PROGRAMAÇÃO ORIENTADA PELOS OBJECTOS

Aulas 20-22

Excepções e asserções

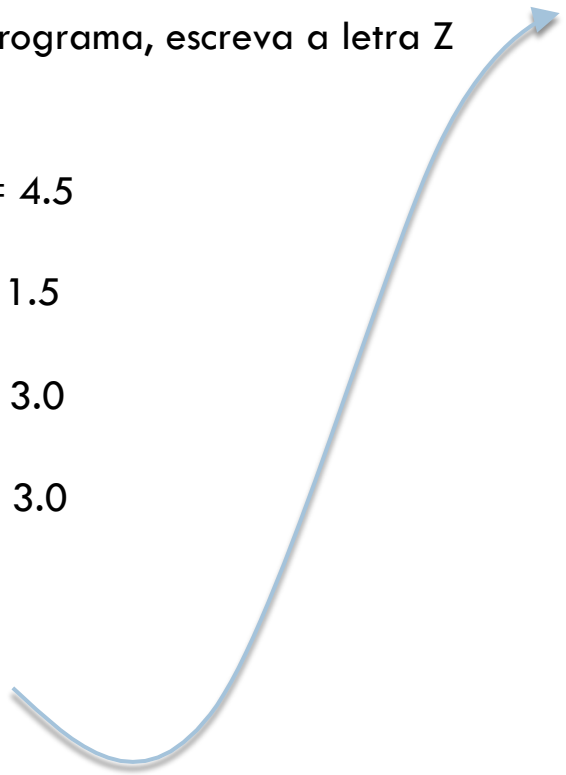
2 Uma calculadora simples

Calculadora simples

- Pretendemos implementar uma calculadora simples mas robusta tão quanto possível, de modo a que eventuais erros na introdução de dados por parte do utilizador não impliquem a interrupção abrupta do programa
- Operações básicas
  - Soma
  - Subtracção
  - Multiplicação
  - Divisão

Exemplos de traço do programa


```
   Calculadora de operacoes basicas
Formato de entrada em cada linha: operador numero
Por exemplo: + 3
Para terminar o programa, escreva a letra Z
Resultado = 0.0
> + 4.5
Resultado + 4.5 = 4.5
> - 3
Resultado - 3.0 = 1.5
> * 2
Resultado * 2.0 = 3.0
> / 1
Resultado / 1.0 = 3.0
```



```
> q 2
q nao e um operador conhecido.
Tente mais uma vez ...

   Calculadora de operacoes basicas
Formato de entrada em cada linha: operador numero
Por exemplo: + 3
Para terminar o programa, escreva a letra Z
Resultado = 0.0
> + 4
Resultado + 4.0 = 4.0
> q 2
q nao e um operador conhecido.
Ja chega! Tente noutra altura.
Fim do programa.
```

Exemplos de traço do programa

```
   Calculadora de operacoes basicas
Formato de entrada em cada linha: operador numero
Por exemplo: + 3
Para terminar o programa, escreva a letra Z
Resultado = 0.0
> * 2
Resultado * 2.0 = 0.0
> + 45
Resultado + 45.0 = 45.0
> / 0.00001
Divisao por zero.
Fim do programa.
```

```
   Calculadora de operacoes basicas
Formato de entrada em cada linha: operador numero
Por exemplo: + 3
Para terminar o programa, escreva a letra Z
Resultado = 0.0
> + 45.2
Resultado + 45.2 = 45.2
> Z
O resultado final e 45.2
Fim do programa.
```

6

Erros em programação

FCT FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

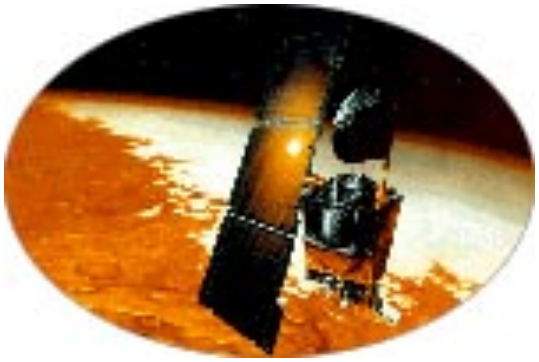
Quando as coisas correm mal

- Erros decorrentes de atividades humanas, como é o caso de programação, existem e existirão sempre
- Os erros de sintaxe de programação são detectados durante a compilação logo facilmente corrigidos
- Os erros lógicos ou de programação são detectados em tempo de execução, podendo levar a
  - Comportamento inesperado e/ou incorreto
  - Terminar abruptamente o programa, o que é extremamente incorreto
- Existem ainda erros associados ao ambiente de execução, sobre os quais um programa terá pouco controlo
  - Exemplo: corte da ligação de rede, erro de escrita em disco, etc.
- Erros lógicos e situações inesperadas devem ser detectadas na medida do possível e geridas convenientemente

Exemplo de falha grave de software

"On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after its lift-off from Kourou, French Guiana. It turned out that the cause of the failure was a software error in the inertial reference system. Specifically a 64 bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16 bit signed integer. The number was larger than 32,767, the largest integer storeable in a 16 bit signed integer, and thus the conversion failed"

Exemplo de falha grave de software



"The 125 million dollar Mars Climate Orbiter is assumed lost by officials at NASA. The failure responsible for loss of the orbiter is attributed to a failure of NASA's system engineer process. The process did not specify the system of measurement to be used on the project. As a result, one of the development teams used Imperial measurement while the other used the metric system of measurement. When parameters from one module were passed to another during orbit navigation correct, no conversion was performed, resulting in the loss of the craft"

Erros e qualidade de programação

- Em vez de programar olhando apenas para o lado positivo, temos de programar também para a eventualidade da existência de erros
- Seguindo uma perspectiva antiga, podemos sempre implementar métodos de modo a que estes devolvam um valor/código de erro como resultado da sua execução, numa óptica de erro
  - Se não analisarmos conveniente tal código de retorno, é possível que não sejam detectadas eventuais falhas do código
  - Esta metodologia de programação é confusa e conduz a código de qualidade inferior (*spaghetti code*)
- É preferível analisar e processar os erros através do mecanismo de excepções

Vantagens do mecanismo de excepções

- Separação do código afecto à gestão e processamento de erros do restante código
  - Programador escreve o fluxo normal de código e remete o tratamento de casos excepcionais para outra localização
  - O trabalho de detectar, reportar e controlar erros fica organizado de uma forma mais efetiva
- Propagação de erros na pilha de chamadas de métodos que controla a execução do programa
  - Permite que os erros sejam propagados até ao método que está "interessado" no seu processamento, e que foi concebido para o efeito
- Agrupamento e diferenciação de diferentes tipos de erros

O que é uma excepção

Excepção

- Uma excepção é um evento pouco frequente, normalmente associado a um erro ou situação anormal, que é detectado por hardware ou software e necessita de algum tipo de processamento especial
  - Exemplos: divisão por zero, fim de ficheiro não esperado, dados inválidos, abertura de um ficheiro que não existe, falta de memória, acesso a um vector para além dos seus limites, etc.
- Uma excepção altera o fluxo de controlo normal do programa
- Formalmente, uma excepção em Java é um objecto com variáveis e métodos associados, e que nos permite avaliar e processar a situação em causa
- Operações associadas a excepções
  - Criação da excepção
  - Lançamento da excepção, com o programa a notificar que algo estranho ocorreu
  - Captura e processamento da excepção, com o programa a direcionar o controlo do programa para código de análise e processamento de erros

Gestão de uma excepção

Lançamento e processamento de uma excepção

- Lançamento de uma excepção em Java
  - Após uma situação de excepção num método, este cria um objecto de excepção e passa-o para o sistema de *runtime*
  - Exemplo:

```
throw new ArithmeticException("...");
```

  - A menos que o método contenha código que capture a excepção, o método que gerou a excepção termina
- Após o lançamento da excepção, que não pode ser menosprezada, a execução do programa continua no gestor de excepções
- Processamento da excepção lançada
  - Execução de código especifico para o efeito
  - O objecto de excepção gerado contém informação sobre o erro em causa, incluindo o seu tipo e estado do programa quando este ocorreu

Exemplo de lançamento de uma excepção


```
public class OtherAccount {

   public void withdraw(double amount) {
      if (amount > balance) {
         IllegalArgumentException exception
           = new IllegalArgumentException(”Not enough money");
         throw exception;
      }
      balance -= amount;
   }
   ...
}
```

Controlo e processamento de excepções

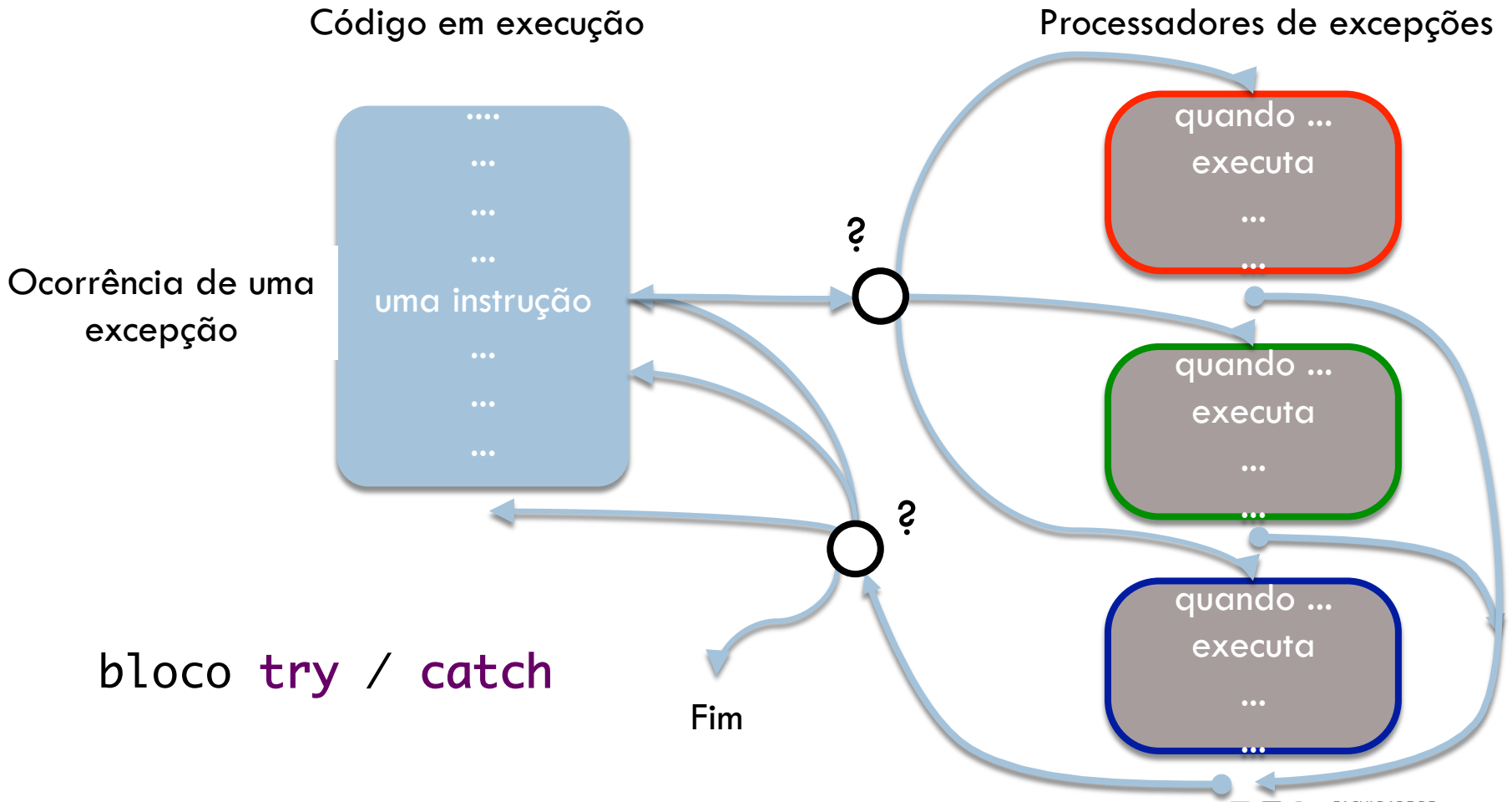
- Em Java, as excepções são geridas através de um bloco try, da cláusula de processamento catch e da cláusula finally
  - O bloco try termina normalmente, ou então com a captura (ou não) de um excepção
  - Se for lançada uma excepção e esta não for capturada por cláusulas catch, será o próprio sistema a capturar a excepção e a terminar o programa de seguida (caso de *default handling exception*)
  - As várias cláusulas catch são testadas relativamente à excepção gerada, tendo como base uma hierarquia de classes de excepções
  - A cláusula finally, se existir, será sempre executada

Controlo e processamento de excepções


```
try {
   <code to be monitored that may raise exceptions,
    i.e. with a throw statement or a method
    invocation that might throw an exception>
} catch ( <ExceptionType1> <Obj1> ) {
    <handler if ExceptionType1 occurs in the try bloc>
} catch ( <ExceptionType2> <Obj2> ) {
    <Handler if ExceptionType2 occurs in the try bloc>
} . . .
} finally {
    <code to be executed before the try bloc ends>
}
```

Importa acima de tudo saber se é lançada e não tanto onde é lançada

Controlo de uma excepção



Recapitulando

- Executam-se as instruções que estão no bloco try
- Se não ocorrer nenhuma excepção, as cláusulas que constam na lista de catch são omitidas
- Mas se ocorrer uma excepção que coincida com um dos tipos indicados em catch, então a execução vai para a respectiva cláusula catch
- Se ocorrer uma excepção de outro tipo, essa excepção é lançada até que seja capturada, eventualmente por outro bloco try
  - No limite, será detectada pelo gestor de excepções próprio do sistema

Exemplo de um bloco try/catch


```
try {
    String filename = ... ;
    FileReader reader = new FileReader(filename);
    Scanner in = new Scanner(reader);
    String input = in.next();
    int value = Integer.parseInt(input);
    ...
}
```

```
catch (NumberFormatException exception) {
    System.out.println("Input was not a number");
}
```

```
catch (IOException exception) {
    exception.printStackTrace();
}
```

Gestão de excepções com cláusula finally

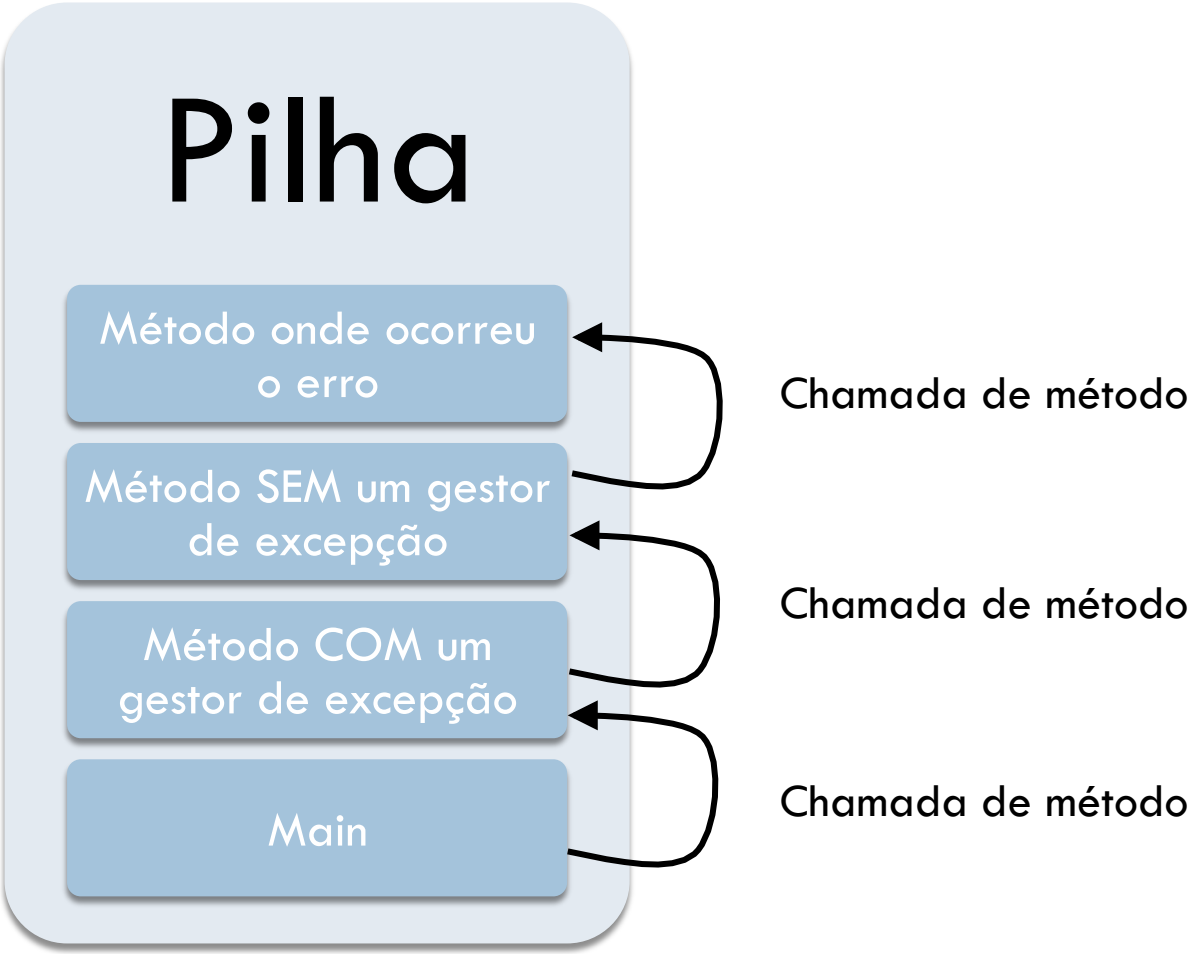
- Quando uma excepção termina um método, existe o risco de também ser omitida a execução de operações consideradas importantes
- Por exemplo, no código seguinte, a instrução `reader.close()` deve ser executada mesmo que seja lançada uma excepção. Nestas situações, deve-se utilizar uma cláusula finally

```
...
reader = new FileReader("ficheiroTeste.txt");
Scanner in = new Scanner(reader);
readData(in);
reader.close(); // pode não chegar aqui !!!
```

Execução da cláusula finally

- Uma cláusula finally no bloco try é sempre executada, de acordo com um dos seguintes cenários:
  - depois da última instrução do bloco try
  - depois da última instrução da cláusula catch que capturou a excepção
  - quando no bloco try é lançada uma excepção e não é capturada por nenhuma cláusula catch

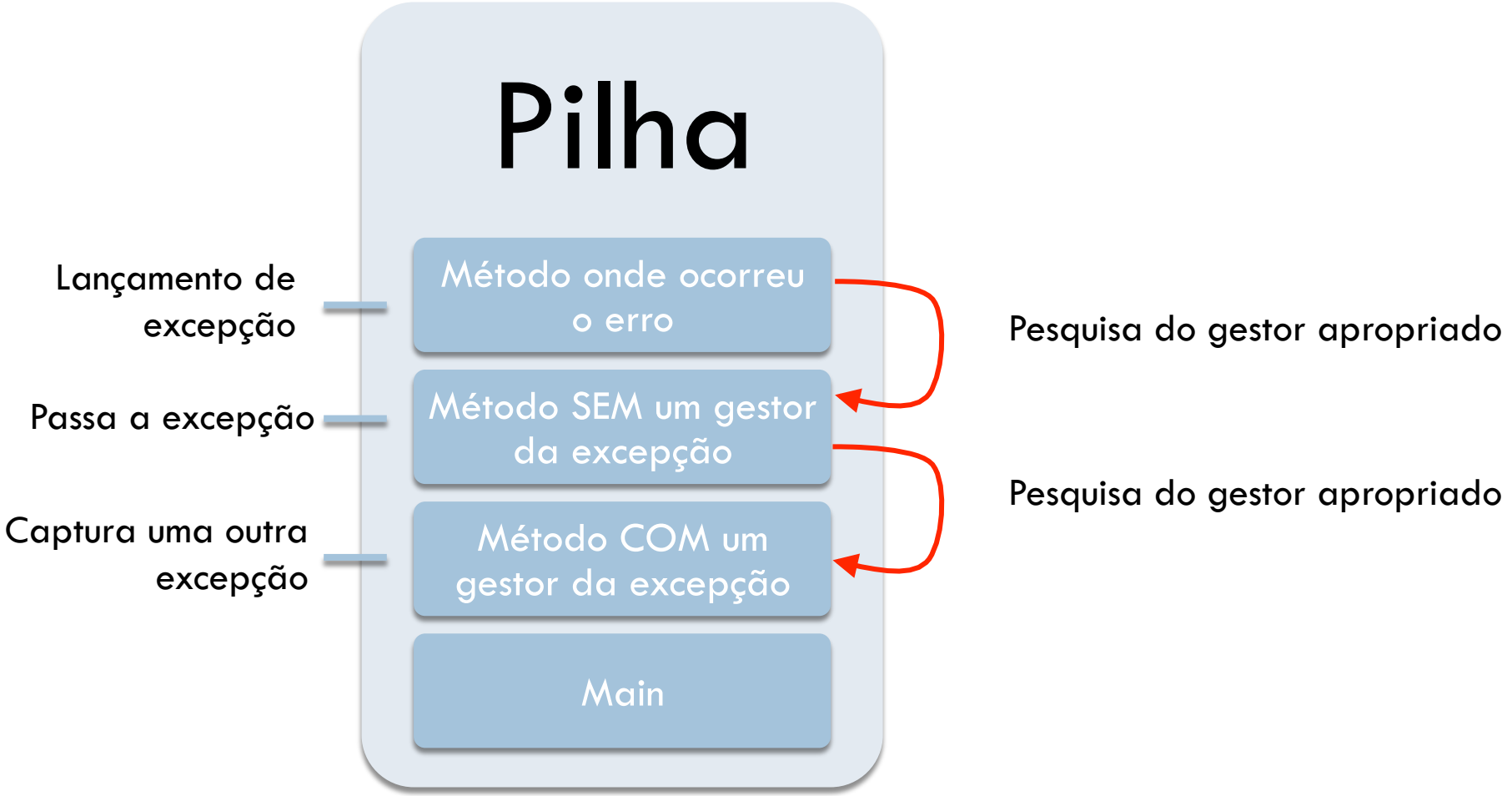
Exemplo de uma pilha de chamadas



Propagação de erros na pilha de chamada

- Quando é lançada uma excepção, o sistema em *runtime* do Java faz um pesquisa em sentido inverso na pilha de chamada com o objectivo de encontrar métodos que estejam associados à gestão dessa excepção

Procura do gestor da excepção



Excepção não detectada pelo código

- Existe uma mensagem de erro quando a excepção não é capturada pelo código
- O gestor de excepções por defeito, em *runtime*
  - Escreve a descrição da excepção
  - Escreve o traço da pilha, indicando a hierarquia de métodos onde ocorreu a excepção
  - Termina o programa

```
Formato de entrada em cada linha: operador numero
Por exemplo: + 3
Para terminar o programa, escreva a letra Z
Resultado = 0.0
> + palavra
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:840)
        at java.util.Scanner.next(Scanner.java:1461)
        at java.util.Scanner.nextDouble(Scanner.java:2387)
        at poo.CalculatorMain.doCalculation(CalculatorMain.java:57)
        at poo.CalculatorMain.main(CalculatorMain.java:15)
```

Implementação de excepções

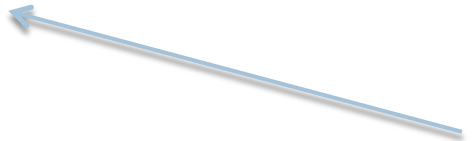
- O Java permite criar excepções. Para isso devemos:
  - Criar uma classe que estenda a classe Exception
  - Redesenhar a classe criada, adicionando variáveis de classe e construtores, e tendo em consideração os objectivos a alcançar
- Na nova classe podemos
  - Invocar um dos métodos standards `getMessage()` e `printStackTrace()`, este último imprime a sequência de chamada de métodos que levaram à situação de excepção
  - Escrever uma mensagem com informação própria da classe, eventualmente recorrendo a variáveis de classe contendo informação pertinente
- Após a sua implementação, a invocação desta classe pelo código segue os mesmos princípios de outras classes de excepção
- Note-se que pode ser conveniente tomar alguma acção corretiva em função da excepção gerada

Alguns conselhos

- Devemos usar o mecanismo das excepções com ponderação
- A ordem das cláusulas catch é importante
  - Como é a primeira cláusula que coincide com a excepção que é executada, devemos colocar a mais específica em primeiro lugar

```
catch (DivideByZeroException exception) {
      . . .
}

catch (Exception exception) {
      ...
}
```



mais específica

A ordem é mesmo importante


```
try {...}
catch (IOException e) {...}
catch (Exception e) {...}
```

Como fazer bem

- Tudo correcto, primeiro trata-se IOException e só depois outras excepções

```
try {...}
catch (Exception e) {...}
catch (IOException e) {...}
```

Como não fazer

- Exception é super-classe de IOException
- O Código de IOException nunca seria executado!

Algumas regras a seguir

- Por norma, devemos colocar em métodos distintos os códigos de lançamento e de captura da excepção

```
public void methodB() {
      ...
      try {
          ... methodA() ...
      }
      catch ( MyException excep ) {
         < Handle_Exception >
      }
      ...
}
```

```
public void methodA() throws MyException {
      . . .
      throw new MyException("My own message");
      . . .
}
```

Tipos de excepções

Tipos de excepções

- Excepções verificadas
  - O compilador verifica se não são ignoradas pelo programa
    - Se forem ignoradas, origina um erro de compilação
  - Associadas a circunstâncias externas que o programador não pode prever, como por exemplo um ficheiro não estar localizado no lugar devido
  - A maior parte destas excepções estão relacionadas com operações de entrada/saída
- Excepções não verificadas
  - Não são sujeitas a verificação de gestão de excepções por parte do compilador
  - São uma extensão da classe RuntimeException ou Error. Em princípio, resultam de erros de programação

Excepções verificadas

- As classes podem não ter capacidade de responder a todas as situações imprevistas
  - Exemplos
    - `Scanner.nextInt()` lança a excepção não verificada InputMismatchException quando o utilizador, incorretamente, fornece um valor não inteiro
    - `Integer.parseInt()` lança NumberFormatException quando o utilizador fornece um valor não inteiro
- Devemos considerar a utilização de excepções verificadas, sobretudo quando se está a lidar com ficheiros
- Exemplo de leitura de um ficheiro com a classe Scanner
  - … mas o construtor de FileReader pode lançar uma excepção FileNotFoundException se o ficheiro não existir !

```
...
String fileName = "ficheiroTeste.txt"
FileReader reader = new FileReader(fileName);
Scanner in = new Scanner(reader);
```

Excepções verificadas

- Duas soluções possíveis:
  - Gerir a situação decorrente da excepção
  - Indicar ao compilador que o método deve terminar quando ocorrer a excepção. Para isso, utiliza-se um especificador de lançamento para que o método possa lançar uma excepção verificada

```
public void read(String filename) throws IOException,
                                         ClassNotFoundException {
    ...
}
```

```
public void read(String filename) throws FileNotFoundException {
   FileReader reader = new FileReader(filename);
   Scanner in = new Scanner(reader);
   ...
}
```

Hierarquia de classes de excepções

Throwable (java.lang)

- Error
  - LinkageError, ...
  - VirtualMachineError, ...
- Exception
  - ClassNotFoundException
  - CloneNotSupportedException
  - IllegalAcessException
  - IOException
    - EOFException
    - FileNotFoundException
    - ...
  - RuntimeException
    - AritmeticException
    - IllegalArgumentException
    - IndexOutOfBoundsException
    - NullPointerException
    - ...
  - ...

checked

unchecked

- *Error*
  - Para gerir erros ocorridos no ambiente de execução, por norma fora do controlo dos utilizadores de programa, como é o caso de erros de memória ou falha do disco rígido
- *Exception*
  - Em princípio, para situações que os utilizadores podem gerir, como é o caso de divisões por zero ou acesso fora dos limites de vectores

Agrupamento e diferenciação de erros

- É fácil organizar o tratamento de erros segundo a hierarquia de classes de excepções uma vez que as excepções são objetos gerados
- Exemplo: java.io.IOException e descendentes
  - IOException é a classe mais genérica, associada a todo o tipo de erros que possam ocorrer relacionados com operações I/O
  - As classes descendentes representam erros mais específicos, como é o caso de FileNotFoundException
- Um método pode detectar uma excepção baseada no seu grupo ou tipo geral se especificarmos na cláusula catch alguma das superclasses respectivas
  - Exemplo: com IOException na cláusula catch, o método consegue detectar todas as excepções de IO, incluindo FileNotFoundException, EOFException, e outras

Valtando à calculadora

Diagrama de classes

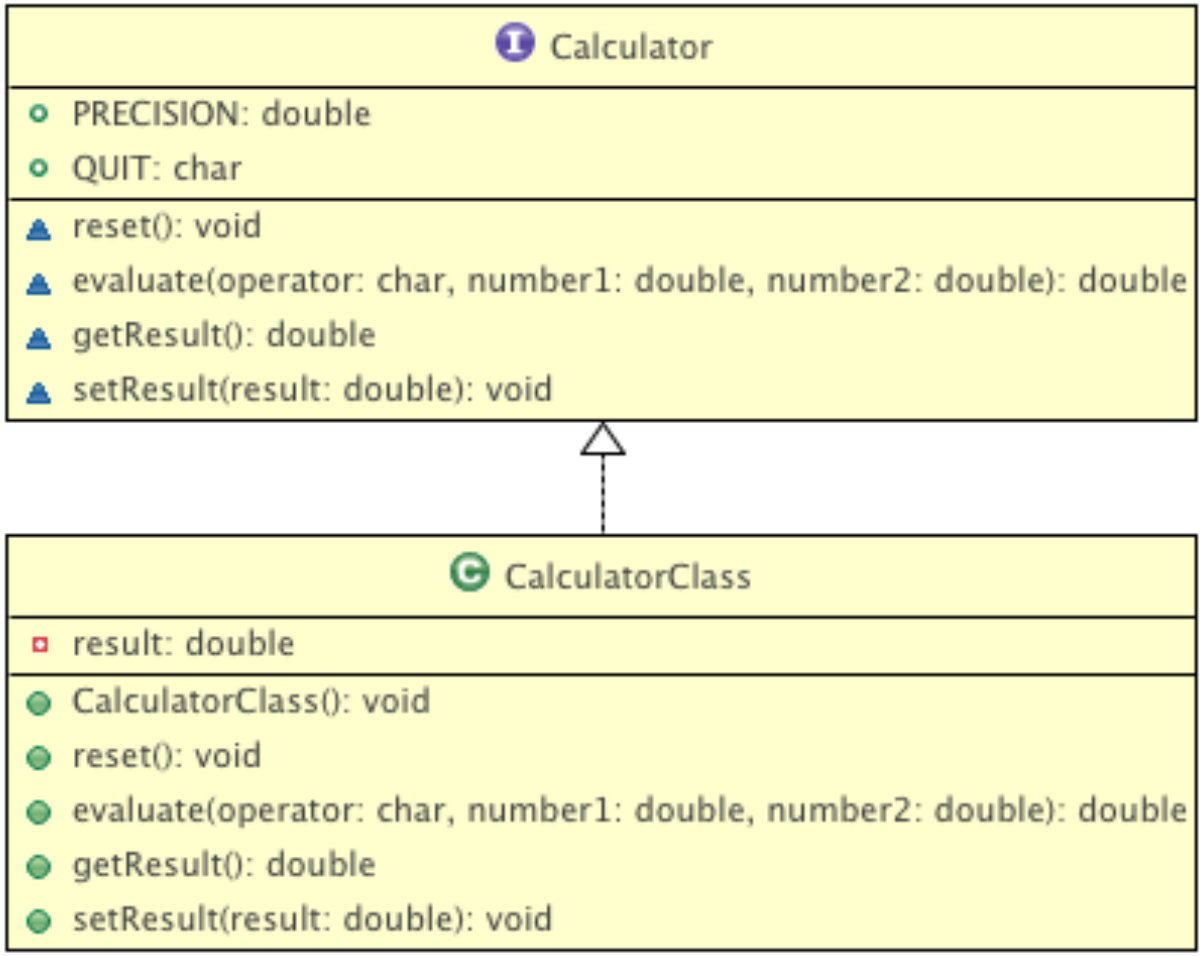
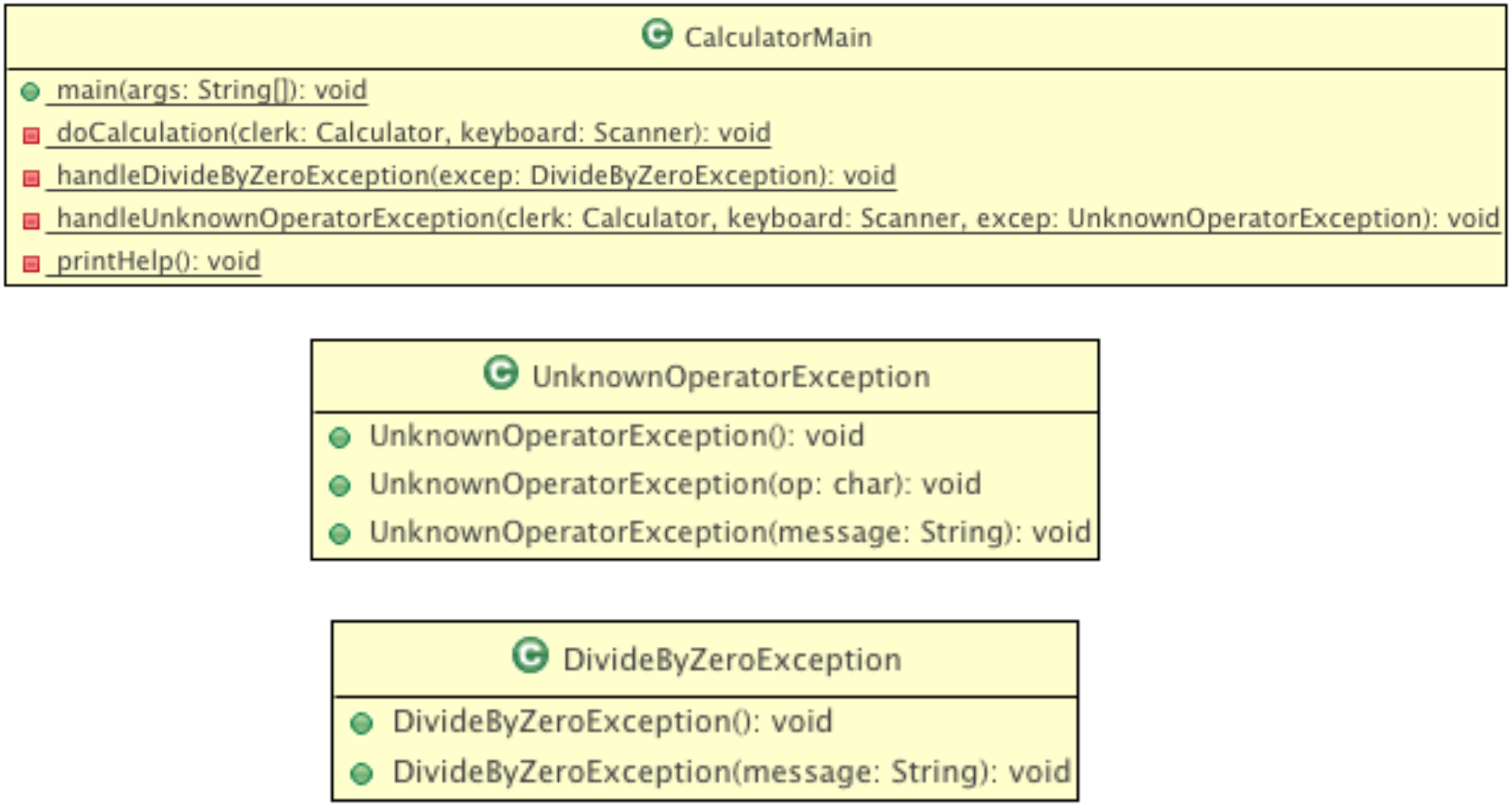


Diagrama de classes



A interface Calculator


```
package poo;

/**
 * @author adrianolopes
 * Simple line-oriented calculator program
 */
public interface Calculator {

    // Numbers this close are treated as if equal to zero
    final double PRECISION = 0.000001;
    final char QUIT = 'Z';

    void reset();
    double evaluate(char operator, double number1,
                            double number2) throws
         DivideByZeroException, UnknownOperatorException;
    double getResult();
    void setResult(double result);
}
```

A classe CalculatorClass


```
package poo;

public class CalculatorClass implements Calculator {

  private double result;

  public CalculatorClass() { result = 0; }

  public void reset() { result = 0; }

  public double evaluate(char operator, double number1,
                         double number2) throws
         DivideByZeroException, UnknownOperatorException {
         . . .
  }

  public double getResult() { return result; }

  public void setResult(double result) {
       this.result = result;
  }

}
```

A classe CalculatorClass


```
public double evaluate(char operator,
          double number1, double number2) throws
        DivideByZeroException, UnknownOperatorException {

  double answer;
  switch (operator) {
    case '+':
      answer = number1 + number2; break;
    case '-':
      answer = number1 - number2; break;
    case '*':
      answer = number1 * number2; break;
    case '/':
      if ( (-Calculator.PRECISION < number2) &&
                (number2 < Calculator.PRECISION) )
            throw new DivideByZeroException();
      answer = number1/number2;
      break;
    default:
      throw new UnknownOperatorException(operator);
  }
  return answer;
}
```

A classe DivideByZeroException


```
package poo;

public class DivideByZeroException extends Exception {

    public DivideByZeroException( ) {
        super("Divisao por zero.");
    }

    public DivideByZeroException(String message) {
        super(message);
    }

}
```

A classe UnknownOperatorException


```
package poo;

public class UnknownOperatorException extends Exception {

    public UnknownOperatorException( ) {
        super("UnknownOperatorException");
    }

    public UnknownOperatorException(char op) {
        super(op + " nao e um operador conhecido.");
    }

    public UnknownOperatorException(String message) {
        super(message);
    }

}
```

A classe CalculatorMain


```
public static void main(String[] args) {

    Calculator clerk = new CalculatorClass( );
    Scanner keyboard = new Scanner(System.in);

    try {
      printHelp();
      doCalculation(clerk, keyboard );
    }
    catch (UnknownOperatorException excep) {
      handleUnknownOperatorException(clerk, keyboard, excep);
    }
    catch (DivideByZeroException excep) {
      handleDivideByZeroException(excep);
    }
    System.out.println("O resultado final e " + clerk.getResult());
    System.out.println("Fim do programa.");
  }
```

A classe CalculatorMain


```
private static void doCalculation(Calculator clerk, Scanner keyboard)
         throws DivideByZeroException, UnknownOperatorException {
  char nextOp;
  double nextNumber;

  boolean done = false;
  clerk.setResult(0);
  double result = clerk.getResult();
  System.out.println("Resultado = " + result);

  while (! done) {
    System.out.print("> ");
    nextOp = (keyboard.next( )).charAt(0);
    if ((nextOp == Calculator.QUIT)) {
      done = true;
    }
    else {
      nextNumber = keyboard.nextDouble( );
      // previous statement may launch an exception !!!
      // leave it as an exercise for students
      result = clerk.evaluate(nextOp, result, nextNumber);
      clerk.setResult(result);
      System.out.println("Resultado " + nextOp + " "
           + nextNumber + " = " + result);
    }
  }
}
```

A classe CalculatorMain


```
private static void handleDivideByZeroException(
            DivideByZeroException excep) {
  System.out.println("Divisao por zero.");
  System.out.println("Fim do programa.");
  System.exit(0);
}

private static void handleUnknownOperatorException(Calculator clerk,
              Scanner keyboard, UnknownOperatorException excep) {
  System.out.println(excep.getMessage());
  System.out.println("Tente mais uma vez ... ");
  try {
    printHelp();
    doCalculation(clerk, keyboard);
  }
  catch(UnknownOperatorException excep2) {
    System.out.println(excep2.getMessage());
    System.out.println("Ja chega! Tente noutra altura.");
    System.out.println("Fim do programa.");
    System.exit(0);
  }
  catch(DivideByZeroException excep3) {
    handleDivideByZeroException(excep3);
  }
}
```

Desafio de programação

- A execução do programa das contas bancárias pressupõe a introdução dos dados de modo consistente por parte do utilizador
  - Quanto ao número
  - Quanto ao tipo de dados
  - Quanto à validação semântica
- Com base no código disponibilizado e no respectivo diagrama de classes, construa um mecanismo de excepções apropriado para
  - Tornar o programa sintaticamente robusto
  - Sinalizar operações semanticamente incorretas, como é o caso de querer levantar dinheiro e o saldo da conta não o permitir
- Crie um ficheiro Javadoc para a aplicação
  - Note que o código fornecido não contém comentários

Asserções

O que é uma asserção

- Mecanismo que permite ao programador verificar se determinado pressuposto é alcançado
- Ao ler as instruções de asserção no código, sabemos que determinada condição no ponto da asserção deve ser sempre satisfeita
- Durante a execução do programa
  - O programa permite inferir se as asserções estabelecidas são verdadeiras ou não
  - Se um asserção não for verdadeira, é lançado uma excepção da classe AssertionError, que é derivada de Error

Ativação de asserções

- O utilizador do programa tem sempre a possibilidade de desativar o mecanismo de verificação de asserções
- Um programa com asserções pode funcionar deficientemente se o utilizador não souber que o programa contém asserções
- Compilação
  - Com mecanismo de asserção

```
javac -source MyProgram.java
```

  - Sem mecanismo de asserção

```
javac MyProgram.java
```

- Ativação de asserções

```
java -enableassertions MyProgram.java
```

(ou **-ea**)

Exemplo de uma asserção

- Uma pedaço de código de uma versão do programa sobre contas bancárias

```
. . .
else if (isCommand(command, IOCommands.SLEEPING_ACCOUNTS) ) {
      assert input.length > 0;
      // now we can look at input[0] safely
      GregorianCalendar afterDate =
              DateSetting(input[0].trim());
      printIterator(poorBank.accountsToPayInterest(afterDate));
}
. . .
```

Formatos de definição de asserções

- Primeiro formato:

```
assert Expression;
```

  - `Expression` é uma expressão booleana
  - Quando o sistema executa a asserção, avalia `Expression` e se for falsa, o sistema lança a excepção `AssertionError`, sem qualquer informação adicional

- Segundo formato:

```
assert Expression1 : Expression2;
```

  - `Expression1` é uma expressão booleana e `Expression2` é uma expressão que tem obrigatoriamente um valor (não pode ser void)
  - Este formato é utilizado quando se pretende dar informação adicional ao lançar a excepção `AssertionError`
    - A representação como String do valor calculado por `Expression2` é usada como mensagem de error detalhada da violação da asserção

Escolha do formato apropriado

- A mensagem de erro é interna e não para o utilizador
- O objectivo da mensagem de erro é ajudar a perceber exatamente o que provocou a violação da asserção, para que o problema possa ser resolvido
  - Por norma, a mensagem é lida juntamente com o *stack trace* da excepção lançada
- Apenas de deve usar o segundo formato quando temos informação adicional realmente útil, que complemente o *stack trace*
  - Por exemplo, se a asserção envolver uma relação entre dois valores x e y, uma expressão representando a mensagem de erro potencialmente interessante seria:

```
"x: " + x + ", y: " + y
```

Invariantes internos da classe

Sem asserções

```
if (i % 3 == 0) {
   ...
} else if (i % 3 == 1) {
   ...
} else {
   // sabemos que (i % 3 == 2)
   ...
}
```

Com asserções

```
if (i % 3 == 0) {
   ...
} else if (i % 3 == 1) {
   ...
} else {
   assert i % 3 == 2 : i;
   ...
}
```

Note-se que o código sem asserções pode falhar. Por exemplo, se `i` for negativo, o resto da divisão também será negativo. Sem a asserção, não detectaríamos o problema!

Invariantes internos da classe

Sem asserções

```
switch (suit) {
   case Suit.CLUBS: ...
     break;
   case Suit.DIAMONDS: ...
     break;
   case Suit.HEARTS: ...
     break;
   case Suit.SPADES: ...
}
```

Com asserções

```
switch (suit) {
   case Suit.CLUBS: ... break;
   case Suit.DIAMONDS: ... break;
   case Suit.HEARTS: ... break;
   default :
     assert false : suit;
    // ou, em vez de assert, throw
    // new AssertionError(suit);
}
```

A alternativa do lançamento de excepção é interessante, porque oferece protecção mesmo com as asserções desligadas. Além disso, se o método tiver de retornar um valor diferente em cada caso, a alternativa funciona bem, ao contrário da opção pelo assert, que não retorna nada por omissão

Invariantes no controlo de fluxo

Sem asserções

```
void foo() {
   for (...) {
     if (...)
        return;
   }
   // Nunca chega aqui!
}
```

Com asserções

```
void foo() {
   for (...) {
     if (...)
        return;
   }
   // Nunca chega aqui!
  assert false;
}
```

Cuidado especial: esta técnica **não deve ser usada** se o próprio compilador do Java for capaz de detectar automaticamente que o código não é atingível. E o compilador consegue detectar esta situação, em muitos casos (mas não em todos)

Pré-Condições, pós-condições e invariantes de classe

- Uma pré-condição é uma condição que se tem de verificar **antes** da execução de uma operação
  - Por convenção, violações a pré-condições de métodos públicos devem dar origem ao lançamento de excepções apropriadas
  - Apenas devemos usar asserções para testar pré-condições de métodos **privados**

```
/**
 * Ajusta o intervalo de refrescamento, que tem de ser legal.
 * @param  interval intervalo de refrescamento, em milisegundos
 */
private void setRefreshInterval(int interval) {
  assert interval > 0 && interval <= 1000/MAX_REFRESH_RATE : interval;
  ... // Ajusta o intervalo de refrescamento
}
```

Pré-Condições, **pós-condições** e invariantes de classe

- Uma pós-condição é uma condição que se tem de verificar **depois** da execução de uma operação
  - Podemos testar pós-condições relativas a métodos com qualquer visibilidade

```
/**
 * Retorna o perímetro de um quadrado
 * @param l é a largura
 * @param c é o comprimento
 * @return l * c
 * @throws ArithmeticException, se o resultado não for o esperado
 */
public double Perimeter() {
  double result = 2 * l + 2 * c;
  assert result == 2 * (l + c); // Exemplo exageradamente simples
  return result;
}
```

Pré-Condições, pós-condições e **invariantes de classe**

- Um invariante de classe é uma condição que se tem de verificar **sempre** em qualquer objecto dessa classe, em qualquer estado que se encontre
  - Excepto em estados transitórios entre dois estados consistentes, durante a execução de um método
    - Mas tem de se verificar quer antes, quer depois da execução do método
  - Exemplo: queremos ter uma árvore sempre balanceada

```
// Returns true if this tree is properly balanced
private boolean balanced() {
   ...
}
// Antes do retorno de qualquer método público, incluindo o
// construtor, poderíamos incluir a asserção seguinte.
// Mas, em geral, não é necessário testar também no início do método

assert balanced();
```

Quando não acrescentar asserções

- Verificação de argumentos em métodos públicos
  - Recorde-se que as asserções podem ser ligadas ou desligadas e que o programa deve funcionar bem sempre, **mesmo que elas estejam desligadas**
    - Os métodos públicos fazem parte do "contrato" de uma classe, que tem de ser obedecido, com ou sem excepções
    - Argumentos errados devem dar origem a uma excepção apropriada, por exemplo:
      - `IllegalArgumentException`
      - `IndexOutOfBoundsException`
      - `NullPointerException`
    - A falha da asserção "esconde" a excepção, o que dificulta a correcta detecção e eliminação do problema

Quando não acrescentar asserções

- Na realização de tarefas do programa obrigatórias para a sua correcta execução
  - Como as asserções podem estar inibidas, isso resultaria na não execução desse código fundamental!
    - Exemplo de **como não fazer:**

```
// Errado! A acção está definida como parte da asserção
assert names.remove(null);
```

    - Como resolver o problema:

```
// Correcto! A acção precede a asserção
boolean nullsRemoved = names.remove(null);
assert nullsRemoved;  // Este código funciona mesmo que
                      // as asserções estejam inibidas
```

Exercício

- Implemente uma interface Stack, que represente uma pilha de inteiros, com as operações
  - void push(int elem)
  - int pop()
- Use asserções para garantir que o respectivo código está correcto