Capítulo 9

Exceções

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 destas 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 estas condições especiais não sejam tratadas o bloco de código onde 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 que ocorre 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 que requer uma ação específica imediata.

Como vimos acima, muitos tipos de erros podem causar exceções: problemas que variam desde erros sérios de hardware, tal como uma falha no disco rígido, a erros simples de programação, tal como tentar acessar um indíce inexistente de um vetor.

Erros causam exceções, mas podem existir exceções que não são erros. Por exemplo, numa função que lê dados de um arquivo, a chegada ao final do arquivo é uma condição excepcional que pode ser considerada como exceção. Normalmente, este tipo de exceção ocorre em subprogramas que podem produzir múltiplos resultados e alteram o fluxo normal de execução do programa.

Linguagens de programação que não oferecem nenhum mecanismo específico para tratamento de exceções acabam produzindo programas menos confiáveis e mais obscuros. Os programas se tornam menos confiáveis porque a linguagem não obriga que se faça tratamento das possíveis exceções. O código fica mais obscuro porque se mistura código relacionado com a funcionalidade desejada com o código responsável por tratar as exceções. Java procura reduzir esses problemas oferecendo um mecanismo para tratamento de exceções, o qual será apresentado em maior detalhe neste capítulo.

9.1 – Tipos de Exceções

Tal como quase tudo em Java, exceções são implementadas através de objetos. Para tanto, é permitido criar classes que descrevem exceções. 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.

Embora sejam objetos de uma classe, exceções são objetos que possuem 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. Este fluxo é determinado pelo mecanismo de tratamento de exceções de Java. Para caracterizar este comportamento diferenciado, toda exceção deve ser declarada como instância de uma subclasse de `java.lang.Throwable`, uma classe especial de Java. Esta classe age como mãe de todos os objetos que são lançados e capturados 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. Por exemplo:

- **void printStackTrace() :** lista a seqüência de métodos chamados até o ponto onde a exceção foi lançada.
- **String getMessage():** Retorna o conteúdo de um atributo que contém 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 abaixo ilustra esta hierarquia:

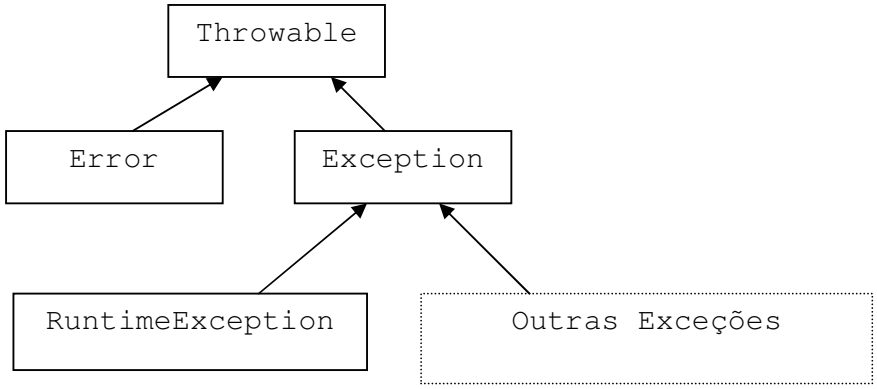

Figura 9. 1 - Classes Principais de Exceções em JAVA

A classe `Error` indica um problema grave de difícil (senão, impossível) recuperação. Dois exemplos são: `OutOfMemoryError` e `StackOverflowError`. Não se espera que um programa nestas condições seja executado até o final. As exceções deste tipo são tratadas pelo próprio Java, implicando normalmente em terminação do programa. Elas não são usadas pelos programadores.

A classe `Exception` representa as exceções que podem ser lançadas por métodos das classes da biblioteca padrão do Java e pelos métodos das classes dos aplicativos. Exceções das subclasses de `Exception` serão 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 pelo programador, embora isto seja possível. Exemplos deste tipo de exceções são: `NullPointerException` e `IndexOutOfBoundsException`. Se, por um lado, a existência deste 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 estas exceções), por outro lado, isso torna os programas um pouco menos confiáveis (uma vez que não se exige o tratamento destas exceções). As exceções desta natureza normalmente indicam problemas sérios e devem implicar na terminação do programa. Contudo, ao incluir este 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 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`.

9.2 – Captura e Tratamento de Exceções

A utilização do mecanismo de exceções em Java é obrigatório para as exceções que não são `RuntimeException`. Qualquer trecho de código que possa lançar uma exceção só poderá ser invocado em um bloco de código supervisionado. Um bloco de código supervisionado é iniciado pela cláusula *try* e é delimitado por chaves.

Java exige que exceções lançadas em um bloco supervisionado devam ser capturadas por um tratador de exceções associado ao bloco supervisionado ou a um outro bloco supervisionado mais externo que contenha o bloco onde ela ocorre. Se a exceção lançada não é tratada dentro dos blocos supervisionados de um método, Java requer que esta exceção seja propagada pelo método. Estas exigências e o conceito de blocos supervisionados são um dos aspectos positivos do mecanismo de exceções de Java pois garantem que o programador trate ou repropague explicitamente qualquer exceção que possa ocorrer induzindo, assim, a construção de software robusto.

Tratadores de exceção são definidos através de cláusulas *catch* colocadas imediatamente após o término de um bloco supervisionado. Pode-se colocar quantos *catch* se desejar. Cada *catch* permite manipular todas as exceções que são membros de uma classe que lhe é associada (listada entre parênteses imediatamente após a cláusula *catch*). O código dos tratadores de exceção é colocado logo após a declaração da classe associada e também é delimitado por chaves. A forma sintática dos blocos try-catch pode ser vista a seguir:

```
try {
// código que pode lançar uma exceção específica
} catch (ExcecaoA exca) {
// código executado se ExcecaoA é diparada
} catch (ExcecaoB excb) {
// código executado se ExcecaoB é diparada
} …
} catch (Exception e) {
// código executado se qualquer outra exceção é
// lançada
}
```

É importante esclarecer que 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. Ressalta-se que, após a execução do código de um tratador, nenhum outro será executado. Portanto, ao final da execução do tratador onde houve o casamento, o programa continuará a partir do final do bloco de código do último tratador de exceções associado ao bloco supervisionado.

Lembre-se que 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 subclasses. Logo, nunca se deve declarar um tratador associado a uma classe antes dos tratadores associados a qualquer de suas subclasses. Caso contrário, teríamos tratadores que nunca seriam ativados. Por exemplo, 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 do bloco superviosionado. O exemplo seguinte ilustra o uso do bloco try-catch:

```
// Excecao.java
public class Excecao {
  public static void main(String[] args) {
    String s;
    int a, b;
    try {
       s = Console.readString();
       a = Integer.valueOf(s).intValue();
       s = Console.readString();
       b = Integer.valueOf(s).intValue();
       resultado = a / b;
    } catch (ArithmeticException e){
       System.out.println("Divisao por zero");
    } catch (NumberFormatException e){
       System.out.println ("Erro na Formatacao ");
    } catch(Exception e) {
      System.out.println("Qualquer outra Excecao");
    }
  }
}
```

Exemplo 9. 1 - Bloco try-catch

No exemplo 9.1, qualquer que seja a exceção que ocorra no bloco try, se não for capturada pelos dois primeiros tratadores, necessariamente o será no terceiro. Se invertêssemos a ordem dos tratadores, os tratadores específico de `ArithmeticException` e de `NumberFormatException` nunca seriam ativados.

9.3 – Lançamento Explícito de Exceções

Uma exceção pode ser lançada implícita ou explicitamente. Exceções são lançadas implicitamente quando ocorre algum tipo de erro em tempo de execução que é identificado pelo sistema Java.

Contudo, exceções também podem ser condições excepcionais que ocorrem no programa, mas que não podem ser identificadas pelo sistema Java. Neste caso, o programador deve lançar explicitamente a exceção no ponto onde elas ocorrem através do uso da cláusula *throw*. O uso desta cláusula pode ser observado no exemplo seguinte:

```
// Excecao1.java
public class Excecao1 {
  public static void main(String[] args) {
    try {
       throw new Exception ("Uma primeira excecao");
```

```
    } catch(Exception e) {
      System.out.println("Excecao capturada");
    }
  }
}
```

Exemplo 9. 2 - A claúsula throws

Observe o uso do operador *new* associado a cláusula *throw*. Você pode perceber, portanto, que um novo objeto da classe `Exception` está sendo criado e imediatamente lançado para ser capturado pelo tratador de exceções associado.

9.4 – Propagação de Exceções

Caso não se encontre o tratador de exceções correspondente no nível do lançamento, a exceção é propagada para um nível mais externo e assim sucessivamente. Caso nenhum tratador a capture, então, na saída do programa será executado o tratamento padrão de exceções de Java (interrupção da execução e apresentação de mensagem indicando onde ocorreu a exceção). O exemplo seguinte mostra como ocorre a propagação de uma exceção:

```
// Excecao2.java
import java.io.*;
import java.util.*;
public class Excecao2 {
  static Random rand = new Random();
  static int pRand(int mod) {
     return Math.abs(rand.nextInt()) % mod + 1;
  }
  public static void main(String[] args) {
    System.out.println("Primeiro try");
    try {
      System.out.println("Segundo try ");
      try {
        System.out.println("Terceiro try ");
        try {
          switch(pRand(4)) {
            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 terceiro try");
        }
      } catch (IOException e) {
         System.out.println("Trata segundo try ");
      }
```

```
    } catch (NullPointerException e){
      System.out.println("Trata primeiro try");
    }
  }
}
```

Exemplo 9. 3 - Propagação de Exceções

No programa do exemplo 9.3, existem três blocos supervisionados aninhados. Dentro do mais interno deles podem ser lançadas 4 diferentes tipos de exceções. No caso de ser lançada a `EOFException`, o mecanismo de exceções tentará casar esta exceção com a do tratador do bloco mais interno. Neste caso, haverá casamento e o tratador do terceiro bloco será executado. Após a execução deste tratador, o programa se encerrará. Se for lançada a `IOException`, o mecanismo de exceções tentará casar esta exceção com a do tratador do bloco mais interno. Neste caso, não haverá casamento e a exceção será propagada para o segundo bloco, onde 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.

Se alterássemos o exemplo 9.3, tratando agora `NumberFormatException` ao invés de `IOException`, ao compilarmos o programa ocorreria um erro. Como `IOException` não é uma `RunTimeException`, Java exige que ela seja tratada explicitamente pelo programador. Como não existiria nenhum tratador para esta exceção, seria ocasionado um erro. Este erro poderia ser contornado através do uso da cláusula *throws*, que indica que uma determinada exceção pode ocorrer no método, mas que não é tratada ali. O exemplo 9.4 ilustra o uso desta cláusula:

```
// Excecao2a.java
import java.io.*;
import java.util.*;
public class Excecao2a {
  static Random rand = new Random();
  static int pRand(int mod) {
     return Math.abs(rand.nextInt()) % mod + 1;
  }
  public static void main(String[] args)
                throws IOException{
    System.out.println("Primeiro try");
    try {
      System.out.println("Segundo try ");
      try {
        System.out.println("Terceiro try ");
        try {
          switch(pRand(4)) {
            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 terceiro try");
        }
      } catch (NumberFormatException e) {
         System.out.println("Trata segundo try ");
      }
    } catch (NullPointerException e){
      System.out.println("Trata primeiro try");
    }
  }
}
```

Exemplo 9. 4 - A Cláusula throws

Observe a declaração da cláusula *throws* logo após a lista de parâmetros do método `main()`. Ela está indicando que dentro do método `main()` pode ocorrer uma exceção `IOException` que não será tratada e que, portanto, será propagada para o código onde o método for chamado.

Todo método é obrigado a indicar em seu cabeçalho as exceções que poderá propagar, isto é, aquelas que podem ocorrer no corpo do método que não são tratadas localmente por ele. No exemplo, o método `main()` teve de declarar que a exceção `IOException` não é tratada pois, em caso contrário, haveria um erro de compilação. Essa origatoriedade serve para alertar explicitamente os potenciais usuários daquele método sobre tal possibilidade. Este é outro aspecto positivo de Java pois facilita o trabalho dos clientes que não precisam ler a implementação do método para descobrirem quais exceções podem ser propagadas por ele. O exemplo seguinte mostra a reimplementação do exemplo 9.4 enfocando a propagação de exceções entre métodos:

```
// Excecao2b.java
import java.io.*;
import java.util.*;
public class Excecao2b {
  static Random rand = new Random();
  static int pRand(int mod) {
     return Math.abs(rand.nextInt()) % mod + 1;
  }
  public static void main(String[] args)
                throws IOException {
    System.out.println("Primeiro try");
    try {
      primeiro();
    } catch (NullPointerException e){
      System.out.println("Trata primeiro try");
    }
  }
  public static void primeiro() throws IOException,
                           NullPointerException {
    System.out.println("Segundo try ");
    try {
      segundo();
```

```
    } catch (NumberFormatException e) {
         System.out.println("Trata segundo try ");
    }
  }
  public static void segundo() throws IOException,
                              NullPointerException {
    System.out.println("Terceiro try ");
    try {
          switch(pRand(4)) {
            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 terceiro try");
    }
  }
}
```

Exemplo 9. 5 - Propagação de Exceções entre Métodos

9.5 – Relançamento de Exceções

Algumas vezes, é preciso tratar parcialmente uma exceção em um determinado bloco supervisionado e relançá-la para ser tratada por outro tratador em um bloco mais externo. Isto tipicamente ocorre devido a falta de toda a informação necessária no nível corrente para promover o tratamento completo da exceção. Veja como isso pode ser feito no exemplo a seguir:

```
// Excecao3.java
import java.io.*;
public class Excecao3 {
  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 aqui ");
    }
  }
}
```

Exemplo 9. 6 - Relançamento de uma Exceção

Neste exemplo, uma exceção é lançada no bloco mais interno, que é capturada pelo tratador deste bloco. Ao final da execução do código deste tratador, a exceção capturada é lançada novamente para ser captura agora pelo tratador do bloco mais externo. Observe

que o relançamento também foi realizado através do uso da cláusula *throw* dentro do tratador de exceções do bloco supervisionado mais interno.

9.6 – Criação de Exceções

Além de criar objetos a partir das classes de exceções existentes, o programador pode também criar novas classes de exceções. Estas novas classes se comportam como as demais classes, podendo o programador incluir novos atributos e métodos ou mesmo sobrescrevê-los. Ao se criar uma nova classe de exceções, deve-se capturar todas as informações importantes do contexto onde a exceção ocorreu e armazená-las nos seus atributos. Assim, quando uma exceção deste novo tipo for capturada por um tratador, ele poderá utilizar os atributos e métodos da classe para obter informações do contexto onde a exceção ocorreu e para lhe dar um tratamento apropriado.

Para criar uma nova classe de exceções basta simplesmente herdar da classe `Exception` ou de suas herdeiras. Veja o exemplo:

```
class MinhaExcecao extends Exception {
  public MinhaExcecao() {}
  public MinhaExcecao(String msg) { super(msg); }
  public MinhaExcecao(String msg, int x) {
     super(msg); i = x;
  }
  public int val() { return i; }
  private int i;
}
public class TestaExcecao {
  public static void f() throws MinhaExcecao {
    System.out.println("Disparou MinhaExcecao");
    throw new MinhaExcecao();
  }
  public static void main(String[] args) {
    try {
      f();
    } catch(MinhaExcecao e) {
      System.out.println(e.val());
      e.printStackTrace();
    }
  }
}
```

Exemplo 9. 7 - Criando uma Exceção

A vantagem de poder criar novas classes de exceções é que se tem toda a liberdade para se definir novos tipos de exceções e seu comportamento de acordo com o aparecimento da demanda. De fato, é impossível antever todos os tipos de exceções (e seus comportamentos) que poderiam ser necessárias. Com a abordagem adotada, esse processo ocorre incrementalmente em reposta a necessidade. Assim, ao se criar uma aplicação ou uma biblioteca, normalmente são definidas um conjunto de exceções que podem ser lançadas pelos métodos das classes desta aplicação ou da biblioteca.

9.7 – A Cláusula finally

Em certas situações queremos que um trecho de código associado ao bloco supervisionado seja executado independente da ocorrência ou não de exceção. Em Java, usamos a cláusula *finally* colocada após o ultimo tratador de exceção para obtermos esta funcionalidade:

```
public class Sempre {
  public static void main(String[] args) {
    System.out.println("Primeiro try");
    try {
      System.out.println("Segundo try");
      try {
        throw new Exception();
      } finally {
        System.out.println("finally do segundo try");
      }
    } catch(Exception e) {
      System.out.println("excecao capturada");
    } finally {
      System.out.println("finally do primeiro try");
    }
  }
}
```

Exemplo 9. 8 - A Cláusula finally

No exemplo 9.8, tanto o *finally* do bloco mais interno quanto o do mais externo serão executados. Como pode ser observado no bloco mais interno, é possível ter blocos supervisionados sem associação de cláusulas *catch*. Porém, não pode haver blocos supervisionados 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 ou não de exceções. No exemplo a seguir, o carro tem de ser desligado após a movimentação independentemente de ter havido superaquecimento:

```
public class CarroBomba {
  class SuperAquecimentoException extends Exception {}
  public void ligar() {}
  public void mover()
                throws SuperAquecimentoException {
    String temperatura = Console.readString();
    if (temperatura.equals("anormal")) {
      throw new SuperAquecimentoException();
    }
  }
  public void desligar() {}
  public static void main(String[] args) {
    CarroBomba c = new CarroBomba();
    try {
      c.ligar();
      c.mover();
```

```
    } catch(SuperAquecimentoException e) {
      System.out.println("vai explodir!!!");
    } finally {
      c.desligar();
    }
  }
}
```

Exemplo 9. 9 - Restabelecendo um Estado do Objeto com finally

A existência da cláusula finally no mecanismo de exceções em java trouxe um problema para a linguagem: uma exceção pode ocorrer e ser perdida sem que tenha sido tratada. O exemplo seguinte ilustra a situação onde isso ocorre:

```
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 9. 10 - Perda de Exceção

No exemplo, a exceção `InfartoException` não é tratada no bloco supervisionado onde ela ocorre. Assim, o fluxo de execução se direciona diretamente ao bloco da cláusula *finally*, onde 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.

9.8 – Retomada

Na maioria das vezes em que ocorre uma exceção, o trecho de código onde ela ocorreu não será repetido após o tratamento. Contudo, em algumas situações, pode ser possível e desejável tratar um erro e tentar repetir a execução do trecho de código onde ele ocorre. Costuma-se chamar este procedimento de retomada. O exemplo seguinte mostra como isso pode ser feito:

```
public class Retomada {
  static class NaoPositivoException
                              extends Exception {}
  public static void main(String[] args) {
    boolean continua = true;
    while (continua) {
      continua = false;
      try {
        System.out.print (
          "Entre um inteiro positivo: ");
        int i = Console.readInteger();
        if (i <= 0) throw new NaoPositivoException();
      } catch(NaoPositivoException e) {
        System.out.println("Tente novamente!!!");
        continua = true;
      }
    }
  }
}
```

Exemplo 9. 11 - Retomada

9.9 – Herança com Exceções

Com o mecanismo de exceções, os métodos de uma classe podem disparar exceções. Quando se cria uma subclasse desta classe, seus métodos são herdados e podem ser sobrescritos. Existem regras que estabelecem como isso deve ser feito, tais como:

1. Os construtores podem adicionar novas exceções a serem propagadas àquelas declaradas no construtor da superclasse.
2. Os construtores devem necessariamente propagar as exceções declaradas no construtor da superclasse que será usado. Se o construtor da superclasse pode propagar exceções, o da subclasse também poderá propagá-las pois o último necessariamente chama o primeiro.
3. Métodos declarados na superclasse não podem ter novas exceções propagadas. A razão para isto é que o código que lida com 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 que usa o método simplesmente tratará uma exceção que não ocorrerá quando ete 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.

```
class InfracaoTransito extends Exception {}
class ExcessoVelocidade extends InfracaoTransito {}
class AvancarSinal extends InfracaoTransito {}

abstract class Dirigir {
  Dirigir() throws InfracaoTransito {
   // Nao e' obrigado disparar a excecao
```

```
  }
  void irTrabalhar () throws InfracaoTransito {}
  abstract void viajar() throws ExcessoVelocidade,
                                AvancarSinal;
  void caminhar() {} // Nao dispara excecao
}

class Acidente extends Exception {}
class Batida extends Acidente {}
class AltaVelocidade extends ExcessoVelocidade {}

interface Perigo {
  void irTrabalhar () throws Batida;
  void congestionamento() throws Batida;
}

public class DirecaoPerigosa extends Dirigir
                             implements Perigo {

  // novas excecoes podem ser adicionadas ao
  // construtor, mas é obrigatório lidar com as
  // excecoes do construtor da superclasse
  DirecaoPerigosa() throws Batida, InfracaoTransito {}
  DirecaoPerigosa (String s) throws ExcessoVelocidade,
                                InfracaoTransito {}

  // metodos devem ser declarados como na superclasse
  //! void caminhar() throws AltaVelocidade {}
  // erro de compilacao

  // interfaces nao podem adicionar excecoes
  // aos metodos da superclasse
  //! public void irTrabalhar() throws Batida {}

  // adicao pode ser feita se metodo nao existe
  // na superclasse
  public void congestionamento() throws Batida {}

  // Nao e obrigatorio disparar a excecao
  // do metodo da superclasse
  public void irTrabalhar() {}

  // metodos sobrescritos podem disparar excecoes de
  // subclasses da classe declarada no metodo
  // da superclasse
  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) {}
    // AvancarSinal nao e' disparada em viajar()
    // da subclasse
    try {
      Dirigir d = new DirecaoPerigosa();
      d.viajar ();
      // No caso de upcast deve-se tratar excecoes
      // da superclasse
    } catch(AvancarSinal e) {
    } catch(ExcessoVelocidade e) {
    } catch(Batida e) {
    } catch(InfracaoTransito e) {}
  }
}
```

Exemplo 9. 12 - Herança com Exceções

Uma última observação é que um método pode não disparar uma excessão declarada como sendo propagada por ele. Isso não causa maiores problemas porque o código que usa o método terá de tratar a exceção (apenas não será usado). Java permite isso para que se possa deixar para uma futura reimplementação do método a inserção do código que dispara a exceção.