Linguagens de Programação
Conceitos e Técnicas

Exceções

Prof. Aléssio M. Jr

Conceituação

- Nem todas condições geradoras de erro podem ser detectadas em tempo de compilação
- Software seguro e confiável deve implementar um comportamento aceitável na presença dessas condições anormais
- Termo exceção designa um evento ocorrido durante a execução de um programa que desvia o fluxo normal
- Uma exceção é uma condição provocada por uma situação excepcional a qual requer uma ação específica imediata

Causas de Exceções

Exceções	Erros	Hardware
		Software
	Fluxo	Múltiplos Resultados

- Erros Hardware
  - Falha no disco rígido
- Erros Software
  - Acesso a índice inexistente de vetor
- Múltiplos resultados
  - Leitura de registro ou fim de arquivo

Abordagens de LPs para Exceções

- Não oferecer recursos próprios
  - Tratamento através de mecanismos já existentes (testes, subprogramas e desvio incondicional)
  - C, PASCAL e MODULA-2
- Possuir mecanismo de tratamento de Exceções
  - Comandos específicos
  - Novo tipo de fluxo de execução
  - ADA, C++ e JAVA

Ausência de Mecanismo de Exceção

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

Aborto

- Reduz a confiança do usuário no sistema
- Dificulta a depuração dos erros
- Muitas exceções podem ser contornadas sem que seja necessário interromper a execução do programa

Teste e Tratamento Imediato

- Carrega muito o texto do programa com código de tratamento
  - Obscuresce a funcionalidade do algoritmo com testes de exceções
  - Subprogramas para tratamento reduzem esse problema
- Programador tem de lembrar, identificar e testar todas as possíveis condições causadoras de exceções
  - Normalmente não ocorre
- Algumas exceções não podem ser tratadas localmente

Teste e Tratamento Imediato

```
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;
}
```

Teste e Tratamento Imediato

```
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);
}
```

Retorno de Código de Erro

- Quem chama deve realizar teste e tratamento para cada código de retorno
  - Sobrecarga de código fica ainda maior
    - Testes no local da exceção e no código de chamada
    - Pode duplicar o tamanho de um programa
- Resolve o problema de tratamento não local da exceção
- Experiência mostra que o programador não testa todos os códigos de retorno possíveis
  - Não é obrigatório fazê-lo

Retorno de Código de Erro

- Resultado da função
  - Nem sempre possível por incompatibilidade com o resultado normal da função
- Variável global
  - Usuário da função pode não ter ciência de que essa variável existe
    - Isso não fica explícito na chamada
    - errno de C
  - Outra exceção pode ocorrer antes do tratamento da anterior
    - Problema maior em programas concorrentes

Retorno de Código de Erro

- Parâmetro de saída
  - Melhor do que o retorno em variável global ou no resultado da função
  - Exige inclusão de um novo parâmetro nas chamadas dos subprogramas
  - Requer a propagação desse parâmetro até o ponto de tratamento da exceção
  - Diminui a redigibilidade do código

Retorno de Código de Erro

```
int f(int x) {
    if (condicao1(x)) return 1;
    if (condicao2(x)) return 2;
    if (condicao3(x)) return 3;
    executaFuncionalidade(x);
    return 0;
}
void g() {
    int resp;
    resp = f(7);
    if (resp == 1) trata1();
    if (resp == 2) trata2();
    if (resp == 3) trata3();
}
```

Outras Opções em C

- Utilização do sistema de manipulação de sinais de sua biblioteca padrão
  - Sinais gerados
    - Por função raise
    - Em resposta a comportamento excepcional
  - Tratamento na função signal
- Uso das funções da biblioteca padrão setjmp e longjmp
  - salvam e recuperam estado do programa
  - longjmp é um goto não local
    - passa o controle do programa para o ponto onde o último setjmp foi executado

Outras Opções em C

- Exigem tratamento imediato da exceção
- Solução signal concentra o tratamento de todas as exceções em uma única função
- Solução setjmp e longjmp permite localizar o tratamento em qualquer ponto do programa
  - Restringe o tratamento ao último setjmp
- Soluções complexas e com baixa legibilidade
- Fica a critério do programador C decidir qual a abordagem de tratamento será utilizada

Mecanismos de Tratamento de Exceções

- Buscam garantir e estimular o tratamento das condições excepcionais sem que haja uma grande sobrecarga do texto do programa
- Quando uma exceção ocorre ela necessita ser tratada
- Tratador de Exceção
  - Bloco ou unidade de código que manipula a exceção
- Sinalização ou disparo da exceção
  - Ação de indicar a ocorrência da exceção e transferir o controle para o tratador

Tipos de Exceções

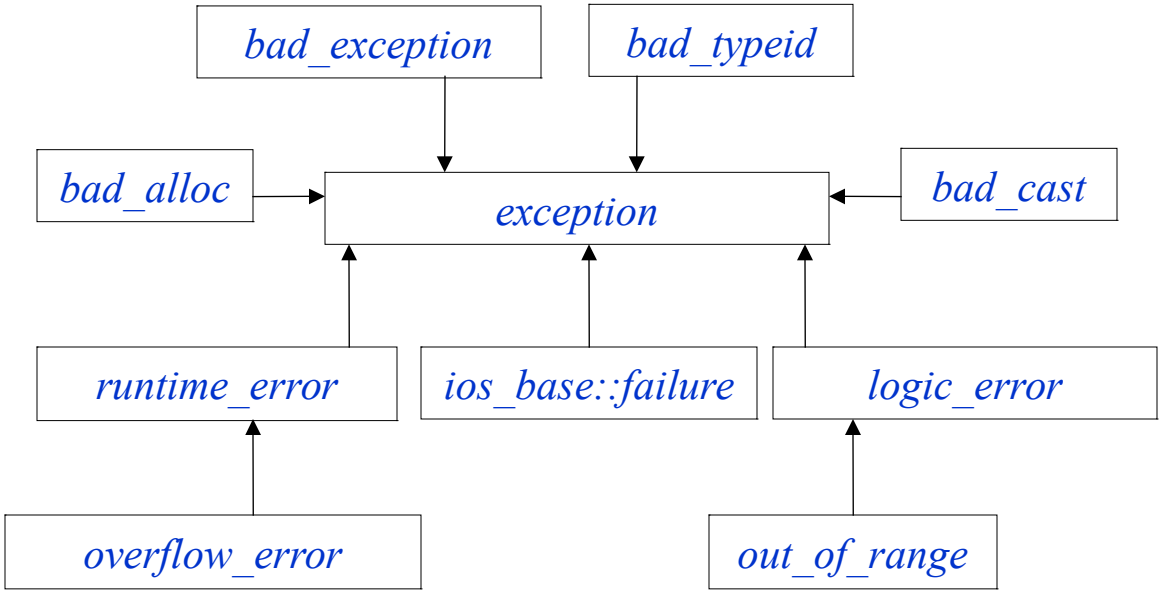
- Diferem na
  - Definição
    - Pela LP – overflow em C++
    - Pelo Programador – estoque baixo
  - Sinalização
    - Pela LP – acesso indevido a vetor em JAVA
    - Pelo Programador – estoque baixo
  - Tratamento
    - Obrigatório – Programador tem de tratar
    - Opcional – Programador pode não tratar

Exceções em LPs OO

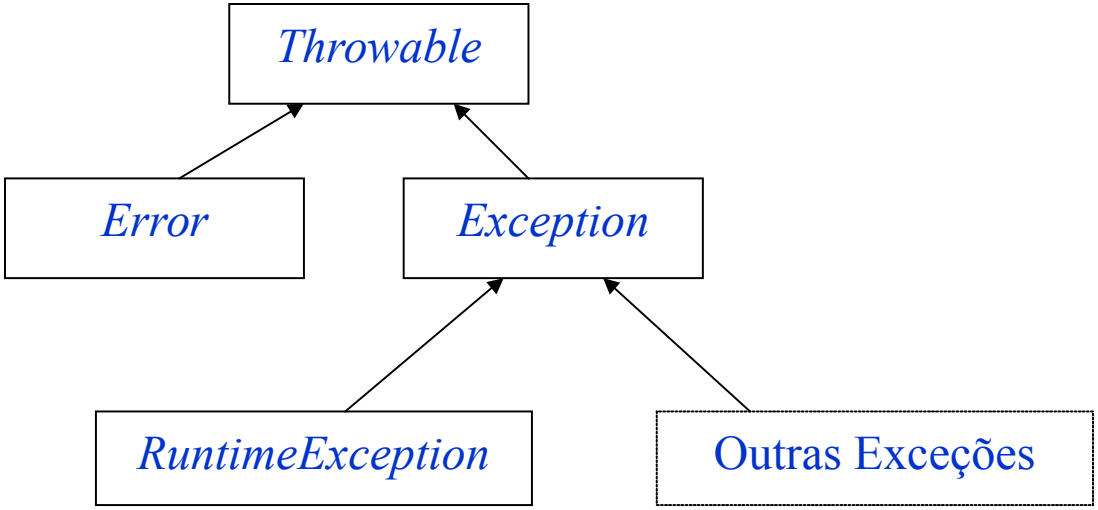
- Objetos
  - Podem ser lançados para outras partes do programa seguindo um fluxo de controle distinto do usual
  - Classes podem ser especiais ou não
- Devem ser organizadas dentro de uma hierarquia de classes
  - Exemplo em C++

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

Exceções Padrão de C++



Exceções em JAVA



- Throwable
  - Classe especial
  - Objetos disparáveis pelo mecanismo de exceção
- Error
  - Classes de erros graves (não recuperáveis)
  - Programador normalmente não manipula

Exceções em JAVA

- Exception
  - Superclasse de todas as exceções manipuláveis pelo programador
- RuntimeException
  - Subclasse especial de exceções
  - Geralmente são disparadas pela LP
  - Programador não é obrigado a tratá-las
  - Embora normalmente indiquem problemas sérios que devem implicar na terminação do programa, permitem que o programador tenha a opção de tratá-las
  - Aumenta a redigibilidade e reduz a confiabilidade pois tratamento não é requerido

Definição de Exceções em JAVA

- Necessário criar subclasse de Exception ou de alguma de suas subclasses

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

Sinalização de Exceções

- Pelo próprio mecanismo de exceções
- Explicitamente pelo programador
  - Uso de try e throw
  - Em C++

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

  - Em JAVA

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

Tratadores de Exceções

- Trecho de código do programa responsável por tomar atitudes em resposta à ocorrência de uma exceção
- Não são chamados explicitamente
  - Não precisam possuir nome
- Uso de `catch` associado a `try`

Tratadores de Exceções

```
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 ");
}
```

Tratadores de Exceções

- Melhora a redigibilidade
  - Não necessita incluir testes de exceções após cada chamada
- Melhora a legibilidade
  - Separa código de tratamento do código de funcionalidade

Tratadores de Exceções

- Distribuição Inapropriada

```
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
}
```

Captura de Qualquer Exceção

- Em JAVA

```
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");
}
```

Captura de Qualquer Exceção

- Em C++

```
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
}
```

Propagação de Exceções

```
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();
```

Propagação de Exceções

```
            }
          } 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");
   }
}
```

Relançamento de Exceções

```
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 ");
   }
}
```

Especificação de Exceções

- Subprograma pode lançar exceção sem tratar
- Subprograma necessita explicitar exceções que pode disparar
- Em C++

```
void f() throw (A,B,C);        // dispara A, B e C
void g() throw();      // não dispara
void h();      // pode disparar qq exceção
```

  - Não verifica se o compromisso assumido na especificação está sendo cumprido
  - Não obriga a função chamadora a tratar todas as exceções possíveis de serem geradas pela função chamada

Especificação de Exceções

```
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";
    }
}
```

Especificação de Exceções

```
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";
    }
}
```

Especificação de Exceções

- Em JAVA
  - Especificação das exceções não tratadas é obrigatória
  - Exceções RuntimeException não precisam

```
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();
```

Especificação de Exceções

```
            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");
    }
}
```

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");
   }
}
```

Propagação de Exceções entre Métodos

```
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");
```

Propagação de Exceções entre Métodos

```
    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");
    }
}
```

Modos de Continuação

- Terminação
  - Assume o erro como crítico
  - Não retorna ao ponto no qual a exceção foi gerada
  - O controle retorna para um ponto mais externo do programa
- Retomada
  - Assume o erro como corrigível
  - A execução pode retornar para o bloco no qual ocorreu a exceção
  - Experiência indica baixa efetividade dessa opção
- Maioria das LPs adota o modelo de terminação

Terminação

- Em C++

```
class ErroI { };
class ErroII { };
class ErroIII { };
void f () throw (ErroI) {
   throw ErroI();
}
main() {
   cout << "comeca aqui\n";
   try {
      cout << "passa por aqui\n";
```

Terminação

```
    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";
}
```

Retomada

- 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;
```

Retomada

```
            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;
            }
         }
      }
   }
```

A Cláusula finally

```
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");
         }
```

A Cláusula finally

```
        } catch(Exception e) {
            System.out.println("Excecao capturada");
        } finally {
            System.out.println("finally do bloco 1");
        }
    }
}
```

A Cláusula finally

```
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() {}
```

A Cláusula finally

```
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!!!");
  }
 }
}
```

Perda de Exceção em JAVA

```
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 ();
   }
```

Perda de Exceção em JAVA

```
public static void main(String[] args) throws Exception {
    Perda p = new Perda();
    try {
       p.infarto();
    } finally {
       p.resfriado();
    }
  }
}
```

Exceções e Polimorfismo

- Aumenta a complexidade dos programas
- Necessário definir regras
  - Em JAVA
    - Os construtores podem adicionar novas exceções a serem propagadas às declaradas no construtor da superclasse
    - Os construtores devem necessariamente propagar as exceções declaradas no construtor da superclasse
    - Métodos declarados na superclasse não podem ter novas exceções propagadas
    - Não é obrigatório propagar as exceções dos métodos da superclasse
    - Os métodos sobrescritos podem disparar exceções que sejam subclasses das exceções propagadas na superclasse

Exceções e Polimorfismo

- Em JAVA

```
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() {}
}
```

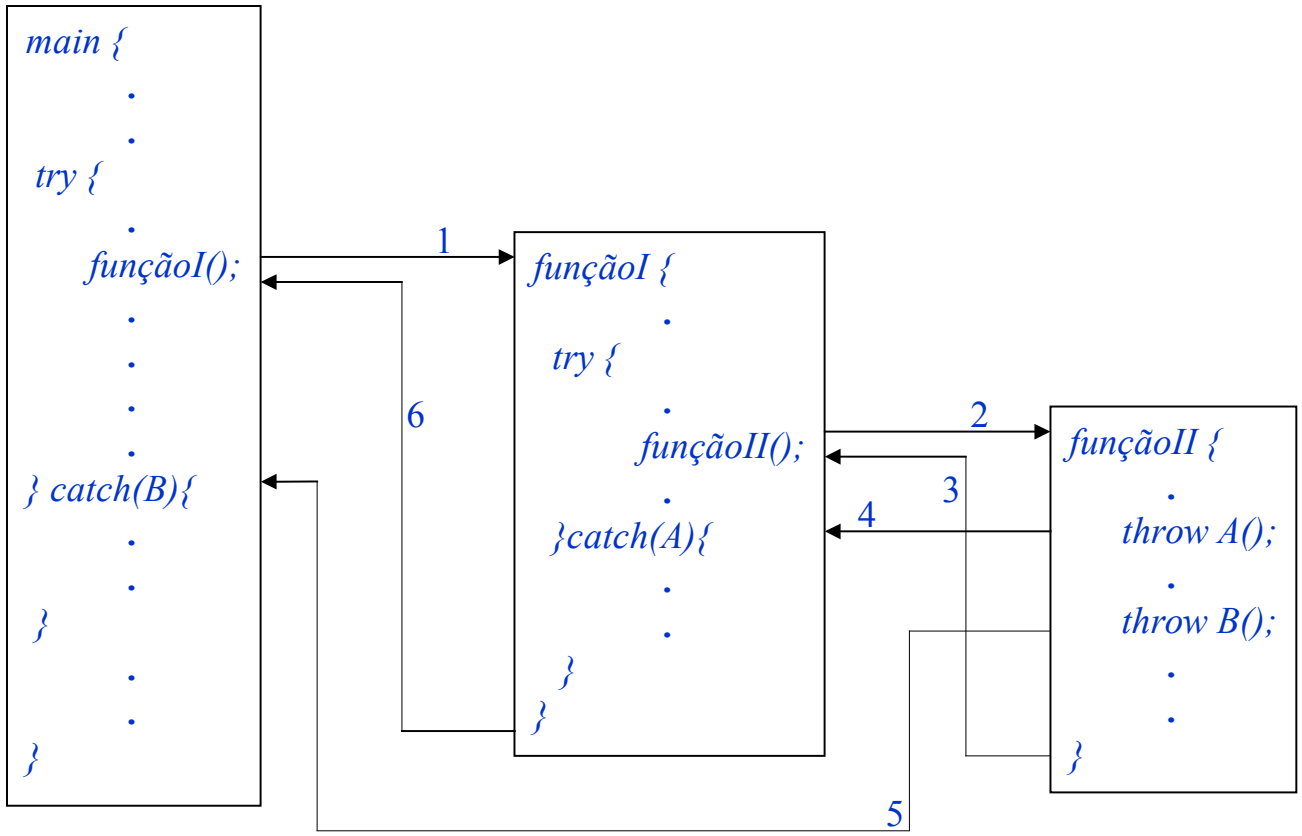
Exceções e Polimorfismo

```
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 {}
```

Exceções e Polimorfismo

```
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) {}
 }
}
```

Fluxo de Controle com Exceções



Vantagens do Mecanismo de Exceções

- Melhoram a legibilidade dos programas
  - Separam o código com a funcionalidade principal do programa do código responsável pelo tratamento de exceções
- Aumentam a confiabilidade e robustez dos programas
  - Normalmente requerem o tratamento obrigatório das exceções ocorridas
  - Promovem a idéia de recuperação dos programas mesmo na presença de situações anômalas
- Incentivam a reutilização e a modularidade do código responsável pelo tratamento

Desvantagens do Mecanismo de Exceções

- Trazem maior complexidade para o aprendizado da linguagem
- Podem reduzir a eficiência computacional dos programas nessa linguagem
- Fragilidades em 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
  - Não existe obrigação de explicitar a exceção relançada para o nível superior