Resolução dos Exercícios do Capítulo VIII

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.

As vantagens de se ter um mecanismo de exceções incorporado à LP são: a melhoria da legibilidade dos programas, pois separam o código com a funcionalidade principal do programa do código responsável pelo tratamento de exceções; o aumento da confiabilidade dos programas, uma vez que normalmente requerem o tratamento obrigatório das exceções ocorridas e porque promovem a idéia de recuperação dos programas mesmo em situações anômalas; e o incentivo a reutilização, redigibilidade 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 à reutilização e à modularidade fornecidas pela programação orientada a objetos).

Exemplo:

```
// Em C

int trata_formatacão_errada (void) {
    printf ("Formatacao Errada");
    return -99;
}
int lenum () {
    char s[30];
    gets(s);
    if (isNumber(s))
            return atoi(s);
    else
            return trata_formatacao_errada();
}
int trata_divisão_zero (void) {
    printf ("Divisão por zero.");
    return -98;
}
int divideInteiros (int numerador, int denominador) {
    if (denominador == 0)
            return tratar_divisão_zero ();
    else
            return numerador / denominador;
}
main() {
    //...
    int num, den, resultado;
    num = lenum();
    if (num != -99) {
            den = lenum();
            if (den != -99) {
```

```
                    resultado = divideInteiros (num, den);
                    if (resultado != -98) {
                    //..
}

// Em JAVA

// ...
int num, den, resultado;
try {
    int num = Integer.valueOf(n).intValue();
    int den = Integer.valueOf(d).intValue();
    int resultado = num / den;
}   catch (NumberFormatException e) {
            System.out.println (e);
}   catch (ArithmeticException e) {
            System.out.println (e);
}
// ...
```

Os trechos de código em C e JAVA, mostrados no exemplo acima, cumprem basicamente a mesma funcionalidade. Ambos lêem duas strings, tentam convertê-las em números, caso tenham sucesso, dividem a primeira pela segunda. Caso não tenham sucesso nas operações de conversão de string para números e de divisão, são mostradas mensagens correspondentes e o programa prossegue a partir do fim do trecho.

A abordagem de JAVA é mais legível do que a de C pois existe uma separação do código do programa que implementa a funcionalidade (trecho escrito dentro do boloco *try*) do código de tratamento de erros. Isso não ocorre em C, uma vez que após a chamada das operações de leitura dos número e divisão é necessário incluir um teste para verificar se ocorreu a condição excepcional.

A abordagem de JAVA também é mais confiável porque não requer que o programador lembre, identifique e teste todas as possíveis condições causadoras de exceções. Isso já é feito pelas funções e operações utilizadas (observe em particular que não é necessário testar se o denominador é zero antes da operação de divisão). Além disso, caso o programador se esqueça de tratar as exceções que possam ocorrer em um determinado trecho, o compilador JAVA pode lembrá-lo (em caso de exceções que não sejam *RuntimeException* ou promover um tipo particular de tratamento, encerrando o programa e imprimindo dados que facilitem a identificação de onde e porque ocorreu a situação excepcional).

Por fim, pode-se observar que a abordagem de JAVA é muito mais redigível que a de C. Precisa-se escrever muito menos para se conseguir a mesma funcionalidade. Em parte, isso ocorre pela existência de funções pré-definidas na biblioteca de classes. No entanto, grande parte desta melhor redigibilidade é obtida pela não necessidade de testar as operações antes ou depois de executá-las para verificar a ocorrência de exceções (por exemplo, não é preciso testar duas vezes se a operação de conversão foi bem sucedida, como ocorre no bloco principal em C). Adicionalmente, pode-se ainda se beneficiar da orientação a objetos para reutilizar tratadores de exceções (por exemplo, a função *toString* das classes *NumberFormatException* e

ArithmeticException é reutilizada no momento de imprimir as mensagens nos tratadores de exceção do exemplo em 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.

Em JAVA, a ocorrência de erros é sinalizada através de exceções, isto é, objetos especiais que carregam informação sobre o tipo de erro detectado. As vantagens de se utilizar um mecanismo como o de JAVA para tratar erros ordinários são a melhoria da legibilidade dos programas, pois separa o código com a funcionalidade principal do programa do código responsável pelo tratamento de exceções; o aumento da confiabilidade e robustez dos programas, uma vez que normalmente requer o tratamento obrigatório das exceções ocorridas e porque promove a idéia de recuperação dos programas mesmo em situações anômalas; e o incentivo a reutilização 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 à reutilização e a modularidade fornecidas pela programação orientada a objetos).

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;
}
```

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

```
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);
}
```

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.

Programa em C++:

```
void tentaLer () throw (nomeExc, arqExc) {
        do {
                try {
                        leNomeArq(n);
                        abreArq(n, p);
                } catch (nomeExc e) {
                        cout << "nome invalido\n";
                        if (!continua()) throw e;
                } catch (arqExc e) {
                        cout << "arquivo inexistente\n";
                        if (!continua()) throw e;
                }
        while (1);
}
main () {
        int vet [100];
        char *n;
        FILE *p;
        try {
```

```
            tentaLer();
            carregaArq (p, v, 100);
            fechaArq (p);
        } catch (nomeExc) {
            cout << "erro de nome\n";
        } catch (arqExc) {
            cout << "erro de nome\n";
        } catch (cargaExc) {
            cout << "erro de carga\n";
        } catch (fechaExc) {
            cout << "erro de fechamento\n";
        }
        ordena (vet);
        imprime (vet);
    }
```

No programa escrito em C++ a legibilidade é melhorada, pois o código com a funcionalidade principal do programa é separado do código responsável pelo tratamento de exceções. A redigibilidade é melhorada porque não se necessita utilizar códigos numéricos e testá-los em vários pontos do programa.

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.

- Exceção ***k*** → O mecanismo de exceções tentará casar essa exceção com a do tratador do bloco definido na função f1 da classe B. Nesse caso, não haverá casamento e a exceção será propagada para o bloco mais interno definido na função f2 da classe A. O tratador desse bloco será executado.
- Exceção ***f*** → O mecanismo de exceções tentará casar essa exceção com a do tratador do bloco definido na função f1 da classe B. Nesse caso, haverá casamento e esse tratador será executado.
- Exceção ***j*** → O mecanismo de exceções tentará casar essa exceção com a do tratador do bloco definido na função f2 da classe A. Nesse caso, haverá casamento e esse tratador será executado.
- Exceção ***g*** → O mecanismo de exceções tentará casar essa exceção com a do tratador do bloco definido na função f2 da classe A. Nesse caso, não haverá casamento e a exceção será propagada para o bloco mais externo definido na função f2 da classe A. O tratador desse bloco será executado.

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).

A linguagem de programação C não oferece qualquer mecanismo específico para o tratamento de exceções, ficando a critério do programador implementá-lo ou não. As linguagens C++ e JAVA possuem mecanismo de tratamento de exceções. Enquanto o mecanismo de C++ não obriga o programador a tratar as exceções e não possui exceções disparadas pelo próprio mecanismo de exceções da linguagem, o mecanismo

de JAVA oferece vários tipos de exceções detectadas automaticamente e força o programador a tratar grande parte delas.

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.

JAVA estabelece a seguinte regra para garantir o uso apropriado do mecanismo de exceções: os construtores devem necessariamente propagar as exceções declaradas no construtor da superclasse. Se o construtor da superclasse pode propagar exceções, o da subclasse também deverá propagá-las pois o último necessariamente chama o primeiro. Assim, o primeiro erro é que o construtor de *DirecaoPerigosa* não propaga a exceção *InfracaoTransito*, quando deveria fazê-lo, pois *DirecaoPerigosa* é subclasse de *Dirigir* e nesta classe o construtor propaga a exceção *InfracaoTransito*.

O outro erro identificável em tempo de compilação é a implementação do método *caminhar* na classe *DirecaoPerigosa*, pois sua implementação dispara a exceção *AltaVelocidade*, que não está listada na especificação desse método na superclasse *Dirigir*. O compilador de JAVA impede que isso possa ser feito porque no caso de se chamar o método *caminhar* de *DirecaoPerigosa* através de uma referência a superclasse *Dirigir*, isso poderia ocasionar o disparo da exceção *AltaVelocidade*, sem que ela fosse devidamente tratada.

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.

As abordagens utilizadas são: o retorno do código de erro indicando a exceção ocorrida em uma variável global, no resultado da função ou em um parâmetro específico.
A opção de utilizar uma variável global 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 dessa ser tratada efetivamente.
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).
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 de a 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.

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();
    }
  }
}
```

A situação ocorre quando a exceção *SemArranque* é disparada na função *ligar*. Se essa exceção ocorre, a função *ligar* não é executada, pois o fluxo do programa é direcionado para o tratador de exceções e, depois para o bloco *finally*. Nesse caso, a função *desligar* será executada, embora o carro não esteja ligado.

<u>*Programa reformulado:*</u>

```
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 ();
                    try {
                            c.mover();
                    } catch (SuperAquecim e) {
                            System.out.println ("vai fundir!!!");
                    } finally {
                            c.desligar ();
                    }
            } catch (SemArranque e) {
```

```
                System.out.println (“tem de empurrar!!!”);
            }
        }
    }
```

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");
    }
}
```

Java exige a especificação das exceções não tratadas nos cabeçalhos dos métodos, as quais não sejam *RuntimeException*, utilizando a cláusula throws.

O erro neste programa é a não identificação da exceção *IOException* no cabeçalho da função *segundo*. Ela não é tratada nessa função e também não é uma *RuntimeException*, portanto, deve ser especificada para que possa ser propagada para o código no qual o método *segundo* foi chamado.

Correção do código:

public static void segundo (int i) throws NullPointerException, IOException { ...

- Execução para i = 1:

  trata no segundo bloco
  saiu do segundo bloco
  depois de primeiro
  saiu do primeiro bloco

- Execução para i = 2:

  trata no terceiro bloco
  saiu do terceiro bloco
  depois de segundo
  saiu do segundo bloco
  depois de primeiro
  saiu do primeiro bloco

- Execução para i = 3:

  trata no primeiro bloco
  saiu do primeiro 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.

A classe *RuntimeException* apresenta um comportamento diferenciado. As exceções dessa classe não necessitam ser tratadas obrigatoriamente pelo programador (o compilador não indica erro quando elas não são tratadas ou propagadas).
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 fornecer tratadores para essas exceções ou especificar sua propagação), por outro lado, isso torna os programas um pouco menos confiáveis (uma vez que não se exige o tratamento dessas exceções, o programador pode se esquecer de fazê-lo).