pontos da nota total.
  - Deve-se entregar o trabalho escrito + código fonte e mais os scripts do banco de dados.

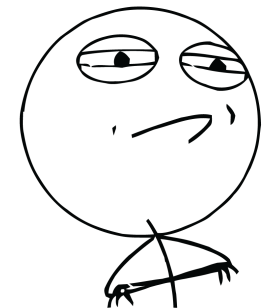
# Trabalho Integrador

- Existe uma entrega parcial que deve ser feita no dia: **11/10/2016**;
- TODOS os requisitos apresentados nessa aula devem estar contemplados na entrega do dia **11/10/2016**.
- Nesse dia não teremos aula, o aluno tem até a meia noite dessa data para enviar o código com o estado atual do projeto.

# Trabalho Integrador

- A biblioteca universitária solicitou que os alunos do curso de Sistemas de Informação da Unisul desenvolvessem um sistema para controle de empréstimo de livros.
- Como os alunos de Tópicos Avançados de Programação conhecem tudo sobre Java, eles foram os incumbidos para esta tarefa!

**CHALLENGE ACCEPTED**



# Trabalho Integrador

- Requisitos:
  - O sistema deve armazenar todas informações em memória de modo que seja utilizada a estrutura de dados mais adequada para as operações em questão;
  - O sistema deve permitir o cadastro de alunos;
  - O sistema deve permitir o cadastro de professores
    - Tanto alunos como professores devem ser mantidos na mesma estrutura;
    - A forma de consulta de ambos é via sua matrícula;



# Trabalho Integrador

- Requisitos:
  - O sistema deve permitir o cadastro de livros;
    - Não são permitidos o cadastro de livros repetidos;
    - Um livro deve ter uma lista de exemplares;
  - O sistema deve permitir o vinculo de um exemplar com um usuário (aluno ou professor) da biblioteca
    - Cada usuário pode ter até 5 livros emprestados simultaneamente (não é permitido pegar mais de um exemplar do mesmo livro)

# Trabalho Integrador

- Característica do aluno:
  - Matricula;
  - Nome;
  - Nome do curso;
  - Exemplares pegos;



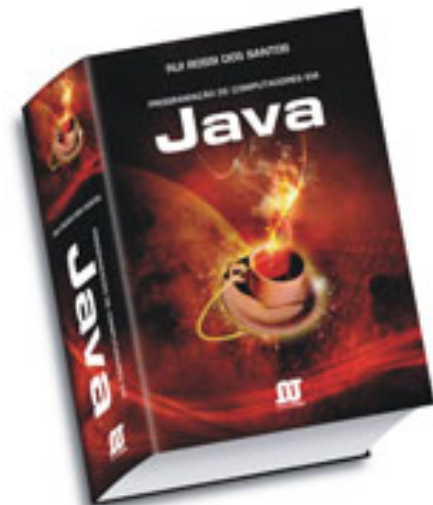
# Trabalho Integrador

- Característica do professor:
  - Matricula;
  - Nome;
  - Lista de cursos que participa;
  - Exemplares pegos;



# Trabalho Integrador

- Característica do livro:
  - Título;
  - Autor;
  - Lista de exemplares;



# Trabalho Integrador

- Característica do exemplar:
  - Código;
  - Localização
  - Edição



# Trabalho Integrador

- O sistema deve possuir uma interface (menu) para navegar entre as opções de operações;
- Outras operações permitidas (além do cadastro):
  - Fazer uma consulta por livro e/ou exemplar;
  - A partir da visualização da consulta o usuário pode fazer as seguintes operações:
    - Alterar os dados do objeto;
    - Excluir o objeto da estrutura de dados.