

Aula 15

Estrutura de Funções e Tratamento de Exceções

Eiji Adachi Barbosa

LES / DI / PUC-Rio

Abril / 2011

- Correção do questionário será entregue até o fim deste mês
- Próxima aula (20/abril/2011) haverá exercício em sala de aula valendo 1 ponto no T2
 - Exercício em dupla (preferencialmente, a mesma dupla dos trabalhos)

- Definições básicas
 - O que é função?
 - Por que / para que usar funções?
 - Pilha de chamadas
- Finalização de uma função
- Função de “Arrumação da casa”
- Especificação de uma função
 - Especificação do acoplamento de uma função
- Tratamento de exceções
 - Em C
 - Em linguagens de programação contemporâneas

- O que é uma **função**?
 - **Função** é uma porção auto-contida de código que:
 - possui um **nome** que pode ser chamado (ativado) de diversos lugares
 - pode **retornar** zero ou mais **valores**
 - pode depender de e alterar zero ou mais **parâmetros**
 - pode **alterar** zero ou mais **valores do estado** do módulo
 - variáveis internas
 - variáveis globais
 - pode **criar, ler e destruir arquivos**, etc...

```
TIPO_RETORNO NOME_FUNCAO ( LISTA_PARAMETROS)
{
    CORPO_FUNCAO
}
```

- Por que / para que usar funções?
 - Princípio dividir para conquistar
 - Dividir sistema em módulos \leftrightarrow Dividir algoritmo em funções
 - Evitar códigos monolíticos
 - Reuso e manutenibilidade

Definições básicas - Pilha de chamadas



```
int main() {  
    firstCall();  
    return 0;  
}  
void firstCall(){  
    printf("First Call\n");  
    secondCall();  
    return;  
}  
int secondCall(){  
    printf("Second Call\n");  
    thirdCall();  
    return 0;  
}  
char thirdCall(){  
    printf("Third Call\nOK, That's  
    enough.\n");  
    return '0';  
}
```

int printf(const char *...)
void firstCall()
int main()

```
int main() {  
    firstCall();  
    return 0;  
}  
void firstCall(){  
    printf("First Call\n");  
    secondCall();  
    return;  
}  
int secondCall(){  
    printf("Second Call\n");  
    thirdCall();  
    return 0;  
}  
char thirdCall(){  
    printf("Third Call\nOK, That's  
    enough.\n");  
    return '0';  
}
```

int printf(const char *...)
int secondCall()
void firstCall()
int main()

```
int main() {  
    firstCall();  
    return 0;  
}  
void firstCall(){  
    printf("First Call\n");  
    secondCall();  
    return;  
}  
int secondCall(){  
    printf("Second Call\n");  
    thirdCall();  
    return 0;  
}  
char thirdCall(){  
    printf("Third Call\nOK, That's  
    enough.\n");  
    return '0';  
}
```

int printf(const char *...)
char thirdCall()
int secondCall()
void firstCall()
int main()

- Encerrando a execução de uma função:
 - Chegar ao fim de uma função void
 - O comando *return <VALUE>*
 - Encerra a execução de uma função imediatamente
 - Se um valor de retorno é informado, a função chamada (*callee*) retorna este valor para a função chamadora (*caller*)
 - A transferência de controle é local, i.e., após o *return* o controle do fluxo de execução passa da função chamada para a função chamadora
 - O comando *exit(int)*
 - Encerra a execução do programa
 - Executa em ordem reversa todas as funções registradas pela função *int atexit(void (*func)(void))*
 - Todos streams são fechados, todos arquivos temporários são apagados
 - O controle de execução retorna ao ambiente-hospedeiro (*host environment*) o valor inteiro passado como argumento

• ,

- Deve existir uma **sub-estrutura de funções** cuja raiz coordena o controle de alocação e liberação de recursos. Exemplo:

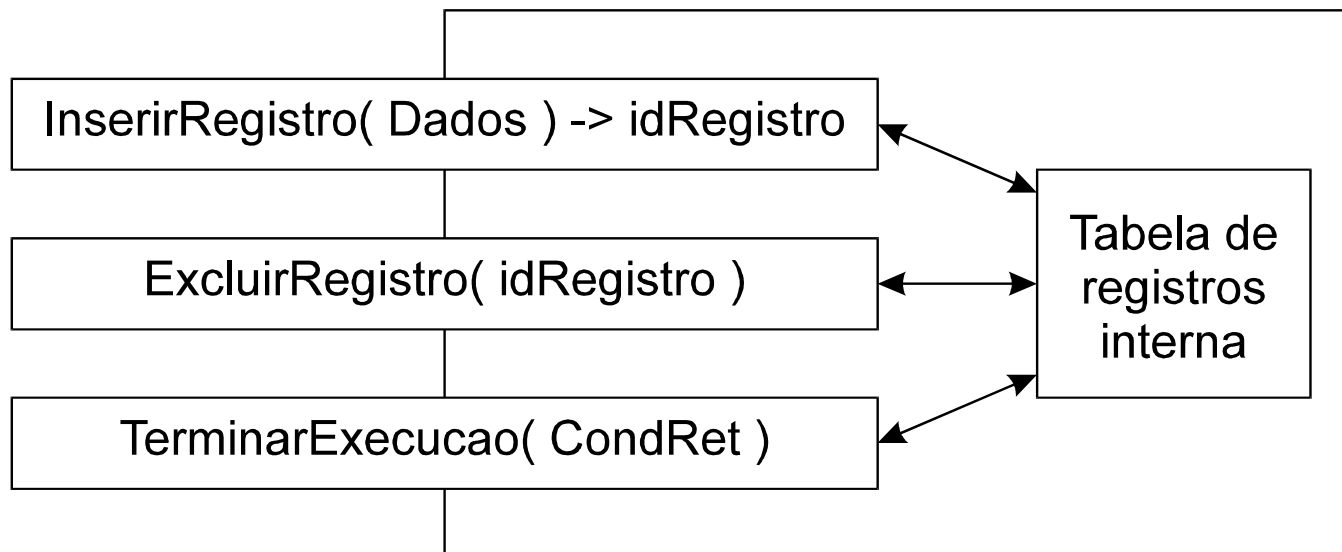
```
pArq = AbrirArquivo( NomeArquivo ) ;  
if ( pArq != NULL )  
{  
    ProcessarArquivo( pArq ) ;  
    FecharArquivo( pArq ) ;  
} /* if */
```
- Devem ser **liberados todos** os espaços dinâmicos e recursos alocados e que não serão retornados, ou que não foram incorporados a uma estrutura de dados ancorada durante a execução
 - requer exame do código com vistas a verificar se existe **algum caminho** em que o espaço ou recurso perde a âncora quando a função termina a execução.

- Ao **terminar a execução** de um **programa** todos os recursos alocados neste programa devem ser liberados, independentemente de como foi terminado o programa
- Exemplos:
 - arquivos abertos
 - memória alocada
- Deve ser **restaurado estado do sistema** para o que era antes de iniciar a execução do programa
- Função *int atexit(void (*func)(void))*

Arrumação da casa: finalização do programa

- Assegurar a liberação de recursos alocados é particularmente **complexo** quando o programa utiliza funções de finalização capazes de **cancelar a execução**
- Pode-se utilizar um módulo de **arrumação da casa** (*housekeeping*), ou de **finalização** para ajudar a resolver
- De qualquer maneira é necessária **disciplina** ao programar

Módulo: Arrumação-da-casa



- A especificação de uma função define sua interface conceitual, em termos de:
 - Objetivo da função
 - Acoplamento
 - Identifica todos os itens da interface
 - Pré-condição (Assertivas de entrada)
 - Pós-condição (Assertivas de saída)
 - Requisitos
 - Propriedades de desempenho a serem satisfeitas: tempo de resposta, capacidade de armazenamento...
 - Hipóteses
 - Windows, Linux, Mac? 32bit, 64bit?

2 Aulas passadas

- **Acoplamento**
 - identifica **todos os itens da interface** e respectivos tipos
 - não somente os elementos da assinatura
 - são exemplos de itens do acoplamento
 - parâmetros recebidos
 - parâmetros modificados (chamada por referência)
 - valores retornados, condições de retorno
 - dados globais manipulados e/ou modificados
 - arquivos manipulados
 - exceções sinalizadas
 - cancelamentos realizados – `exit(num)`

- 1. De forma visual, tais como: modelo de componentes
 - comentários podem ser usados para indicar dados resultantes ou modificados
- 2. Interface em notação similar a uma linguagem de programação
 - listam-se como parâmetros **todos os dados** ao entrar
 - independentemente se serão explícitos ou implícitos
 - sem se preocupar com a forma de realizar
 - por exemplo: diz-se **Tabela** ao invés de **pTabela**, mesmo se fisicamente a tabela será identificada por um ponteiro para a sua cabeça
 - listam-se como saída **todos os dados** resultantes ou modificados
 - exemplo

```
InserirSimbolo( tpTabela Tabela, tpSimbolo Simbolo ) ⇒  
    tpTabela  Tabela , tpIdSimb IdSimbolo ,  
    tpCondRet CondRet
```

- 3. Texto explanatório, separando entrada e saída

InserirSimbolo(tpTabela Tabela, tpSimbolo Simbolo) \Rightarrow
tpTabela Tabela, tpIdSimb IdSimbolo, tpCondRet CondRet

Recebe dados

Tabela - Tabela em que será inserido o símbolo

Simbolo - Símbolo válido a inserir

Produce resultados

Se não ocorreu erro

Se o símbolo era novo:

Cria um idSimbolo diferente dos existentes

Tabela acrescida do par < idSimbolo, Simbolo >

Se o Simbolo já existia

Tabela inalterada

FimSe

idSimbolo o id associado ao Simbolo

Se ocorreu erro:

Tabela inalterada

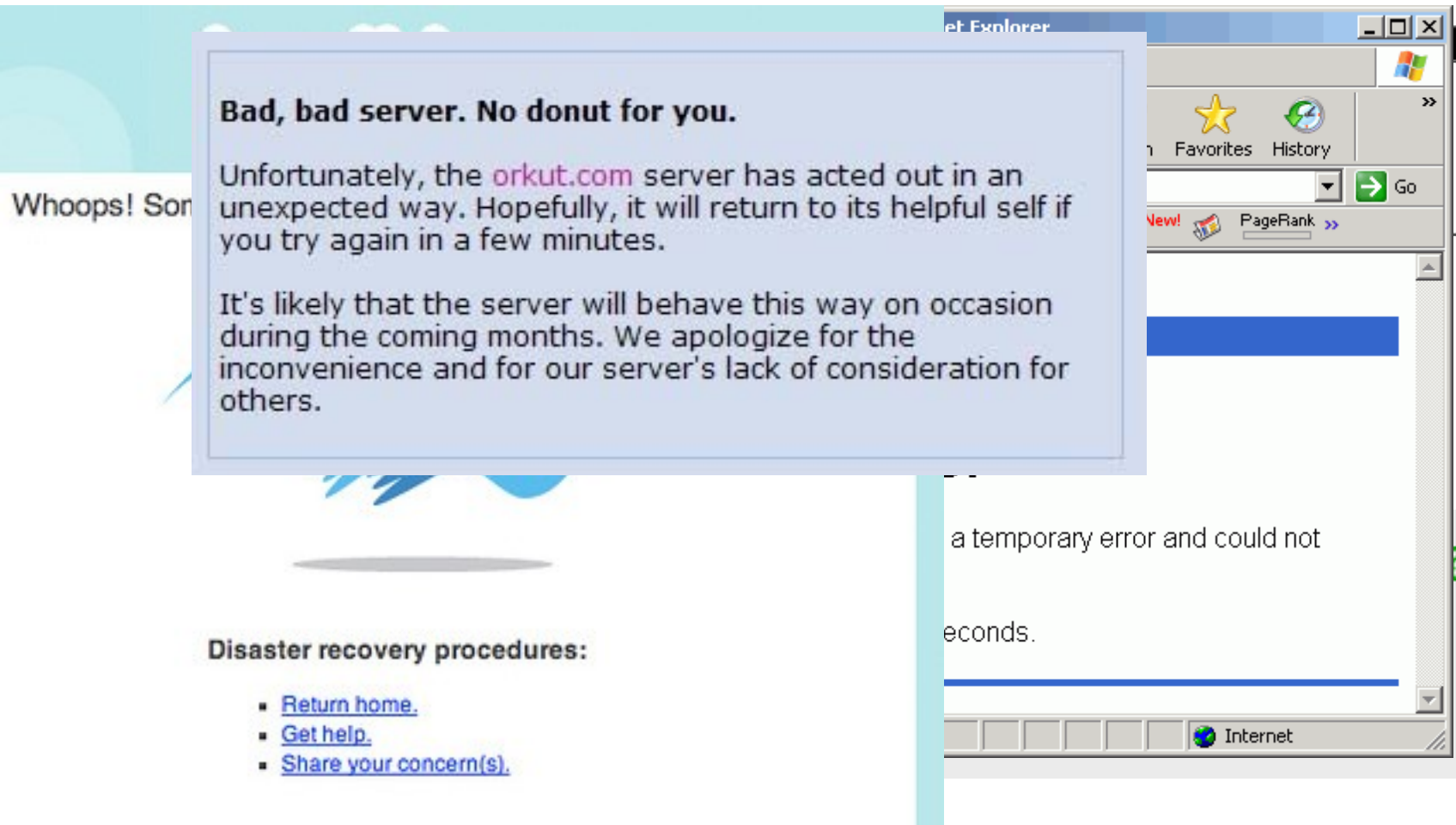
idSimbolo = NIL_SIMBOLO

FimSe

CondRet ver tipo tpCondRetTabela

- Por que é importante tratar exceções?
 - Você pode irritar o seu usuário

- Por que é importante tratar exceções?
 - Você pode irritar o seu usuário



- Por que é importante tratar exceções?
 - Os usuários podem perder a confiança no seu produto
 - ... e você pode passar uma tremenda vergonha!

- Vídeo do Bill Gates:
 - <http://www.youtube.com/watch?v=TrAD25V7II8>

- Por que é importante tratar exceções?
 - Pode custar milhões de dólares/reais/euros
 - Ou até mesmo vidas

- Vídeo do Ariane 5:
 - <http://www.youtube.com/watch?v=kYUrqdUyEpl>

- Em outubro de 1996 o foguete francês Ariane 501 se autodestruíu 5 minutos após decolar
- Motivo:
 - cálculo errado do ângulo de ataque
- Causa:
 - O Ariane 5 reutilizou um módulo do Ariane 4 para calcular o “Alinhamento Interno”, elemento relacionado com a componente horizontal da velocidade
 - O valor gerado pelo módulo do Ariane 4 foi muito maior do que o esperado pelo módulo do Ariane 5, pois a trajetória do Ariane 5 difere da trajetória do Ariane 4
 - O módulo do Ariane 5 tentou converter o valor do “Alinhamento Interno” de um número de 64 bits em ponto flutuante para um inteiro de 16 bits com sinal
 - Valor em ponto flutuante era maior do que poderia ser representado por um inteiro
 - Operação de conversão não estava protegida
 - *Overflow Exception*

- Exemplos de tratadores:
 - **recuperação por retrocesso**: o tratador desfaz modificações nas variáveis locais e retorna um resultado indicando insucesso na operação
 - **fechamento**: o tratador garante que uma conexão/arquivo aberta(o) seja fechada(o)
 - **notificação ao usuário**: nenhuma recuperação, somente notifica ao usuário que uma exceção ocorreu
 - **armazenamento de erro**: criação de registro de erro em arquivo especial
 - **re-sinaliza uma exceção diferente...** Para a função chamadora
 - **nova tentativa**: a mesma função ou uma diferente implementação (daquela função) é invocada
 - é usada em técnicas como bloco de recuperação ou programação N-Versões
 - **tratador vazio**: “silencia” a exceção – má pratica e deve ser evitado
 - **assert**: o tratador desempenha algum tipo de operação de assert; quanto a assertiva não é válida, isto resulta em uma nova exceção, possivelmente levando ao término da execução do programa

- Linguagens contemporâneas (Java, JavaScript, C++, C#, Python...) possuem mecanismos de tratamento de exceções implementados na própria linguagem
 - TRY-CATCH-FINALLY
- A linguagem C não traz suporte específico para tratamento de exceções, por isso o faz através de um idioma:
 - Testando condições de retorno
 - Modificando / testando variáveis globais ou parâmetros passados por referência

São usados de forma complementar!

- O comando *return* é utilizado pela função chamada para indicar sob qual condição (normal | excepcional) sua execução encerrou
- A função chamadora deve testar o código retornado pela função chamada a fim de tomar ações corretivas, caso necessário
- Preferencialmente, as condições de retorno devem ser declaradas como um elemento *enum*

```
typedef enum {  
    LIS_CondRetOK ,  
    /* Concluiu corretamente */  
  
    LIS_CondRetListaVazia ,  
    /* A lista não contém elementos */  
  
    LIS_CondRetFimLista ,  
    /* Foi atingido o fim de lista */  
  
    LIS_CondRetNaoAchou ,  
    /* Não encontrou o valor procurado */  
  
    LIS_CondRetFaltouMemoria  
    /* Faltou memória ao tentar criar um  
    elemento de lista */  
} LIS_tpCondRet ;
```

```
LIS_tpCondRet LIS_InserirElementoAntes  
( LIS_tppLista pLista , void * pValor )  
{  
    tpElemLista * pElem ;  
  
    pElem = CriarElemento( pLista , pValor ) ;  
    if ( pElem == NULL ) {  
        return LIS_CondRetFaltouMemoria ;  
    } /* if */  
  
    ....  
  
    return LIS_CondRetOK ;  
  
} /* Fim função: LIS &Excluir elemento */
```

```
int main(void){  
    ...  
    LIS_tpCondRet condRet = InserirElementoAntes( lista, pValor );  
  
    switch( condRet ) {  
        case LIS_CondRetFaltouMemoria:  
            ...  
        case LIS_CondRetOK:  
            ...  
        default :  
            printf("Condição de retorno inesperada");  
    }  
}
```

- A função chamada deve modificar variáveis globais ou parâmetros passados por referência para indicar sob qual condição (normal | excepcional) sua execução encerrou
- A função chamadora deve testar a variável global, ou o parâmetro passado por referência, a fim de tomar ações corretivas, caso necessário

```
LIS_tpCondRet LIS_InserirElementoAntes
( LIS_tppLista pLista , void * pValor, char ** errorMsg )
{
    tpElemLista * pElem ;

    pElem = CriarElemento( pLista , pValor ) ;
    if ( pElem == NULL ) {
        char str[] = "Não foi possível alocar memória para um novo elemento";
        int size = strlen( str ) + 1;
        (*errorMsg) = (char*)malloc( sizeof(char) * size );
        memcpy( (*errorMsg), str, size );
        return LIS_CondRetFaltouMemoria ;
    } /* if */
    ....

    return LIS_CondRetOK ;

} /* Fim função: LIS &Excluir elemento */
```

Usar uma variável global,
seria análogo...

Usando parâmetro passado por referência



```
int main(void){  
    ...  
    char *errorMsg;  
    LIS_tpCondRet condRet = InserirElementoAntes( lista, pValor, &errorMsg );  
  
    switch( condRet ) {  
        case LIS_CondRetFaltouMemoria:  
            printf( "%s", errorMsg );  
        case LIS_CondRetOK:  
            ...  
        default :  
            printf("Condição de retorno inesperada");  
    }  
}
```

Usar uma variável global,
seria análogo...

- A sinalização de uma exceção não é explícita
 - Usa-se o comando *return*, parâmetros passados por referência ou variáveis globais
- Nem sempre é possível retornar um elemento enumerado como condição de retorno
 - *Ex.: Trabalho 1 – Implemente uma função que receba os três parâmetros (Nome, Iniciais, Idade) e retorne por referência a estrutura preenchida.*
- Como prover mais informações a respeito do problema / exceção?
 - *Ex.: Qual a severidade? Que condições levaram a esta ocorrência?*

- Há um overhead na criação de tipos enumerados para cada módulo
- A associação entre as exceções descritas nos tipos enumerados e quais exceções que podem ser levantadas por uma função depende exclusivamente da especificação da função
 - Difícil entender o acoplamento excepcional entre funções: quais exceções devem ser tratadas?
- Não há separação textual do código de tratamento de exceção
 - Código torna-se rapidamente extenso, complexo e pouco compreensível
- Como assegurar que as pós-condições da função serão satisfeitas, mesmo em casos excepcionais?

- Linguagens como Java, JavaScript, C++, C#, Python ... provêm mecanismos de tratamento de exceções implementados na própria linguagem
 - TRY - define uma região protegida contra a ocorrência de exceções
 - CATCH - define um tratador, i.e., um trecho de código que implementa um conjunto de ações de recuperação
 - FINALLY - define um trecho de código que sempre será executado, mesmo quando exceções ocorrerem
 - THROW - sinaliza a ocorrência de uma exceção
 - THROWS - especifica o acoplamento excepcional de uma função

```
static void escreveArquivo(Arquivo) throws  
    FileNotFoundException,  
        CharCodingException,  
        PermissionException;
```

```
static void escreveArquivo(Arquivo a) throws
    FileNotFoundException,
        CharCodingException,
        PermissionException {

    Buffer bf = buscaArquivo( a );
    if( bf == null )
        throw new FileNotFoundException();
}
```

Melhor separação textual



```
ARQ_tpCondRet leArquivo(){
    condRet = OK;
    abreArquivo();
    se( arquivoAberto() ){
        determineTamanhoArquivo();
        se( determinouTamanho() ){
            aloqueMemoria();
            se( alocouMemoria() ){
                copieDados();
                se( ! copiouDados() ){
                    condRet = ERRO_COPIAR_DADOS;
                }
            } senão {
                condRet = ERRO_ALOCAR_MEM;
            }
        } senão {
            condRet = ERRO_DET_TAM;
        }
        fecheArquivo();
        se( ! fechouArquivo() ){
            condRet = ERRO_FECHAR_ARQ;
        }
    } senão {
        condRet = ERRO_ABRIR_ARQ;
    }
}
```

```
leArquivo(){
    try{
        abreArquivo();
        determineTamanhoArquivo();
        aloqueMemoria();
        copieDados();
    } catch( abrirErro ){...}
    catch( determinarTamanhoErro ) {...}
    catch( alocarMemoriaErro ) {...}
    catch( copiarDadosErro ) {...}
    finally {
        try{
            fecheArquivo();
        } catch( fecharArquivoErro ){...}
    }
}
```

Execução e transferência não-local




```
Lista preencheLista(){
    Arquivo arq = null;
    Lista lista = null;
    try{
        arq = abreArquivo( "Dados.txt" );
        lista = criaLista();
        for( i=0; i<SIZE; i++ ){
            adiciona( lista, arq, i );
        }
        //faz mais alguma coisa
    } catch( RegistroException ){
        print( "Registro lido incorretamente" );
    } finally {
        if( arq != null){
            fechaArquivo( arq );
        }
    }
    return lista;
}
```

```
void adiciona(Lista lista, Arquivo arq,
               int i ) throws RegistroException {

    Registro r = leRegistro( arq, i );
    Elemento e = criaElement( r );
    adicionaElemento( lista, e, i );
}
```

catch(RegistroException)
finally
preecheLista()

Execução e transferência não-local



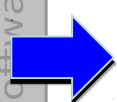
```
Lista preencheLista(){
    Arquivo arq = null;
    Lista lista = null;
    try{
        arq = abreArquivo( "Dados.txt" );
        lista = criaLista();
        for( i=0; i<SIZE; i++ ){
            adiciona( lista, arq, i );
        }
        //faz mais alguma coisa
    } catch( RegistroException ){
        print( "Registro lido incorretamente" );
    } finally {
        if( arq != null){
            fechaArquivo( arq );
        }
    }
    return lista;
}
```

```
void adiciona(Lista lista, Arquivo arq,
              int i ) throws RegistroException {

    Registro r = leRegistro( arq, i );
    Elemento e = criaElement( r );
    adicionaElemento( lista, e, i );
}
```

abreArquivo
catch(RegistroException)
finally
preecheLista()

Execução e transferência não-local



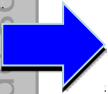
```
Lista preencheLista(){
    Arquivo arq = null;
    Lista lista = null;
    try{
        arq = abreArquivo( "Dados.txt" );
        lista = criaLista();
        for( i=0; i<SIZE; i++ ){
            adiciona( lista, arq, i );
        }
        //faz mais alguma coisa
    } catch( RegistroException ){
        print( "Registro lido incorretamente" );
    } finally {
        if( arq != null){
            fechaArquivo( arq );
        }
    }
    return lista;
}
```

```
void adiciona(Lista lista, Arquivo arq,
               int i ) throws RegistroException {

    Registro r = leRegistro( arq, i );
    Elemento e = criaElement( r );
    adicionaElemento( lista, e, i );
}
```

criaLista
catch(RegistroException)
finally
preecheLista()

Execução e transferência não-local

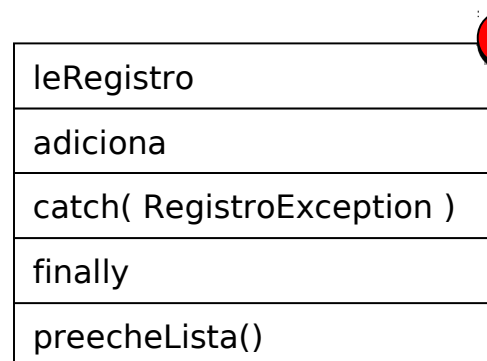


```
Lista preencheLista(){
    Arquivo arq = null;
    Lista lista = null;
    try{
        arq = abreArquivo( "Dados.txt" );
        lista = criaLista();
        for( i=0; i<SIZE; i++ ){
            adiciona( lista, arq, i );
        }
        //faz mais alguma coisa
    } catch( RegistroException ){
        print( "Registro lido incorretamente" );
    } finally {
        if( arq != null){
            fechaArquivo( arq );
        }
    }
    return lista;
}
```



```
void adiciona(Lista lista, Arquivo arq,
    int i ) throws RegistroException {
```

```
    Registro r = leRegistro( arq, i );
    Elemento e = criaElement( r );
    adicionaElemento( lista, e, i );
}
```



Execução e transferência não-local

```
Lista preencheLista(){
    Arquivo arq = null;
    Lista lista = null;
    try{
        arq = abreArquivo( "Dados.txt" );
        lista = criaLista();
        for( i=0; i<SIZE; i++ ){
            adiciona( lista, arq, i );
        }
        //faz mais alguma coisa
    } catch( RegistroException ){
        print( "Registro lido incorretamente" );
    } finally {
        if( arq != null){
            fechaArquivo( arq );
        }
    }
    return lista;
}
```

```
void adiciona(Lista lista, Arquivo arq,
               int i ) throws RegistroException {
    Registro r = leRegistro( arq, i );
    Elemento e = criaElement( r );
    adicionaElemento( lista, e, i );
}
```

Esse trecho de código não é executado!

leRegistro
adiciona
catch(RegistroException)
finally
preecheLista()

Execução e transferência não-local

```
Lista preencheLista(){
    Arquivo arq = null;
    Lista lista = null;
    try{
        arq = abreArquivo( "Dados.txt" );
        lista = criaLista();
        for( i=0; i<SIZE; i++ ){
            adiciona( lista, arq, i );
        }
        //faz mais alguma coisa
    } catch( RegistroException ){
        print( "Registro lido incorretamente" );
    } finally {
        if( arq != null){
            fechaArquivo( arq );
        }
    }
    return lista;
}
```

```
void adiciona(Lista lista, Arquivo arq,
               int i ) throws RegistroException {

    Registro r = leRegistro( arq, i );
    Elemento e = criaElement( r );
    adicionaElemento( lista, e, i );
}
```

Esse trecho de código não é executado!

print
catch(RegistroException)
finally
preecheLista()

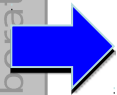
Execução e transferência não-local

```
Lista preencheLista(){
    Arquivo arq = null;
    Lista lista = null;
    try{
        arq = abreArquivo( "Dados.txt" );
        lista = criaLista();
        for( i=0; i<SIZE; i++ ){
            adiciona( lista, arq, i );
        }
        //faz mais alguma coisa
    } catch( RegistroException ){
        print( "Registro lido incorretamente" );
    } finally {
        if( arq != null){
            fechaArquivo( arq );
        }
    }
    return lista;
}
```

```
void adiciona(Lista lista, Arquivo arq,
               int i ) throws RegistroException {

    Registro r = leRegistro( arq, i );
    Elemento e = criaElement( r );
    adicionaElemento( lista, e, i );
}
```

Esse trecho de código não é executado!



fechaArquivo
finally
preecheLista()

- Dificuldade em testar código de tratamento de exceção:
 - Código tratador é pouco executado
 - Exceções ocorrem raramente
 - Como lançar a exceção e testar o tratador?
 - Descobrir quais assertivas devem ser quebradas
 - Mecanismos de injeção de erros
 - Qual ação esperada do tratador?

- Cap. 8 do livro Programação Modular

FIM