Capítulo VIII – Exceções

"Depois veio Darwin e nos disse que mesmo neste pequeno mundo entre mundos não éramos tão excepcionais assim."

Luis Fernando Verissimo

Nem todas condições geradoras de erro podem ser detectadas em tempo de compilação, mas um software seguro e confiável deverá implementar um comportamento aceitável mesmo na presença dessas condições anormais. Divisão por zero, falha na abertura de um arquivo, fim de arquivo, `overflow`, acesso a um índice inválido e utilização de um objeto não inicializado são exemplos típicos de condições anormais.

Caso essas condições especiais não sejam tratadas, o bloco de código na qual estão inseridas será interrompido quando executado ou gerará inconsistências, comprometendo a confiabilidade do software.

No estudo de linguagens de programação o termo exceção é usado para designar um evento ocorrido durante a execução de um programa que desvia o fluxo normal de instruções. Em outras palavras, uma exceção é uma condição provocada por uma situação excepcional a qual requer uma ação específica imediata.

Como se pode ver acima, muitos tipos de erros podem causar exceções. As causas de exceções podem variar desde erros sérios de hardware, tal como uma falha no disco rígido, a erros simples de programação, tal como tentar acessar um índice inexistente de um vetor.

Erros causam exceções, mas podem existir exceções que não são erros. Por exemplo, em uma função usada para encontrar um dado em uma sequência, a chegada ao final da sequência, sem encontrar o dado, é uma condição excepcional considerada como exceção. Normalmente, esse tipo de exceção ocorre em subprogramas capazes de produzir múltiplos resultados e alterar o fluxo normal de execução do programa.

A tabela 7.1 apresenta uma classificação das possíveis motivações para ocorrência de exceções durante a execução de um programa.

Exceções	Erros	Hardware
		Software
	Fluxo	Múltiplos Resultados

Tabela 8. 1 - Causas de Exceções

Nesse capítulo são apresentados o conceito de exceções, sua importância, como se apresentam e como podem ser tratadas nas diversas LPs. Inicialmente, discute-se como linguagens que não oferecem mecanismos de exceções lidam com situações anômalas. Em seguida, discute-se de forma geral os mecanismos de exceções oferecidos por linguagens de programação, apresentando-se como uma exceção é sinalizada, tratada, propagada, relançada e especificada. Mostra-se também como o fluxo de controle do programa pode ser continuado após o tratamento de uma exceção e como funciona a integração dos conceitos de orientação a objetos com os de tratamento de exceções.

8.1 Ausência de Mecanismo Específico para Exceções

Linguagens de programação podem não oferecer qualquer mecanismo específico para tratamento de exceções. Essas LPs tendem a produzir programas menos confiáveis e mais obscuros. Os programas podem se tornar menos confiáveis porque a linguagem não requer o tratamento das possíveis exceções. O código pode ficar mais obscuro porque o programador pode misturar código relacionado com a funcionalidade desejada com o código responsável por tratar as exceções.

Linguagens como C, PASCAL e MODULA-2 não oferecem mecanismos próprios para o tratamento de exceções e aumentam, assim, o trabalho dos programadores. Eles devem tratar as condições de erro em todos os locais onde possam aparecer. Várias alternativas têm sido adotadas pelos programadores para lidar com esse tipo de problema nessas LPs. As mais conhecidas são listadas a seguir:

1. Deixar abortar o programa.
2. Testar a condição excepcional antes de ela ocorrer e realizar o tratamento imediato.
3. Retornar código de erro indicando a exceção ocorrida em uma variável global, no resultado da função ou em um parâmetro específico.

A alternativa de deixar abortar o programa não é uma boa solução porque muitas exceções podem ser contornadas sem que seja necessário interromper a execução do programa. Além disso, o simples aborto do programa reduz a confiança do usuário no sistema e também dificulta a depuração dos erros, visto que não fornece indicações sobre o problema ocorrido.

A alternativa de testar a condição excepcional e tratá-la imediatamente apresenta vários problemas. Em primeiro lugar, ela carrega muito o texto do programa com código de tratamento, obscurescendo a funcionalidade

do algoritmo. Esse problema pode ser reduzido utilizando-se subprogramas específicos para tratamento de exceções. O local de tratamento da exceção é então separado do código onde a exceção ocorre, tornando-o mais claro. Além disso, por separar o código de tratamento da exceção em um único subprograma, ele pode ser reutilizado em outras situações nas quais o mesmo tipo de exceção ocorre. O exemplo 8.1 ilustra o uso desse tipo de abordagem em C.

```
int divideInteiros (int numerador, int denominador) {
  if (denominador == 0 )
    trata_divisao_zero();
  else
    return numerador / denominador;
}
int trata_divisao_zero(void) {
  printf("Divisao por zero");
  return 1;
}
```

Exemplo 8. 1 – Uso de Subprogramas Específicos no Tratamento de Exceções em C

Observe o uso da função *trata_divisao_zero*, a qual evita a colocação integral do código da função dentro da função *divideInteiros.* Caso vá se realizar uma outra função de divisão (por exemplo, divisão de números reais), basta chamar novamente *trata_divisao_zero* para realizar o mesmo tratamento dessa exceção.

Mesmo trazendo alguns benefícios, essa abordagem ainda é problemática porque o programador tem de lembrar, identificar e testar todas as possíveis condições causadoras de exceções. Isto normalmente não ocorre. Outro problema é a sobrecarga do texto do programa com testes de exceções, misturando o algoritmo com a identificação de exceções. Essa técnica ainda provoca, portanto, perda de legibilidade, uma vez que mistura a essência do subprograma (sua funcionalidade básica) com o código de identificação do erro. Por fim, algumas exceções não podem ser tratadas localmente, só podendo ser tratadas adequadamente em um ponto mais externo do código. O exemplo 8.2 mostra essa alternativa em uma função C.

```
void  executaFuncionalidade(int x) {
  printf (“Faz alguma coisa!!!”);
};
void f(int x) {
  if (condicao1(x)) trata1;
  if (condicao2(x)) trata2;
  if (condicao3(x)) {
    printf(“Nao consegue tratar aqui”);
```

```
    exit(1);
  }
  executaFuncionalidade(x);
}
```

Exemplo 8. 2 - Teste e Tratamento Imediato de Exceções em C

A função *f* do exemplo 8.2 tem o objetivo de executar uma determinada funcionalidade. Contudo, antes de poder executar essa funcionalidade é necessário verificar se o valor passado para o parâmetro *x* é apropriado. Caso o valor de *x* satisfaça a *condicao1* ou a *condicao2*, ocorre uma situação anômala que demanda um tratamento específico. Caso o valor de *x* satisfaça a *condicao3*, ocorre uma outra situação anômala, mas que não pode ser tratada de dentro de *f*. Nesse caso, a solução adotada pelo programador de *f* é abortar a execução.

A função *f* mostra que essa alternativa de tratamento de exceções não é muito satisfatória. Além de ainda ser necessário abortar a execução em certa condição, ela carrega o código de testes, dificultando a legibilidade. Mais que isso, ela não apóia o programador na identificação das exceções que precisam ser tratadas. Por via de regra, o programador acaba esquecendo de testar alguma condição excepcional.

A alternativa de retornar um código de erro implica na necessidade do código chamador realizar um tratamento para cada código de retorno. Embora essa alternativa resolva o problema de tratamento não local da exceção, ela mantém os dois outros problemas. De fato, o problema de sobrecarga de código fica ainda maior, uma vez que agora se necessita testar, em toda chamada de função, todos os possíveis códigos de erro retornáveis. Isso pode facilmente duplicar o tamanho de um programa. O exemplo 8.3 mostra como isso pode ocorrer em um código C.

```
int f(int x) {
  if (condicao1(x)) return 1;
  if (condicao2(x)) return 2;
  if (condicao3(x)) return 3;
  executaFuncionalidade(x);
}
void g() {
  int resp;
  resp = f(7);
  if (resp == 1) trata1;
  if (resp == 2) trata2;
  if (resp == 3) trata3;
}
```

Exemplo 8. 3 - Tratamento Não Local de Exceções em C

No exemplo 8.3, o tratamento das situações anômalas não é mais feito dentro de *f*. Isso agora é feito na função *g*, chamadora de *f*. Agora, além de se fazer testes dentro de *f*, têm-se também de fazer testes em *g*, após o retorno da chamada de *f*. As chances do programador esquecer de tratar alguma exceção também aumentam uma vez que o esquecimento também pode ocorrer nos testes em *g*.

A opção de usar o resultado da função como código de retorno nem sempre é possível porque pode haver incompatibilidade de valores e de tipo com o resultado normal da função (afinal, o retorno da função normalmente é usado para retornar o resultado da função e não um código). Por exemplo, a função *obtemTopo* do exemplo 6.1 retorna o valor *–1*, quando a pilha de números naturais está vazia. Essa solução não poderia ser usada caso a pilha fosse de números inteiros, uma vez que o valor *–1* é um número inteiro que poderia estar no topo da pilha.

C usa normalmente uma variável global, definida na biblioteca padrão e chamada *errno*, para retornar o código de retorno. Essa solução não é muito boa porque o usuário da função pode não ter ciência de que essa variável existe (uma vez que isso não fica explícito na sua chamada) e também porque uma outra exceção pode ocorrer antes do tratamento da anterior, sobrescrevendo o código de retorno da primeira exceção antes dela ser tratada efetivamente. Em particular, essa opção não funciona bem em programas concorrentes.

Já a opção de usar um parâmetro para retornar o código de exceção é melhor do que o retorno em variável global ou no resultado da função. Não obstante, ela exige a inclusão de um novo parâmetro nas chamadas dos subprogramas e requer a propagação desse parâmetro até o ponto de tratamento da exceção, diminuindo a redigibilidade do código.

Contudo, o grande problema relacionado com essa solução é o fato da experiência ter mostrado que, na maioria das vezes, o programador que chama a função não testa todos os códigos de retorno possíveis, uma vez que não é obrigatório fazê-lo.

C, especificamente, oferece ainda outras opções para o tratamento de exceções, tal como a utilização do sistema de manipulação de sinais de sua biblioteca padrão. Sinais podem ser gerados a partir da função *raise* ou em resposta a um comportamento excepcional do programa como um `overflow` ou um acesso indevido a memória e podem ser tratados através da função *signal* quando um desses eventos ocorre.

Outra opção é usar as funções da biblioteca padrão *setjmp* e *longjmp* para salvar e recuperar respectivamente um determinado estado do programa. A função *longjmp* é um *goto* não local pois passa o controle do programa para o ponto onde o último *setjmp* foi executado. Assim, na ocorrência de

um erro, pode-se usar o *longjmp* para ir para o local de tratamento definido em *setjmp*.

Ambas soluções com utilização das funções da biblioteca padrão requerem o imediato tratamento da exceção, mas uma solução concentra o tratamento em *signal* e a outra, a que usa *longjmp*, permite a sua localização em qualquer ponto do programa. Além disso, ambas soluções são complexas e difíceis de serem entendidas.

Como C não oferece um mecanismo específico para tratamento de exceções, fica a critério do programador implementar ou não o tratamento de exceções. Fica também a seu critério decidir qual a abordagem de tratamento será utilizada.

8.2 LPs com Mecanismos de Tratamento de Exceções

LPs com mecanismos de tratamento de exceções buscam garantir e estimular o tratamento das condições excepcionais sem que haja uma grande sobrecarga do texto do programa. Adicionalmente, elas oferecem um meio mais apropriado para se modularizar e reutilizar o código de tratamento de erros.

Linguagens como ADA, C++, JAVA e EIFFEL possuem mecanismos próprios de tratamento de exceções os quais facilitam a vida dos programadores tornando o código mais legível uma vez que separam o código executor da funcionalidade desejada do código de tratamento do erro.

Quando uma exceção ocorre ela necessita ser tratada. O bloco ou unidade de código que manipula a exceção é denominado tratador de exceção e a ação de indicar a ocorrência da exceção e transferir o controle para o tratador é denominada sinalização ou disparo da exceção. Tratadores de exceção se comportam como procedimentos chamados implicitamente pela ocorrência de uma exceção.

8.2.1 Tipos de Exceções

As LPs podem possuir exceções pré-definidas como parte da própria linguagem ou de sua biblioteca padrão (por exemplo, a exceção `overflow` normalmente é uma exceção pré-definida) e também permitir ao programador criar exceções específicas para uma aplicação ou biblioteca (por exemplo, uma aplicação de cadastro de produtos de uma loja pode necessitar disparar uma exceção sinalizando a condição de tamanho do estoque abaixo do normal).

Tal como quase tudo em C++ e JAVA, exceções são implementadas através de objetos. Para tanto, é permitido criar classes descritoras de exce-

ções. Embora sejam objetos de uma classe, exceções são objetos com uma característica diferenciada dos demais objetos. Elas podem ser lançadas para outras partes do programa seguindo um fluxo de controle distinto dos conhecidos até agora. Esse fluxo é determinado pelo mecanismo de tratamento de exceções de C++ e JAVA.

Por serem classes, as exceções podem (e devem) ser organizadas dentro de uma hierarquia de modo a descrever de forma natural o relacionamento entre os diferentes tipos de exceções. O exemplo 8.4 define uma hierarquia de exceções em C++.

```
class ErroMedico {};
class ErroDiagnostico: public ErroMedico {};
class ErroCirurgia: public ErroMedico {};
```

Exemplo 8. 4 - Hierarquia de Exceções em C++

C++ possui apenas oito exceções padrão organizadas em uma hierarquia própria, a partir da classe *exception*. A figura 8.1 apresenta um esquema parcial dessa hierarquia [STROUSTRUP, 2000].

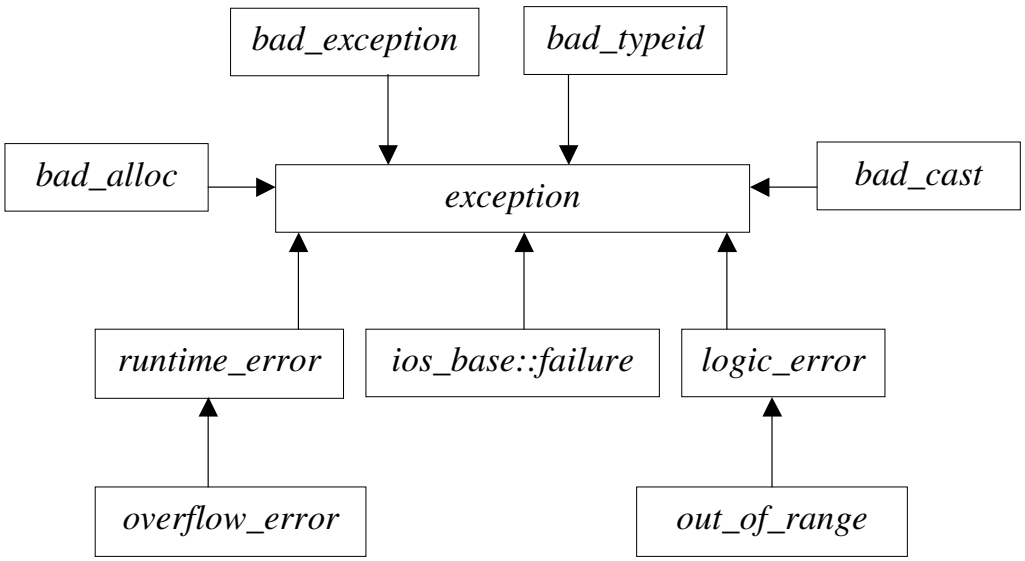

Figura 8. 1 - Hierarquia de Exceções Padrão em C++ (adaptado de [STROUSTRUP, 2000])

Como pode ser notado no exemplo 8.4, nem toda exceção de C++ é derivada de *exception*, embora as exceções padrão sejam. Além disso, nem toda exceção derivada de *exception* é uma exceção da biblioteca padrão. Os programadores podem acrescentar suas próprias exceções à hierarquia de *exception*.

Em JAVA, ao contrário de C++, toda exceção deve ser declarada como instância de uma subclasse de *java.lang.Throwable*, uma classe especial de JAVA. Essa classe age como mãe de todos os objetos lançados e cap-

turados usando o mecanismo de tratamento de exceções. Os principais métodos definidos na classe *java.lang.Throwable* recuperam a mensagem de erro associada com a exceção e imprimem a pilha rastreada mostrando onde ocorreu a exceção. Alguns métodos de *java.lang.Throwable* são listados a seguir.

> *void printStackTrace* **-** lista a seqüência de métodos chamados até o ponto onde a exceção foi lançada.
> *String getMessage* **-** retorna o conteúdo de um atributo com uma mensagem indicadora da exceção.
> *String ToString* **-** retorna uma descrição da exceção e de seu conteúdo.

Existem três categorias essenciais de exceções em JAVA: *Error*, *RuntimeException*, e *Exception*. *Error* e *Exception* são subclasses diretas de *Throwable* e *RuntimeException* é subclasse direta de *Exception*. A figura 8.2 ilustra essa hierarquia.

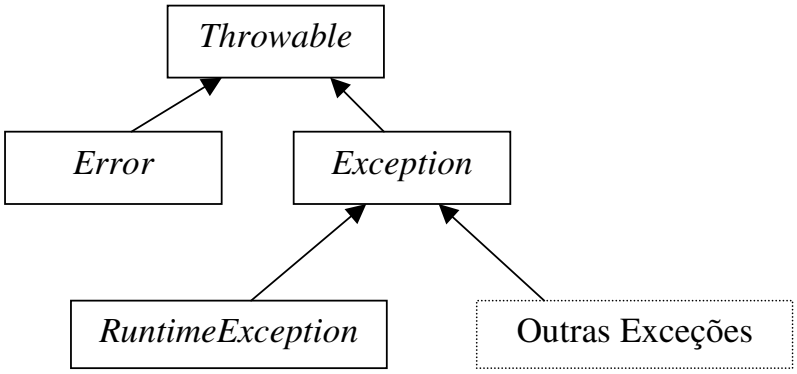

Figura 8. 2 - Principais Classes de Exceções em JAVA

A classe *Error* indica um problema grave de difícil (senão, impossível) recuperação. Dois exemplos são: *OutOfMemoryError* e *StackOverflowError*. Não se pode esperar de um programa nessas condições ser executado até o final. As exceções desse tipo são tratadas pelo próprio JAVA, implicando normalmente em terminação do programa. Elas não são usadas pelos programadores.

A classe *Exception* representa as exceções lançadas por métodos das classes da biblioteca padrão do JAVA e pelos métodos das classes dos aplicativos. Exceções das subclasses de *Exception* podem ser lançadas, capturadas e tratadas pelos programadores.

Dentre as exceções pré-definidas na biblioteca padrão, a classe especial *RuntimeException* se destaca por apresentar um comportamento diferenciado. Elas não necessitam ser lançadas explicitamente pelo programa. O sistema JAVA se incumbe de fazer isso quando elas ocorrem. Além disso, elas não necessitam ser tratadas obrigatoriamente pelo programador, em-

bora isto seja possível. Exemplos desse tipo de exceções são: *NullPointerException* e *IndexOutOfBoundsException*.

Se, por um lado, a existência desse tipo de exceções em JAVA poupa o programador de uma grande dose de trabalho (uma vez que ele não necessita identificar os pontos do programa onde a exceção ocorre, nem tampouco precisa fornecer tratadores para essas exceções), por outro lado, isso torna os programas um pouco menos confiáveis (uma vez que não se exige o tratamento dessas exceções).

As exceções dessa natureza normalmente indicam problemas sérios e devem implicar na terminação do programa. Contudo, ao incluir esse tipo de exceções com comportamento diferenciado na linguagem, oferece-se ao programador a opção de tratá-las e, de alguma maneira, continuar a execução do programa.

As outras subclasses de *Exception*, agrupadas na figura 8.2 pelo retângulo tracejado, apresentam o comportamento padrão, isto é, precisam ser lançadas explicitamente no código e, obrigatoriamente, necessitam ser tratadas (ou propagadas) pelo programador.

Assim, o programador não irá usar diretamente nem a classe *Throwable*, nem a classe *Error*. De fato, o programador normalmente utilizará uma classe ou criará uma subclasse na hierarquia definida a partir de *Exception*. O exemplo 8.5 mostra a definição de uma exceção em JAVA.

```
class UmaExcecao extends Exception {
  private float f;
  public UmaExcecao(String msg, float x) {
    super(msg);
    f = x;
  }
  public float contexto() {
    return f;
  }
}
```

Exemplo 8. 5 - Definição de Exceção em JAVA

A definição de uma exceção em JAVA ocorre como a definição de uma classe qualquer, como pode ser observado no exemplo 8.5. A exceção *UmaExcecao* possui um atributo específico *f*, um construtor e um método específico chamado *contexto*. O único requisito adicional de uma classe exceção é ser herdeira direta ou indireta de *Exception*. O construtor de *UmaExcecao* usa a palavra *super* para chamar o construtor de *Exception*.

8.2.2 Sinalização de Exceções

A sinalização de exceções pode ser feita automaticamente pelo próprio mecanismo de exceções ou pode ser feita explicitamente pelo próprio programador em casos específicos da aplicação. Enquanto no primeiro caso as exceções podem ser disparadas em qualquer ponto do programa, de maneira geral, no segundo caso elas só podem ser sinalizadas em trechos demarcados do programa.

Em C++ e JAVA, por exemplo, objetos de tipo exceção são criados através do uso da cláusula *throw* em trechos demarcados pelo bloco *try*. O exemplo 8.6 mostra o disparo de uma exceção em C++.

```
try {
  throw ErroMedico();
}
```

Exemplo 8. 6 - Sinalização de Exceção em C++

A única distinção da sinalização de exceções de C++ para JAVA é a requisição por parte da última do uso do operador *new* para a criação do objeto exceção. O exemplo 8.7 mostra o disparo de uma exceção em JAVA.

```
try {
  throw new Exception();
}
```

Exemplo 8. 7 - Sinalização de Exceção em JAVA

8.2.3 Tratadores de Exceções

Tratadores de exceção são trechos de código do programa responsáveis por tomar atitudes em resposta à ocorrência de uma exceção. Eles não são chamados explicitamente e, por isso, não precisam possuir nome. Por outro lado, é preciso haver uma forma de identificar o tratador a ser ativado quando uma determinada exceção ocorre. Isso normalmente é feito através do casamento com o nome ou tipo da exceção.

Em C++ e JAVA, tratadores de exceções são definidos através do uso de uma ou mais cláusulas *catch* colocadas imediatamente após o encerramento de um bloco *try*. Pode-se colocar quantos *catch* se desejar. A cada *catch* está associada uma classe de exceções, listada entre parênteses imediatamente após essa palavra, a qual permite manipular todas as exceções membros dessa classe. O código dos tratadores de exceção é colocado logo após a declaração da classe associada e também é delimitado por chaves. Apresenta-se no exemplo 8.7 um bloco *try-catch* completo em JAVA.

```
String n = "635";
String d = "27";
try {
   int num = Integer.valueOf(n).intValue();
   int den = Integer.valueOf(d).intValue();
   int resultado = num / den;
} catch (NumberFormatException e){
   System.out.println ("Erro na Formatacao");
} catch (ArithmeticException e){
   System.out.println ("Divisao por zero ");
}
```

Exemplo 8. 8 - Definição de Tratadores de Exceções em JAVA

As operações de atribuição às variáveis inteiras *num* e *den*, no bloco *try*, envolvem a conversão das strings *n* e *d* para valores inteiros. Caso a formatação dessas strings não fosse adequada, ocorreria uma exceção a qual seria capturada pelo tratador de exceções definido na primeira cláusula *catch*. Já a operação de divisão também poderia disparar uma exceção aritmética a qual seria capturada pelo tratador de exceções definido na segunda cláusula *catch*.

Observe como o mecanismo de exceções melhora a legibilidade do código. Em uma linguagem sem tratamento de exceções, após a chamada de cada operação de conversão seria necessário colocar testes para verificar se ocorreu uma exceção. Isso implicaria na colocação de código para verificação de erro entre as linhas de código responsáveis pela execução da funcionalidade desejada. Tal problema não ocorre com o mecanismo de exceções.

As operações de conversão de *String* podem disparar uma exceção do tipo *NumberFormatException* e a operação de divisão pode disparar uma *ArithmeticException*. O casamento do tipo da exceção disparada com o tipo declarado na cláusula *catch* determina qual dos tratadores deve ser executado.

O processo de casamento da exceção ocorrida com a exceção declarada no tratador é feito de maneira sucessiva, isto é, tenta-se casar a exceção com a declarada no primeiro *catch* e, se não houver casamento, tenta-se casar com a declarada no segundo, e assim sucessivamente. Uma vez que tenha havido o casamento em uma cláusula *catch*, o código correspondente a esta cláusula é executado. Após a execução do código de um tratador, nenhum outro será executado.

O casamento é feito com qualquer membro da classe declarada no tratador. Em outras palavras, o casamento acontece se a exceção ocorrida for uma instância da classe declarada ou de qualquer uma das suas subclas-

ses. Logo, nunca se deve declarar um tratador associado a uma classe antes dos tratadores associados a qualquer de suas subclasses. Caso contrário, os tratadores das subclasses nunca seriam ativados. No exemplo 8.9, em C++, os tratadores das exceções *ErroDiagnostico* e *ErroCirurgia* nunca são executados.

```
try {
  // codigo no qual varias excecoes podem ser sinalizadas
} catch (ErroMedico &e){
  // trata qualquer erro medico
} catch (ErroDiagnostico &e){
  // trata apenas erro de diagnostico
} catch (ErroCirurgia &e){
  // trata apenas erro de cirurgia
}
```

Exemplo 8. 9 - Distribuição Inapropriada de Tratadores

Isso ocorre porque o tratador da exceção *ErroMedico*, superclasse de *ErroDiagnostico* e *ErroCirurgia*, aparece antes dos tratadores dessas exceções. Assim, a ocorrência de qualquer exceção, seja *ErroDiagnostico*, *ErroCirurgia* ou qualquer outra subclasse de *ErroMedico* será sempre capturada pelo primeiro tratador. Para colocar os tratadores de *ErroDiagnostico* e *ErroCirurgia* em atividade, basta colocar o tratador de *ErroMedico* em último lugar.

Em JAVA, como a classe *Exception* é a classe mãe de todas as exceções, ela pode ser usada para capturar qualquer exceção. Como ela captura qualquer exceção, ela deve ser sempre colocada no último tratador associado ao bloco *try*. O exemplo 8.10 mostra como isso pode ser feito no exemplo da divisão.

```
try {
  int num = Integer.valueOf(n).intValue();
  int den = Integer.valueOf(d).intValue();
  int resultado = num / den;
} catch (NumberFormatException e){
  System.out.println ("Erro na Formatacao ");
} catch (ArithmeticException e){
  System.out.println("Divisao por zero");
} catch (Exception e){
  System.out.println ("Qualquer outra Excecao");
}
```

Exemplo 8. 10 - Captura de Qualquer Exceção em JAVA

Observe como o último tratador passa a ser responsável por tratar qualquer exceção disparada que não seja uma *NumberFormatException* ou *ArithmeticException*.

Como não existe em C++ uma superclasse única para todas as exceções, esse tipo de abordagem não pode ser usado. C++ utiliza o operador ... (reticências) para capturar as exceções de qualquer tipo. Observe como ele funciona no exemplo 8.11.

```
try {
  // codigo que dispara excecoes
} catch (ErroDiagnostico &e){
  // trata apenas erro de diagnostico
} catch (ErroCirurgia &e){
  // trata apenas erro de cirurgia
} catch (ErroMedico &e){
  // trata qualquer erro medico
} catch ( ... ) {
  // trata qualquer outra excecao
}
```

Exemplo 8. 11 - Captura de Qualquer Exceção em C++

O último *catch* captura qualquer exceção ocorrida no bloco *try* que não seja um *ErroDiagnostico*, *ErroCirurgia* ou *ErroMedico*. O tratador com reticências deve necessariamente ser colocado na última cláusula *catch*. Caso contrário, existirão tratadores que nunca serão ativados.

8.2.4 Propagação de Exceções

Eventualmente podem ocorrer situações nas quais o tratamento de exceção não deve ser realizado no mesmo bloco *try-catch* no qual a exceção ocorre. Nesses casos, a exceção deve ser propagada para ser tratada por tratadores associados a blocos *try-catch* mais externos.

Por isso, caso não se encontre o tratador correspondente à exceção no bloco *try-catch* no qual ela ocorre, ela é propagada para um nível mais externo e assim sucessivamente.

Quando uma função invocada lança uma exceção, cujo tratamento não é especificado na própria função, ela é retornada para a função chamadora (apesar da função chamada não possuir esse tipo de retorno!). Quando isso ocorre a função chamadora desvia o controle para o bloco de código de tratamento de exceções ou, caso não possua tratamento próprio, retorna a exceção para o nível mais externo. Esta propagação ocorre até a exceção ser capturada por um tratador ou até se atingir o nível mais externo do programa.

Caso uma exceção disparada não seja capturada por algum tratador, o procedimento mais comum é abortar a execução do programa, apresentando eventualmente uma mensagem indicando onde ocorreu a exceção. Em C++, se uma exceção não for capturada por algum tratador, a função *terminate* será chamada. A função *terminate* contém um ponteiro para função cujo valor `default` aponta para a função *abort*. Caso se queira especificar outra função de terminação, basta utilizar a função *set_terminate* para apontar *terminate* para essa outra função. Em JAVA, somente exceções *RuntimeException* não necessitam ser capturadas obrigatoriamente por tratadores. Nesse caso, a execução será interrompida e uma mensagem indicando informações sobre a ocorrência da exceção é apresentada. O exemplo 8.12 mostra como ocorre a propagação de uma exceção em JAVA.

```
public static void main(String[] args) {
  System.out.println("Bloco 1");
  try {
    System.out.println("Bloco 2");
    try {
      System.out.println("Bloco 3");
      try {
        switch(Math.abs(new Random().nextInt())%4+1){
          default:
          case 1: throw new NumberFormatException();
          case 2: throw new EOFException();
          case 3: throw new NullPointerException();
          case 4: throw new IOException();
        }
      } catch (EOFException e) {
        System.out.println("Trata no bloco 3");
      }
    } catch (IOException e) {
      System.out.println("Trata no bloco 2");
    }
  } catch (NullPointerException e){
    System.out.println("Trata no bloco 1");
  }
}
```

Exemplo 8. 12 - Propagação de Exceções entre Blocos *try-catch* de uma Mesma Função em JAVA

No programa do exemplo 8.12, existem três blocos *try-catch* aninhados. Dentro do mais interno deles quatro diferentes tipos de exceções podem ser lançadas. No caso de ser lançada a *EOFException*, o mecanismo de exceções tentará casar essa exceção com a do tratador do bloco mais in-

terno. Nesse caso, haverá casamento e o tratador do terceiro bloco será executado. Após a execução desse tratador, o programa se encerrará. Se for lançada a *IOException*, o mecanismo de exceções tentará casar essa exceção com a do tratador do bloco mais interno. Nesse caso, não haverá casamento e a exceção será propagada para o segundo bloco, no qual haverá casamento. O tratador do segundo bloco será executado e o programa se encerrará. Caso a exceção lançada seja *NullPointerException*, o processo se repetirá, mas somente haverá casamento no tratador do primeiro bloco. Por fim, se a exceção lançada for *NumberFormatException*, ela não se casará com nenhum tratador. Por se tratar de uma *RuntimeException*, que não exige tratamento, ela será propagada para fora do método *main* e o programa se encerrará mostrando a mensagem indicando onde ocorreu a exceção.

8.2.5 Relançamento de Exceções

Algumas vezes é preciso tratar parcialmente uma exceção em um determinado bloco *try* e relançá-la para ser tratada por outro tratador em um bloco *try* mais externo. Isso normalmente ocorre devido a falta de informação necessária no bloco *try* no qual ocorreu a exceção para promover o seu tratamento completo. Veja como isso pode ser feito em JAVA no exemplo 8.13.

```
public static void main(String[] args) {
  try {
    try {
      throw new IOException();
     } catch (IOException e) {
      System.out.println("Trata primeiro aqui");
      throw e;
    }
  } catch (IOException e) {
    System.out.println("Continua tratando aqui ");
  }
}
```

Exemplo 8. 13 - Relançamento de Exceção em JAVA

No exemplo 8.13, uma exceção é lançada no bloco *try* mais interno e é capturada pelo tratador desse bloco. Ao final da execução do código desse tratador, a exceção capturada é lançada novamente para ser capturada agora pelo tratador do bloco *try-catch* mais externo. Observe que o relançamento também foi realizado através do uso da cláusula *throw* dentro do tratador de exceções do bloco *try-catch* mais interno.

8.2.6 Especificação de Exceções

Em certas situações é interessante permitir a um subprograma lançar uma exceção sem, contudo, tratá-la. Nesse caso, quando a exceção ocorrer, ela será propagada para o trecho de código invocador do subprograma, o qual deverá tratá-la ou repropagá-la, conforme o caso.

Portanto, ao se escrever um programa no qual se precise chamar um determinado subprograma, é necessário saber quais possíveis exceções esse subprograma pode disparar. Assim, o usuário desse subprograma pode providenciar o tratamento ou repropagação das exceções disparáveis.

Uma forma possível de identificar quais exceções podem ser disparadas por um subprograma seria consultar o código fonte do subprograma. Essa forma é inconveniente por duas razões. Primeiramente se trata de um procedimento muito trabalhoso, uma vez que seria necessário, além de consultar o código do subprograma, consultar os códigos dos subprogramas chamados por esse subprograma e assim sucessivamente. A segunda razão decorre do fato dos códigos fontes dos subprogramas não estarem disponíveis para os usuários desse subprograma em muitas situações.

Outra possibilidade seria deixar para o compilador indicar quais exceções precisam ser tratadas. Essa possibilidade também não é adequada porque o programador necessitaria compilar o código para então saber quais exceções precisa tratar. Além disso, o processo de verificação de exceções na compilação de um arquivo usuário de subprogramas compilados previamente em outros arquivos é de difícil execução quando os fontes desses módulos não estão disponíveis.

A solução frequentemente adotada para informar ao usuário do subprograma quais exceções deve tratar requer a especificação dessas exceções no cabeçalho do subprograma. Agora, para o usuário saber quais exceções um dado subprograma pode disparar e não tratar, basta olhar para o cabeçalho desse subprograma, não sendo mais necessário consultar o código fonte que o implementa.

Além disso, esse procedimento pode facilitar o processo de verificação das exceções durante a compilação. Analisando apenas o código de um subprograma, o compilador pode indicar facilmente se ele dispara exceções não tratadas em sua implementação e não especificadas em seu cabeçalho. Do mesmo modo, o compilador pode indicar se um subprograma chama outro subprograma e não trata ou repropaga as exceções disparáveis pelo subprograma chamado.

Em C++, *throw* também pode ser usado para indicar aos usuários de uma função quais exceções podem ser lançadas. Existem três especificações possíveis, ilustradas no exemplo 8.14.

```
void f() throw (A,B,C);
void g() throw();
void h();
```

Exemplo 8. 14 - Formas de Especificação de Exceção em C++

A primeira forma ilustrada no exemplo 8.14 indica que a função *f* pode lançar exceções do tipo *A*, *B* ou *C*. A segunda maneira indica que a função *g* não pode lançar exceções. O terceiro modo indica que a função *h* pode lançar qualquer exceção.

O compilador C++ não verifica se o compromisso assumido na especificação está sendo cumprido. Portanto, se a função *f* lançar uma exceção D, isso só será tratado em tempo de execução. Nesse caso, a função *unexpected* será executada. A função *set_unexpected* pode ser usada para determinar o que *unexpected* deve fazer. O exemplo 8.15 mostra como isso pode ser feito.

```
class ErroI { };
class ErroII { };
void f () throw (ErroI) {
  throw ErroII();
}
void exc() {
  cout <<"erro nao esperado";
  exit(1);
}
main() {
  set_unexpected(exc);
  try {
    f ();
  } catch (ErroI){
    cout<<"erro em f";
  }
}
```

Exemplo 8. 15 – Descumprimento de Especificação de Exceção pela Própria Função em C++

No exemplo 8.15, quando a função *f* for executada, será disparada a exceção *ErroII*, a qual não é tratada ou repropagada pela função *f*. Isso causará a execução da função *exc*.

C++ também não obriga a função chamadora a tratar todas as exceções possíveis de serem geradas pela função chamada. O código do exemplo 8.16 será aceito pelo compilador e interrompido em tempo de execução.

```
class ErroI { };
class ErroII { };
```

```
void f () throw (ErroI) {
  throw ErroI();
}
void exc(){
  cout <<"erro nao esperado";
  exit(1);
}
main() {
  set_unexpected(exc);
  try {
    f ();
  } catch (ErroII) {
    cout<<"erro em f";
  }
}
```

Exemplo 8. 16 - Descumprimento de Especificação de Exceção pela Função Invocante em C++

JAVA exige a especificação das exceções não tratadas nos cabeçalhos dos métodos, os quais não sejam *RuntimeException*, através do uso da cláusula *throws*. O exemplo 8.17 ilustra o uso dessa cláusula em JAVA.

```
public static void main(String[] args) throws IOException {
  System.out.println("Bloco 1");
  try {
    System.out.println("Bloco 2");
    try {
      System.out.println("Bloco 3");
      try {
        switch(Math.abs(new Random().nextInt())%4+1){
          default:
          case 1: throw new NumberFormatException();
          case 2: throw new EOFException();
          case 3: throw new NullPointerException();
          case 4: throw new IOException();
        }
      } catch (EOFException e) {
        System.out.println("Trata no bloco 3");
      }
    } catch (NumberFormatException e) {
      System.out.println("Trata no bloco 2");
    }
  } catch (NullPointerException e){
    System.out.println("Trata no bloco 1");
  }
```

```
}
```

Exemplo 8. 17 - Especificação de Exceções em JAVA

Observe a declaração da cláusula *throws* logo após a lista de parâmetros do método *main* no exemplo 8.17. Ela indica que, dentro do método *main*, pode ocorrer uma exceção *IOException*, a qual não será tratada. Portanto, a ocorrência dessa exceção implicará na sua propagação para o código no qual o método for chamado. Como *IOException* não é uma *RuntimeException*, se a cláusula *throws* não fosse utilizada, o compilador reportaria um erro de não tratamento de exceção.

A imposição de JAVA para que todo método tenha de indicar as exceções propagáveis em seu cabeçalho serve para alertar explicitamente os potenciais usuários daquele método sobre tal possibilidade. Esse é outro aspecto positivo de JAVA. Isso facilita o trabalho dos usuários pois não mais precisam ler a implementação do método para descobrirem quais exceções podem ser propagadas por ele. O exemplo 8.18 mostra a reimplementação do exemplo 8.17 enfocando a propagação de exceções entre métodos.

```
public static void main(String[] args) throws IOException {
  System.out.println("Bloco 1");
  try {
    primeiro();
  } catch (NullPointerException e){
    System.out.println("Trata no bloco 1");
  }
}
public static void primeiro() throws IOException,
                                     NullPointerException {
  System.out.println("Bloco 2");
  try {
    segundo();
  } catch (NumberFormatException e) {
    System.out.println("Trata no bloco 2");
  }
}
public static void segundo() throws IOException,
                                    NullPointerException {
  System.out.println("Bloco 3");
  try {
    switch(Math.abs(new Random().nextInt())%4+1) {
      default:
      case 1: throw new NumberFormatException();
      case 2: throw new EOFException();
```

```
        case 3: throw new NullPointerException();
        case 4: throw new IOException();
      }
    } catch (EOFException e) {
      System.out.println("Trata no bloco 3");
    }
  }
```

Exemplo 8. 18 - Propagação de Exceções entre Métodos em JAVA

Ao contrário de *IOException* e *NullPointerException*, *NumberFormatException* não necessita ser especificada no cabeçalho da função *segundo* porque ela é uma *RuntimeException*.

8.2.7 Modos de Continuação Após o Tratamento de Exceções

Quando um programa executa sem a ocorrência de exceções ele segue o seu fluxo de controle normal. Contudo, para onde deve ir o fluxo de controle após a ocorrência e o tratamento de exceções? Para responder a essa pergunta é importante ter em mente o fato de, em muitos casos, o local de tratamento da exceção se encontrar bem distante do ponto no qual ela ocorre.

Em geral, existem dois modos de continuação do fluxo de controle após o tratamento das exceções:

- Terminação: assume o erro como crítico, não existindo condições de retornar ao ponto no qual a exceção foi gerada. O controle retorna para um ponto mais externo do programa. Nessa alternativa todas as unidades na pilha de registros de ativação, a partir da unidade na qual ocorreu a exceção até a unidade anterior a que o tratador de exceção foi executado, são encerradas. A execução do programa continua na unidade na qual o tratador foi encontrado, após a região de tratamento.
- Retomada: assume o erro como corrigível e a execução pode retornar para o bloco no qual ocorreu a exceção. Portanto, o retorno é feito para o bloco gerador do erro. Embora, à primeira vista, essa solução pareça ser a mais apropriada, uma vez que a exceção seria supostamente tratada, a experiência tem indicado uma baixa efetividade dessa alternativa.

Como consequência da baixa efetividade do modelo de retomada, a maioria das LPs tem adotado atualmente o modelo de terminação. Essa é a postura de C++ e JAVA. O exemplo 8.19, em C++, mostra como funciona o modo de continuação por terminação.

```
class ErroI { };
```

```
class ErroII { };
class ErroIII { };
void f () throw (ErroI) {
  throw ErroI();
}
main() {
  cout << "comeca aqui\n";
  try {
     cout << "passa por aqui\n";
     try {
      f ();
     } catch (ErroIII){
       cout<<"não passa por aqui\n";
     }
       cout<<"também não passa por aqui\n";
     } catch (ErroI){
       cout<<"erro I em f\n";
     } catch (ErroII){
       cout<<"não passa por aqui\n";
     }
   cout << "termina aqui\n";
}
```

Exemplo 8. 19 - Modo de Continuação por Terminação em C++

Durante a execução de *f*, a exceção *ErroI* é disparada. Ela não é capturada pelo tratador associado ao bloco *try-catch* no qual ocorreu a exceção (o único tratador existente é para *ErroIII*). A exceção é propagada então para o bloco *try-catch* mais externo, onde é capturada pelo primeiro tratador. Ao final da execução desse tratador, o programa continua a partir do final do bloco de código do último tratador de exceções associado ao bloco *try-catch* mais externo.

Muito embora o modo de continuação por retomada não seja efetivo na maioria das ocorrências de exceções, em algumas situações, pode ser possível e desejável tratar um erro e tentar repetir a execução do trecho de código no qual ele ocorre. O exemplo 8.20 mostra como isso pode ser feito em JAVA.

```
import java.util.*;
public class Retomada {
  static class ImparException extends Exception {}
  public static void main(String[] args) {
    boolean continua = true;
    Random r = new Random();
    while (continua) {
```

```
        continua = false;
        try {
          System.out.print ("Escolha um numero par: ");
          int i = r.nextInt();
          if (i%2 != 0) throw new ImparException();
        } catch(ImparException e) {
          System.out.println("Tente novamente!!!");
          continua = true;
        }
      }
    }
}
```

Exemplo 8. 20 - Implementação de Modo de Continuação por Retomada em JAVA

A abordagem apresentada no exemplo 8.20 só pode ser utilizada quando o tratamento da exceção ocorre no mesmo bloco *try-catch* no qual a exceção ocorreu.

Em certas situações pode ser interessante executar um trecho de código associado ao bloco *try-catch* independentemente da ocorrência do tratamento da exceção no mesmo bloco *try-catch* ou não. Em JAVA, usa-se a cláusula *finally* colocada após o último tratador de exceção para obter essa funcionalidade. O exemplo 8.21 ilustra o uso desse mecanismo.

```
public class Sempre {
  public static void main(String[] args) {
    System.out.println("Bloco 1");
    try {
      System.out.println("Bloco 2");
      try {
        throw new Exception();
      } finally {
        System.out.println("finally do bloco 2");
      }
    } catch(Exception e) {
      System.out.println("Excecao capturada");
    } finally {
      System.out.println("finally do bloco 1");

  }
}
```

Exemplo 8. 21 - Cláusula *finally* em JAVA

No exemplo 8.21 tanto o *finally* do bloco mais interno quanto o do mais externo serão executados. Como pode ser observado no bloco mais inter-

no, é possível ter blocos *try* sem associação de cláusulas *catch.* Porém, não pode haver blocos *try* sem que haja pelo menos uma associação de cláusulas *catch* ou *finally*.

A cláusula *finally* é muito usada quando se necessita restabelecer um estado de um objeto independentemente da ocorrência e propagação de exceções ou não. No exemplo 8.22, o carro tem de ser desligado após a movimentação independentemente de ter havido fogo ou superaquecimento.

```
public class CarroBomba {
  class SuperAquecimentoException extends Exception {}
  class FogoException extends Exception {}
  Random r = new Random();
  public void ligar() {}
  public void mover() throws SuperAquecimentoException,
                             FogoException {
    float temperatura = r.nextFloat();
    if (temperatura > 100.0) {
      throw new SuperAquecimentoException();
    }
    throw new FogoException();
  }
  public void desligar() {}
  public static void main(String[] args) {
    try {
      CarroBomba c = new CarroBomba();
      try {
          c.ligar();
          c.mover();
      } catch(SuperAquecimentoException e) {
          System.out.println("vai fundir o motor!!!");
      } finally {
          c.desligar();
      }
    } catch (FogoException e) {
      System.out.println ("vai explodir!!!");
    }
  }
}
```

Exemplo 8. 22 - Cláusula *finally* para Garantir Procedimento de Finalização em JAVA

Na ocorrência de uma *FogoException* o carro só é desligado por causa do uso da cláusula *finally*. Se essa cláusula não existisse e o comando de desligar fosse colocado após o término do bloco *try-catch* mais interno, quando ocorresse uma *FogoException,* o fluxo de controle do programa

seria direcionado diretamente para o tratador de exceções do bloco *try-catch* mais externo.

A existência da cláusula *finally* no mecanismo de exceções em JAVA trouxe um problema para a linguagem: uma exceção pode ocorrer e ser ignorada [ECKEL, 2002]. O exemplo 8.23 ilustra uma situação na qual isso pode ocorrer.

```
public class Perda {
  class InfartoException extends Exception {
    public String toString() { return "Urgente!"; }
  }
  void infarto() throws InfartoException {
    throw new InfartoException ();
  }
  class ResfriadoException extends Exception {
    public String toString() { return "Descanse!"; }
  }
  void resfriado() throws ResfriadoException {
    throw new ResfriadoException ();
  }
  public static void main(String[] args) throws Exception {
    Perda p = new Perda();
    try {
      p.infarto();
    } finally {
      p.resfriado();
    }
  }
}
```

Exemplo 8. 23 - Perda de Exceção em JAVA

No exemplo 8.23, a exceção *InfartoException* não é tratada no bloco *try-catch* no qual ocorre. Assim, o fluxo de execução se direciona diretamente ao bloco da cláusula *finally*, no qual uma nova exceção (*ResfriadoException*) ocorre. Com isso, a primeira exceção é ignorada e a nova exceção passa a ser o foco de tratamento e propagação.

8.2.8 Exceções e Polimorfismo

A combinação dos mecanismos de orientação a objetos, tais como herança e amarração tardia de tipos, com os mecanismos de tratamento de exceções aumenta a complexidade da linguagem. De um modo geral, são estabelecidas regras na linguagem para garantir o uso apropriado do mecanismo de exceções. JAVA, por exemplo, estabelece as seguintes regras:

1. Os construtores podem adicionar novas exceções a serem propagadas às declaradas no construtor da superclasse.
2. Os construtores devem necessariamente propagar as exceções declaradas no construtor da superclasse usado. Se o construtor da superclasse pode propagar exceções, o da subclasse também deverá propagá-las pois o último necessariamente chama o primeiro.
3. Métodos declarados na superclasse não podem ter novas exceções propagadas. Afinal, o código usuário dos objetos da superclasse deve ser capaz de lidar com os objetos da subclasse. Se novas exceções pudessem ser propagadas, elas poderiam ser disparadas e aquele código não as trataria.
4. Não é obrigatório propagar as exceções dos métodos da superclasse. Isso não é problema porque o código usuário do método simplesmente tratará uma exceção que não ocorrerá quando ele for chamado por um objeto da subclasse.
5. Os métodos sobrescritos podem disparar exceções que sejam subclasses das exceções propagadas na superclasse. Isso não é problema porque o tratador da superclasse captura todas as exceções que sejam subclasses da classe associada ao tratador.

O exemplo 8.24 ilustra como o mecanismo de exceções se integra à orientação a objetos em JAVA. No exemplo são definidas inicialmente algumas classes de exceções organizadas em hierarquias. É definida também uma classe abstrata *Dirigir*.

As especificações do construtor dessa classe e o método *irTrabalhar* indicam a propagação da exceção *InfraçãoTrânsito*. Contudo, os corpos desses métodos não disparam exceção alguma. Isso não causa maiores problemas porque o código usuário desses métodos terá necessariamente de tratar ou propagar a exceção especificada. Por outro lado, isso é muito útil porque permite a programação do código usuário sem que a implementação desses métodos esteja concluída, aumentando assim a produtividade da equipe de programação, a qual pode trabalhar em paralelo desenvolvendo o código da classe e o código usuário. Além disso, essa característica também permite preparar uma versão inicial do programa para uma futura reimplementação desses métodos (agora disparando a exceção) sem que a inserção do código lançador da exceção implique na necessidade de alteração do código usuário.

```
class InfracaoTransito extends Exception {}
class ExcessoVelocidade extends InfracaoTransito {}
class AltaVelocidade extends ExcessoVelocidade {}
class AvancarSinal extends InfracaoTransito {}
class Acidente extends Exception {}
class Batida extends Acidente {}
```

```
abstract class Dirigir {
  Dirigir() throws InfracaoTransito { }
  void irTrabalhar () throws InfracaoTransito {}
  abstract void viajar() throws ExcessoVelocidade, AvancarSinal;
  void caminhar() {}
}
interface Perigo {
  void irTrabalhar () throws Batida;
  void congestionamento() throws Batida;
}
public class DirecaoPerigosa extends Dirigir implements Perigo {
  DirecaoPerigosa() throws Batida, InfracaoTransito {}
  DirecaoPerigosa (String s) throws ExcessoVelocidade,
                                    InfracaoTransito {}
  // void caminhar() throws AltaVelocidade {}
  // public void irTrabalhar() throws Batida {}
  public void irTrabalhar() {}
  public void congestionamento() throws Batida {}
  void viajar() throws AltaVelocidade {}
  public static void main(String[] args) {
    try {
      DirecaoPerigosa dp = new DirecaoPerigosa ();
      dp.viajar ();
    } catch(AltaVelocidade e) {
    } catch(Batida e) {
    } catch(InfracaoTransito e) {}
    try {
    Dirigir d = new DirecaoPerigosa();
      d.viajar ();
    } catch(AvancarSinal e) {
    } catch(ExcessoVelocidade e) {
    } catch(Batida e) {
    } catch(InfracaoTransito e) {}
  }
}
```

Exemplo 8. 24 - Exceções e Polimorfismo

No exemplo 8.24 também são definidas a interface *Perigo* e uma subclasse de *Dirigir*, denominada *DirecaoPerigosa*, que implementa *Perigo*. As especificações dos métodos de *Perigo* indicam a propagação da exceção *Batida*.

Como *DirecaoPerigosa* é subclasse de *Dirigir*, seus construtores devem necessariamente propagar a exceção propagada no construtor de *Dirigir.*

Como pode ser observado no exemplo, os construtores de *DirecaoPerigosa* propagam outras exceções além da propagada pelo construtor de *Dirigir*.

A implementação dos métodos *caminhar* e *irTrabalhar* de *DirecaoPerigosa* se encontram comentadas porque elas indicam a propagação de exceções não listadas na especificação desses métodos na superclasse *Dirigir*. Isso geraria erro de compilação, caso não estivessem como comentários. Uma implementação válida de *irTrabalhar* é incluída no exemplo para mostrar que métodos sobrescritos não necessitam propagar as exceções propagadas na implementação desses métodos na superclasse.

A implementação do método *viajar* de *DirecaoPerigosa* mostra que métodos sobrescritos podem disparar exceções de subclasses da exceção propagada no método da superclasse.Por fim, o método *main* mostra a diferença nas exigências de tratamento de exceções quando se utiliza referências para *DirecaoPerigosa* e para *Dirigir*. No primeiro caso, só é necessário tratar as exceções propagadas pelo construtor de *DirecaoPerigosa* e pelo método *viajar* dessa classe. No segundo caso, mesmo que o objeto criado seja um *DirecaoPerigosa*, o compilador exige que sejam tratadas as exceções propagadas pelo método *viajar* de *Dirigir*. Isso ocorre porque diferentes implementações do método *viajar* podem ser utilizadas, dependendo da classe do objeto referenciado.

8.3 Considerações Finais

Nesse capítulo foi visto como funcionam os mecanismos de tratamento de exceções e quais as suas vantagens. Eles melhoram a legibilidade dos programas pois separam o código com a funcionalidade principal do programa do código responsável pelo tratamento de exceções. Além disso, os mecanismos de tratamento de exceções também aumentam a confiabilidade e robustez dos programas, uma vez que normalmente requerem o tratamento obrigatório das exceções ocorridas e porque promovem a idéia de recuperação dos programas mesmo na presença de situações anômalas.

Outros benefícios dos mecanismos de tratamento de exceções são incentivar o reuso e a modularidade do código responsável pelo tratamento. Em particular, em linguagens orientadas a objetos, exceções são instâncias de classes, o que faz com que essa parte do programa herde todas as propriedades relativas ao reuso e a modularidade fornecidas pela programação orientada a objetos.

Tratadores de exceção oferecem um mecanismo de fluxo de controle diferenciado do oferecido por subprogramas. Ao se chamar um subprograma, o fluxo de controle é transferido para o subprograma e, após sua execução, o fluxo de controle retorna para o ponto onde o subprograma foi

chamado. Já no caso do lançamento de uma exceção, o fluxo de controle é transferido para um ponto, na cadeia dinâmica formada pelos sucessivos blocos *try-catch*, invocados para chegar ao ponto de ocorrência da exceção, no qual existe um tratador capacitado a lidar com aquela exceção. Também diferentemente do fluxo de subprogramas, após o tratamento da exceção, o fluxo de controle passa para o final do bloco *try-catch* no qual a exceção foi tratada, ao invés de retornar para o ponto no qual a exceção foi disparada. A figura 8.3 ilustra as diferenças entre o fluxo de controle de subprogramas e tratamento de exceções.

No fluxo de controle normal do programa, indicado pela sequência de setas 1-2-3-6, não há ocorrência de exceções. Quando a exceção *A* ocorre, o fluxo de controle é indicado pela sequência 1-2-4-6. Já na ocorrência da exceção *B*, o fluxo de controle é indicado pela sequência 1-2-5. Nesse último caso, o fluxo de controle não é retornado para a *funçãoI*, que chamou a *funçãoII*, na qual ocorreu a exceção.

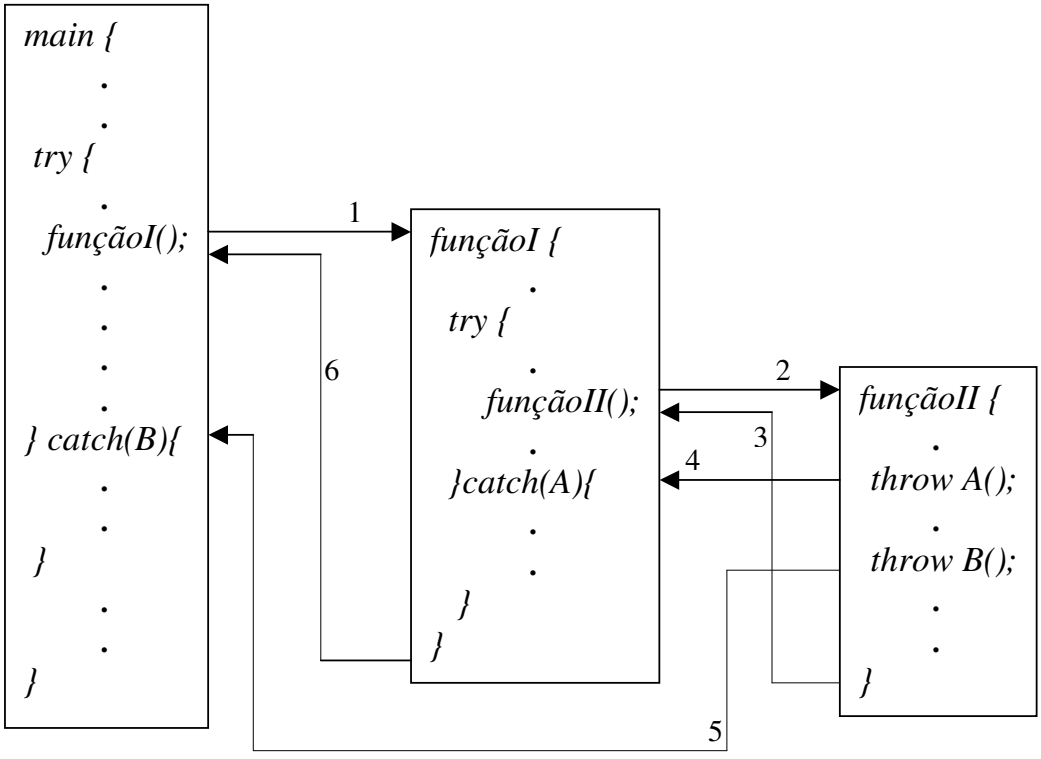

Figura 8. 3 - Fluxo de Controle de Exceções

A combinação dos mecanismos de orientação a objetos com os de tratamento de exceções traz uma maior complexidade para a linguagem, tornando mais difícil o aprendizado da linguagem e podendo induzir a ocorrência de erros de programação. Questões sobre como a linguagem integra o tratamento de exceções com os conceitos de polimorfismo, herança, inicialização e destruição de objetos devem ser esclarecidas antes do uso

do tratamento de exceções associado a esses conceitos. Por exemplo, em C++, é preciso saber que na ocorrência de uma exceção, todos objetos criados pelo bloco de código responsável pela exceção são automaticamente destruídos, desde que seus construtores tenham sido completados. Se a exceção for gerada dentro de um construtor, então recursos já alocados por esse construtor não serão liberados.

A inclusão do mecanismo de tratamento de exceções em uma LP pode reduzir a eficiência computacional dos programas nessa linguagem. Isso ocorre principalmente porque os programas em linguagens não possuidoras desse mecanismo normalmente não fazem tratamento de exceções algum. Por exemplo, um programa com uso de vetores em JAVA é menos eficiente do que um em C porque o mecanismo de exceções de JAVA verifica automaticamente a legalidade dos acessos aos índices do vetor, o que não ocorre em C.

Ao se comparar o tratamento de exceções em C++ e JAVA, observa-se que essas linguagens implementam mecanismos semelhantes, mas com diferenças significativas entre eles.

Inicialmente, o mecanismo de exceções não foi implementado em C++. Posteriormente, esse mecanismo foi incluído inspirado na implementação de ADA. Para manter a compatibilidade com versões anteriores de C++ e para não comprometer a eficiência computacional dos programas, em comparação aos programas em linguagem C, não foi adotada uma postura mais rigorosa no mecanismo de tratamento de exceções de C++.

Os pontos seguintes sintetizam fragilidades do mecanismo de C++:

- número reduzido de exceções pré-definidas na biblioteca padrão;
- as funções não são obrigadas a especificar as exceções que podem propagar;
- não detecção em tempo de compilação da quebra de compromisso com uma dada especificação (lançamento de exceção não prevista na especificação);
- não existe obrigação de explicitar a exceção relançada para o nível superior.

JAVA baseou-se em C++ para implementar um mecanismo de tratamento de exceções bastante seguro, claro e que favorece a construção de software confiável. Esse mecanismo torna praticamente impossível a construção de software sem utilização de um bom esquema de tratamento de exceções.

JAVA oferece vários tipos de exceções pré-definidas verificadas automaticamente, tais como, exceções capazes de verificar o acesso indevido a vetores e a realização inapropriada da operação de estreitamento.

JAVA também apresenta a vantagem de constatar em tempo de compilação a utilização indevida do mecanismo de tratamento de exceções. Caso uma exceção disparada não seja tratada ou propagada, o compilador indicará um erro de não tratamento dessa exceção.

Apesar disso, JAVA deixa uma brecha para o não tratamento de exceções do tipo *RuntimeException*. Para exceções desse tipo, o compilador não indica erro quando elas não são tratadas ou propagadas. Embora isso sacrifique um pouco do rigor do mecanismo de exceções de JAVA, essa opção foi escolhida para livrar o programador do desconforto de ficar especificando vários tipos de exceções freqüentes, reduzindo assim a redigibilidade dos programas em JAVA.

Uma linguagem com um mecanismo de exceções bastante interessante é EIFFEL [MEYER, 1988]. Essa LP baseia seu mecanismo de tratamento de exceção na filosofia de projeto por contrato. Nessa concepção, clientes e fornecedores honram seus compromissos de modo a evitar o surgimento de exceções.

Uma rotina cliente deve garantir o cumprimento de pré-condições ao chamar uma rotina. Essa, por sua vez, deve garantir o cumprimento de pós-condições. A rotina cliente se beneficia das pós-condições e a rotina servidora se beneficia das pré-condições. Esse contrato é especificado formalmente na rotina servidora. Além das asserções de pré-condições e pós-condições, podem ser especificadas também asserções invariantes que devem ser válidas antes e após a execução da rotina. Violações dessas asserções ativam o mecanismo de tratamento.

Assim, EIFFEL apresenta um mecanismo mais estruturado se comparado com os oferecidos por JAVA e C++, uma vez que as exceções são inseridas no programa como efeito do método de programação por contrato. Em contraste, as exceções de JAVA e C++ são inseridas nos programas sem uma disciplina de programação, isto é, sem um critério sistemático sobre quando e como é importante utilizar o mecanismo de exceções. Outra diferença significativa em EIFFEL é a possibilidade de se adotar tanto o modo de continuação por retomada quanto por terminação.

8.4 Exercícios

1. Explique as vantagens de se possuir um mecanismo de exceções incorporado a LP. Ilustre essas vantagens apresentando exemplos de código com funcionalidade equivalente em C e JAVA.

2. Erros ordinários podem ser tratados no mesmo ambiente no qual foram identificados. Cite vantagens no uso de um mecanismo de exceções como o de JAVA para o tratamento desse tipo de erro.

3. Analise o seguinte trecho de programa em C:

```
int leArquivo ( int v [] ) {
   char *n;
   int cod;
   FILE *p;
   cod = leNomeArq (n);
   if (cod == 0) {
      printf ("nome invalido");
      return -1;
   }
   cod = abreArq (n, p);
   if (cod == 0) {
      printf("arquivo  inexistente");
     return -1;
   }
   cod = carregaArq (p, v, 100);
   if (cod == 0) return -2;
   cod = fechaArq (p);
   if (cod == 0) return -3;
   return 0;
}

  main ( ) {
     int cod;
     int vet [100];
     cod = tentaLer (vet);
     switch (cod) {
        case 0: break;
        case -1:
           printf ("erro de nome");
           break;
        case -2:
           printf ("erro de carga");
           break;
        case -3:
           printf("erro de fechamento");
      };
      ordena (vet);
      imprime (vet);
  }
```

```
int tentaLer (int v [ ] ) {
   int cod;
   do {
      cod = leArquivo (v);
      if (cod == -1) {
         if  (!continua ( ))
            return cod;
      } else {
         return cod;
      }
   } while (1);
}
```

Considere que as funções *leNomeArq*, *abreArq*, *carregaArq* e *fechaArq* retornam 0 (zero) se não forem bem sucedidas e 1 (um), caso contrário. Considere também que a função *continua* pergunta ao usuário se ele deseja tentar novamente e retorna 1 (um) em caso afirmativo e 0 (zero) em caso negativo. Refaça esse programa usando o mecanismo de tratamento de exceções de C++. Na versão em C++, as funções *leNomeArq*, *abreArq*, *carregaArq* e *fechaArq* retornam *void* mas disparam respectivamente as seguintes exceções *nomeExc, arqExc, cargaExc* e *fechaExc*. Compare as duas soluções em termos de redigibilidade e legibilidade, justificando.

4. Considere o seguinte esqueleto de programa em C++:

```
class B {
   int k;
   float f;
public:
   void f1() {
      …
      try { …
         throw k;
         …
         throw f;
         …
      }catch (float){
         …
      }
      …
   }
}
```

```
class A {
   int j;
   float g;
   B b;
public:
   void f2() {
      …
      try { …
         try { …
            b.f1();
            …
            throw j;
            …
            throw g;
            …
         }catch(int){
            …
         }
         …
      }catch (float){
         …
      }
      …
   }
}
```

```
main ( ) {
   A a;
   …
   a.f2 ( );
   …
}
```

Indique, para cada possível exceção disparada, o local onde ela será tratada.

5. Explique os mecanismos oferecidos por C, C++ e JAVA para o tratamento de exceções. Enfoque sua explicação na comparação dos seguintes aspectos: a) obrigatoriedade ou não do tratamento de exceções por um usuário de uma função que dispara exceções; b) existência de exceções disparadas pelo próprio mecanismo de exceções da linguagem (tal como quando ocorre divisão por zero).

6. Considere o seguinte trecho de código em JAVA.

```
class InfracaoTransito extends Exception {}
class ExcessoVelocidade extends InfracaoTransito {}
class AltaVelocidade extends ExcessoVelocidade {}
class Acidente extends Exception {}
class Defeito extends Exception {}
abstract class Dirigir {
  Dirigir() throws InfracaoTransito {}
  void irTrabalhar () throws InfracaoTransito {}
  abstract void viajar() throws
     ExcessoVelocidade, Defeito;
  void caminhar() {}
}
public class DirecaoPerigosa extends Dirigir {
  DirecaoPerigosa() throws Acidente {}
  void caminhar() throws AltaVelocidade {}
  public void irTrabalhar() {}
  void viajar() throws AltaVelocidade {}
  public static void main(String[] args) {
    try {
         DirecaoPerigosa dp = new DirecaoPerigosa ();
         dp.viajar ();
    } catch(AltaVelocidade e) {
    } catch(Acidente e) {
    } catch(InfracaoTransito e) {
   }
    try {
         Dirigir d = new DirecaoPerigosa();
         d.viajar ();
    } catch(Defeito e) {
    } catch(ExcessoVelocidade e) {
    } catch(Acidente e) {
    } catch(InfracaoTransito e) {}
  }
}
```

O trecho de código acima apresenta dois erros identificáveis em tempo de compilação. Que erros são esses? Justifique sua resposta.

7. Apresente as abordagens que linguagens como C podem usar para lidar com erros em situações nas quais não há conhecimento suficiente para tratar o erro no local onde ele ocorre. Essas abordagens devem passar as informações do erro para um contexto mais externo para que ele possa ser tratado. Enumere e explique os problemas com cada uma delas.

8. Embora o esqueleto de programa JAVA seguinte seja válido sintaticamente, ele não se comporta apropriadamente em uma situação específica (considere que uma operação só é completada se realizada sem ocorrência de exceção). Identifique que situação é essa. Justifique sua resposta. Reformule o programa para que esse problema seja corrigido.

```
public class DefeitoCarro {
  class SemArranque extends Exception {}
  class SuperAquecim extends Exception {}
  public void ligar() throws SemArranque {
    …
    if (…) throws SemArranque();
      …
  }
  public void mover() throws SuperAquecim {
    …
    if (…) throws SuperAquecim();
        …
  }
  public void desligar() {}
  public static void main(String[] args) {
    DefeitoCarro c = new DefeitoCarro ();
    try {
      c.ligar();
      c.mover();
    } catch(SemArranque e) {
      System.out.println("tem de empurrar!!!");
    } catch(SuperAquecim e) {
      System.out.println("vai fundir!!!");
    } finally {
      c.desligar();
    }
  }
}
```

9. Ao se compilar o seguinte programa JAVA ocorre um erro de compilação relacionado ao uso de exceções. Identifique qual é esse erro, atentando para o fato que nenhuma exceção disparável no programa é herdeira de *RuntimeException*, e diga como você o corrigiria. Considerando que o problema foi corrigido tal como você propôs, mostre o que será impresso durante a execução do programa. Mostre o que seria impresso caso o valor atribuído a variável *i* do método *main* fosse *2*. Mostre, por fim, o que seria impresso se o valor de *i* fosse *3*.

```
class testaExcecoes {
    public static void main(String[] args) {
      int i = 1;
      try {
        primeiro(i);
       System.out.println("depois de primeiro");
      } catch (NullPointerException e){
        System.out.println("trata no primeiro bloco");
      }
      System.out.println("saiu do primeiro bloco");
    }
    public static void primeiro(int i) throws NullPointerException {
      try {
        segundo(i);
        System.out.println("depois de segundo");
      } catch (IOException e) {
        System.out.println("trata no segundo bloco");
      }
      System.out.println("saiu do segundo bloco");
    }
    public static void segundo(int i) throws NullPointerException {
      try {
        switch(i) {
          default:
          case 1: throw new IOException();
          case 2: throw new EOFException();
          case 3: throw new NullPointerException();
        }
        System.out.println("depois do switch");
      } catch (EOFException e) {
        System.out.println("trata no terceiro bloco");
      }
      System.out.println("saiu do terceiro bloco");
    }
}
```

10. Existem posições controversas com relação a incorporação de um mecanismo rigoroso de tratamento de exceções em LPs. Enquanto alguns defendem o rigor, outros preferem um mecanismo mais flexível. Por exemplo, alguns programadores JAVA são defensores do uso exclusivo das exceções da classe *RuntimeException* ou de suas subclasses. Indique a característica dessas classes de exceção que justifica essa postura. Apresente argumentos favoráveis e contrários a posição adotada por esses programadores.