
Programação Orientada Pelos Objectos
Excepções

2

Excepções?

Como acrescenta um elemento `v` a um vector `a`?


```
a[count++] = v;
```

- Na maioria das vezes, funciona sem problema
- Contudo, se o vector estiver cheio, gera um erro de execução!
  - O vector está cheio e não cabe mais nenhum elemento

Problema resolvido com uma ***abordagem preventiva***


```
if (count < N)
   a[count++] = v;
else
   // lidar com o array cheio
   // por exemplo, fazer resize
```

- Basta actuar de forma preventiva, com a ajuda de uma instrução condicional
  - É o que temos feito até agora!

Problemas levantados pela ***abordagem preventiva***

- Código poluído por testes relativos a situações raras
  - Esses testes prejudicam a compreensão do que se faz nas situações normais
- A biblioteca da linguagem pode não disponibilizar meios para testar preventivamente determinadas situações
  - Por exemplo, a falta de direitos de escrita de um ficheiro, ou o disco estar cheio.

Como funciona o tratamento de excepções?

- Em algumas situações, preferimos deixar o erro de execução ocorrer efectivamente
  - Lidamos com esse erro *à posteriori*, com a ajuda do mecanismo de tratamento de excepções

Podemos fazer bem melhor

- Há muito que a comunidade das linguagens de programação conhece uma forma de resolver este tipo de problemas bastante melhor:
  - Um **mecanismo de tratamento de excepções**

Não podemos programar métodos que nos devolvam um determinado código de erro?

- Poder, podemos
- Mas o código fica, normalmente, com o mesmo problema de legibilidade do código construído com a abordagem preventiva

O que ganhamos com um **mecanismo de tratamento de excepções**?

- Podemos escrever o código das situações normais "ignorando" as situações especiais
  - Podemos tratar as situações especiais numa zona separada do código, escolhida de acordo com as nossas conveniências
  - O ganho de legibilidade, em certos programas, é substancial

Qual é o preço a pagar pelo tratamento à posteriori?

- Necessitamos de uma linguagem que suporte o tratamento de excepções
  - O Java suporta-o
- Este mecanismo tem uma implementação complexa e envolve novos conceitos de tempo de execução, tais como:
  - Lançar excepção
  - Propagar excepção
  - Capturar excepção

O que é que um mecanismo de excepções nos oferece?

- 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 clara e efectiva
- Propagação de erros até à 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

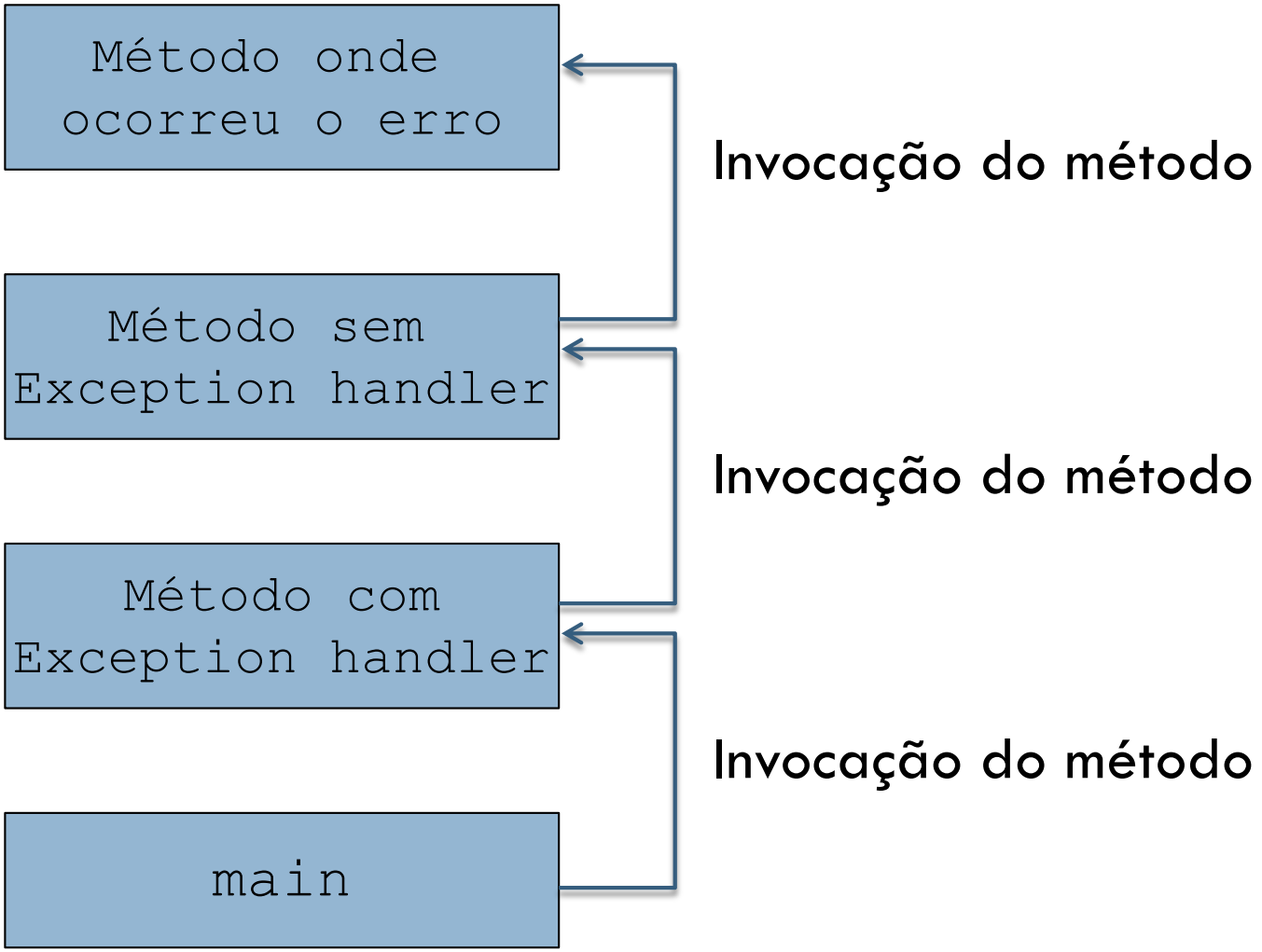
- Uma excepção é um evento pouco frequente, normalmente associado a um erro ou situação anormal, que é detectado por hardware ou software e necessita de algum tipo de processamento especial
  - Exemplos: divisão por zero, fim de ficheiro não esperado, dados inválidos, abertura de um ficheiro que não existe, falta de memória, acesso a um vector para além dos seus limites, etc.
- A ocorrência de uma excepção altera o fluxo de controlo normal do programa

Excepção

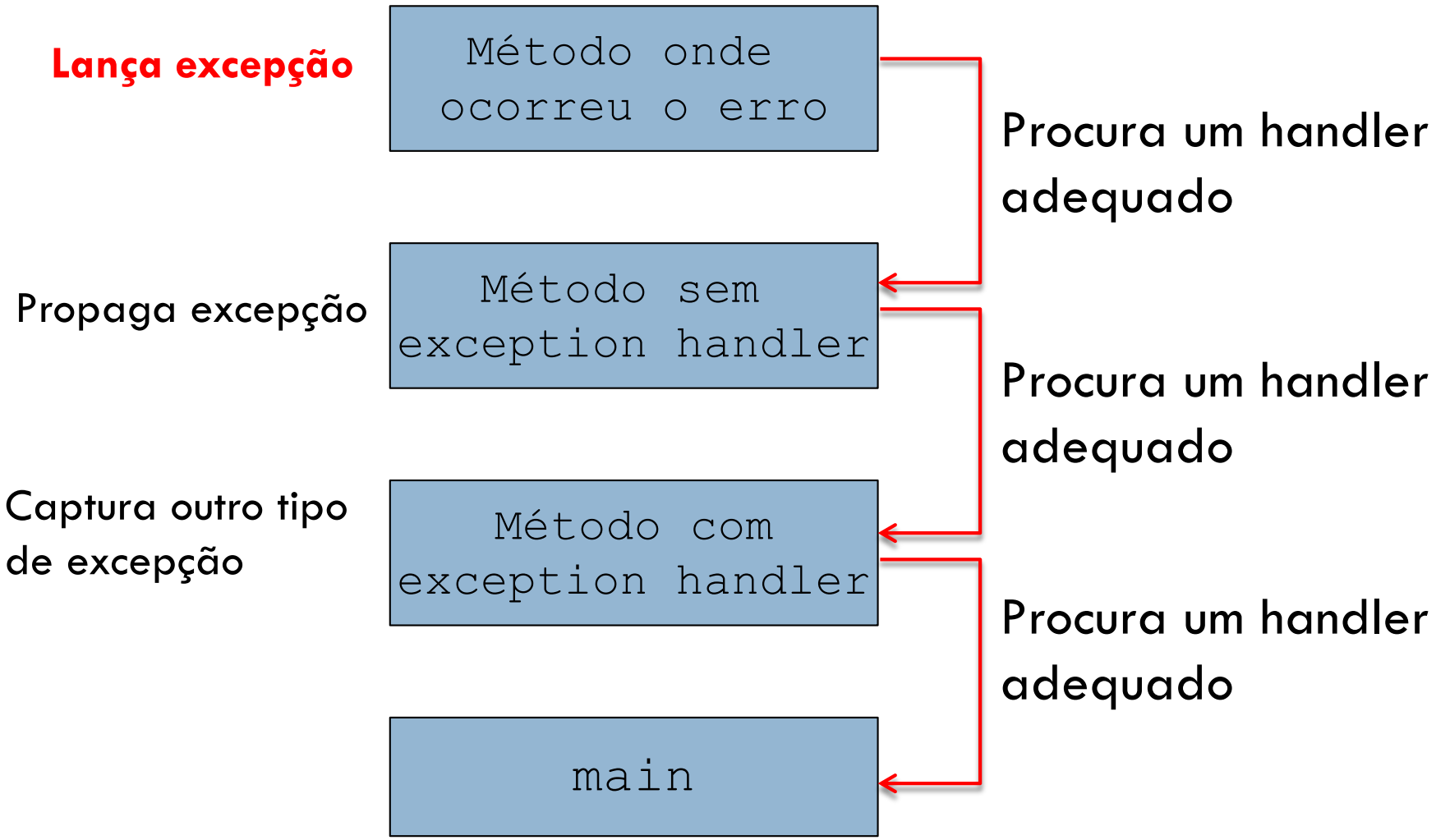
- Uma excepção em Java é um objecto com variáveis e métodos associados
  - Permite avaliar e processar a situação em causa
- Operações associadas a excepções
  - Criação da excepção
  - Lançamento da excepção
    - Programa notifica a ocorrência de um erro
  - Captura e processamento da excepção
    - Programa direciona o controlo para código de análise e processamento de erros

Gestão de uma excepção

A pilha de chamadas



Procurando o gestor da excepção



Lançamento e processamento de excepções

- Lançamento de uma excepção em Java
  - Ocorrência de uma excepção
    - É criado um objecto de excepção que é passado para o sistema *runtime*
  - Exemplo

```
throw new NullPointerException();
```

  - Se o método que lançou a excepção não a capturar, o método termina
- Após o lançamento da excepção a execução continua no gestor de excepções
- Processamento da excepção lançada
  - Execução de código especifico para o efeito
  - O objecto de excepção contém informação sobre o erro, 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) throws NegativeAmountException,
                                            SaldoInsuficienteException{
   if (amount < 0)
        throw new NegativeAmountException();
   if (amount > balance)
        throw new SaldoInsuficienteException();

   balance -= amount;
 }
}
```

Controlo e processamento de excepções


```
try {
   // Código que pode lançar excepções
   // seja com uma instrução throw ou
   // a invocação de um método que pode lançar excepções
} catch ( <ExceptionType1> <Obj1> ) {
    // Trata excepçoes do tipo <ExceptionType1>
} catch ( <ExceptionType2> <Obj2> ) {
    // Trata excepçoes do tipo <ExceptionType2>
} . . .
} finally {
    // Código a ser executado no final do bloco try
}
```

Recapitulando

- Executam-se as instruções que estão no bloco `try`
- Se não ocorrer nenhuma excepção, as cláusulas que constam na lista de `catch` não são executadas
- Se ocorrer uma excepção 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 por defeito do próprio 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, há o risco de ser omitida a execução de operações importantes
- Exemplo
  - 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`
  - ...

Propagação de erros na pilha de chamada

- Quando é lançada uma excepção, o sistema de execução do Java faz um pesquisa em sentido inverso na pilha de chamadas com o objectivo de encontrar métodos ou blocos que estejam associados à gestão ou tratamento dessa excepção

Excepção não detectada pelo código

- O gestor de excepções do java por defeito
  - 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: operador numero
Q para terminar o programa
Resultado = 0.0
+ palavra
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:909)
        at java.util.Scanner.next(Scanner.java:1530)
        at java.util.Scanner.nextDouble(Scanner.java:2456)
        at Main.doCalculation(Main.java:51)
        at Main.main(Main.java:18)
```

Implementação de excepções

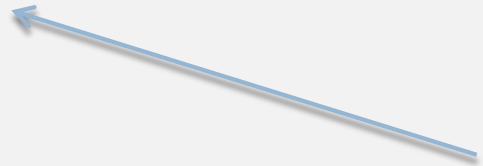
- O Java permite criar excepções:
  - Criar uma classe que estenda a classe `Exception` ou `RuntimeException`
  - Redesenhar a classe criada, adicionando variáveis de classe e construtores
- Na nova classe podemos
  - Invocar um dos métodos standards `e.getMessage()` e `e.printStackTrace()`
  - Escrever uma mensagem com informação própria da classe
  - A invocação da classe de excepção segue os mesmos princípios de outras classes de excepção
- Note-se que pode ser conveniente tomar alguma ação corretiva em função da excepção gerada

Alguns conselhos

- Devemos usar o mecanismo das excepções com ponderação
- A ordem das cláusulas `catch` é importante
  - É executada a primeira cláusula que coincide com a excepção
  - Colocar a mais específica em primeiro lugar

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

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



Alguns conselhos

- Em geral, o código de lançamento e de captura da excepção estão em métodos distintos

```
public void methodB() {

      try {
           ... methodA() ...
      }
      catch ( MyException exception ) {
         // Tratar excepção
         ...
      }
      ...
}
```

```
public void methodA() throws MyException {
      . . .
      throw new MyException();
      . . .
}
```

Tipos de excepções

Tipos de excepções

- Excepções verificadas
  - O compilador verifica se são ignoradas pelo programa
    - Se forem ignoradas, origina um erro de compilação
  - Associadas a circunstâncias externas que o programador não pode prever
  - A maior parte destas excepções estão relacionadas com operações de entrada/saída
- Excepções não verificadas
  - Não são sujeitas à 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 inesperadas
  - Exemplo
    - `Scanner.nextInt()` lança a excepção não verificada `InputMismatchException` quando o utilizador, incorretamente, fornece um valor não inteiro
- Devemos considerar excepções verificadas sobretudo quando se está a lidar com ficheiros
  - Exemplo
    - Na leitura de um ficheiro com a classe `Scanner` 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:
  - Capturar a excepção
  - Propagar a excepção
    - Usar 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) {
    try {
         reader = new FileReader(filename);
         Scanner in = new Scanner(reader);
         ...
    }
    catch (ClassNotFoundException exception ) { ... }
    catch (IOException exception ) { ... }
}
```

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, fora do controlo dos utilizadores do programa
    - Por exemplo, erros de memória ou falha do disco rígido
- **Exception**
  - Para situações que os utilizadores podem gerir
    - Por exemplo, divisão por zero ou acesso fora dos limites de vectores

Agrupamento e diferenciação de erros

- Organizar o tratamento de erros segundo a hierarquia de classes de excepções
- Exemplo: `java.io.IOException` e descendentes
  - `IOException` é a classe mais genérica, associada aos 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 tipo ou em alguma das superclasses respectivas
  - Exemplo: `IOException` na cláusula catch, captura todas as excepções de I/O, incluindo `FileNotFoundException`, `EOFException`

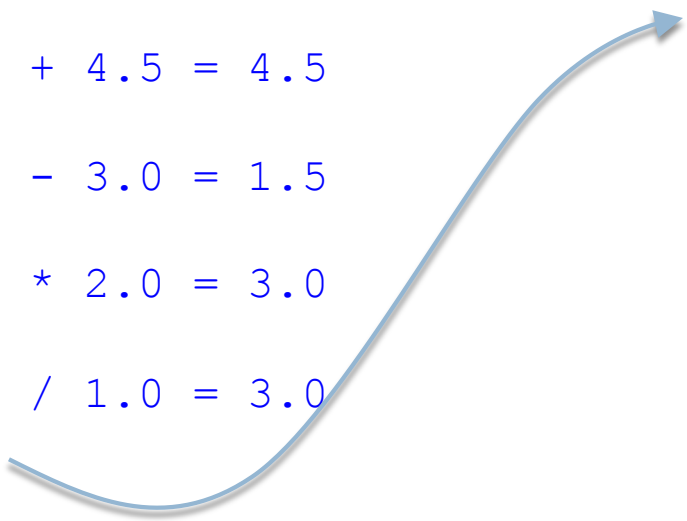
Uma calculadora simples

Calculadora simples

- Implementar calculadora simples mas robusta 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


```
Formato de entrada: operador numero
Q para terminar o programa
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
Operador desconhecido: q
Tente mais uma vez ...

Formato de entrada: operador numero
Q para terminar o programa
Resultado = 0.0
+ 4
Resultado + 4.0 = 4.0
q 2
Operador desconhecido: q
Ja chega! Tente noutra altura.
Fim do programa.
```

Exemplos de traço do programa

```
Formato de entrada: operador numero
Q para terminar o programa
Resultado = 0.0
* 2
Resultado * 2.0 = 0.0
+ 45
Resultado + 45.0 = 45.0
/ 0.00001
Divisao por zero.
Fim do programa.
```

```
Formato de entrada: operador numero
Q para terminar o programa
Resultado = 0.0
+ 45.2
Resultado + 45.2 = 45.2
Q
Resultado final e 45.2
Fim do programa.
```

Diagrama de classes

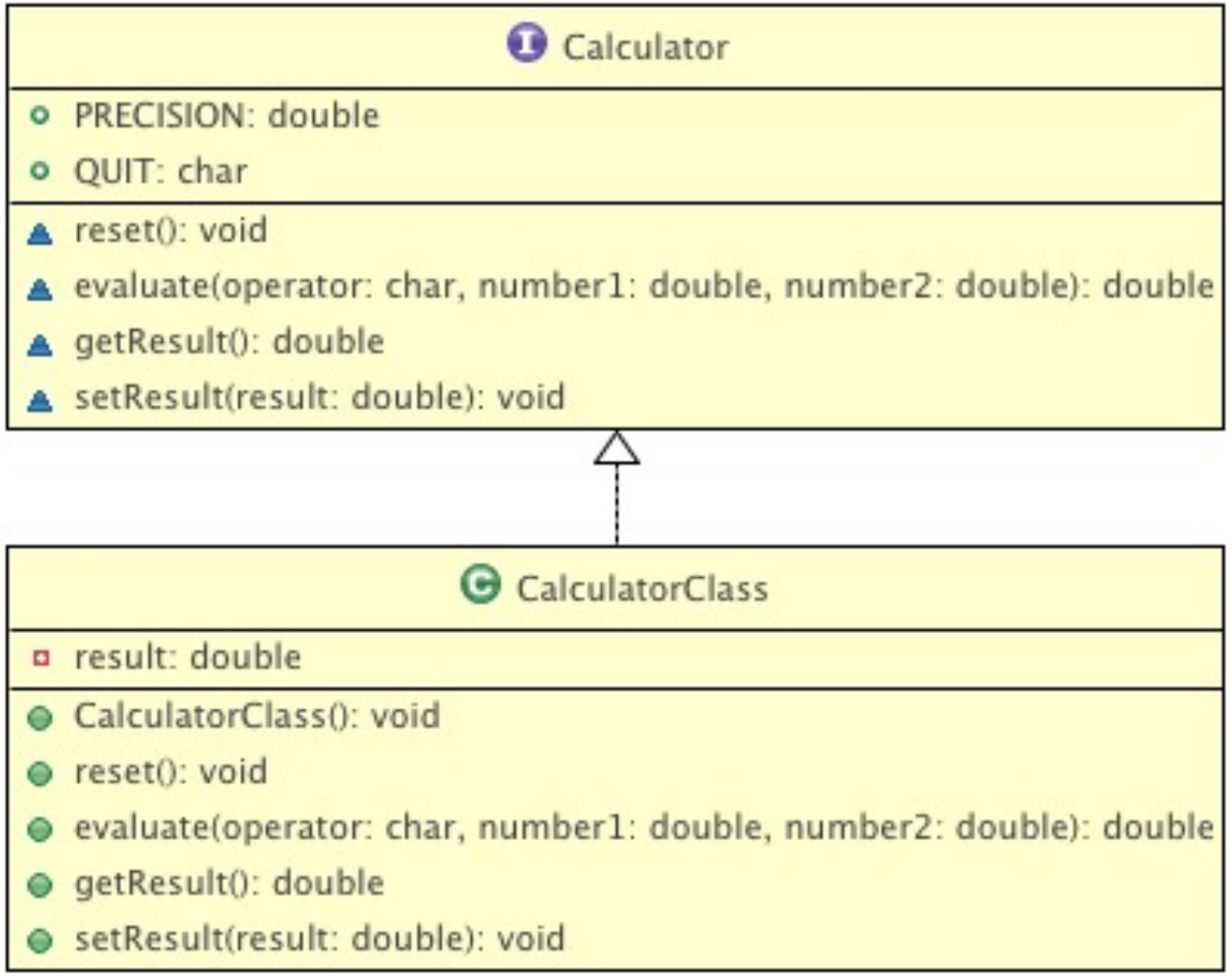
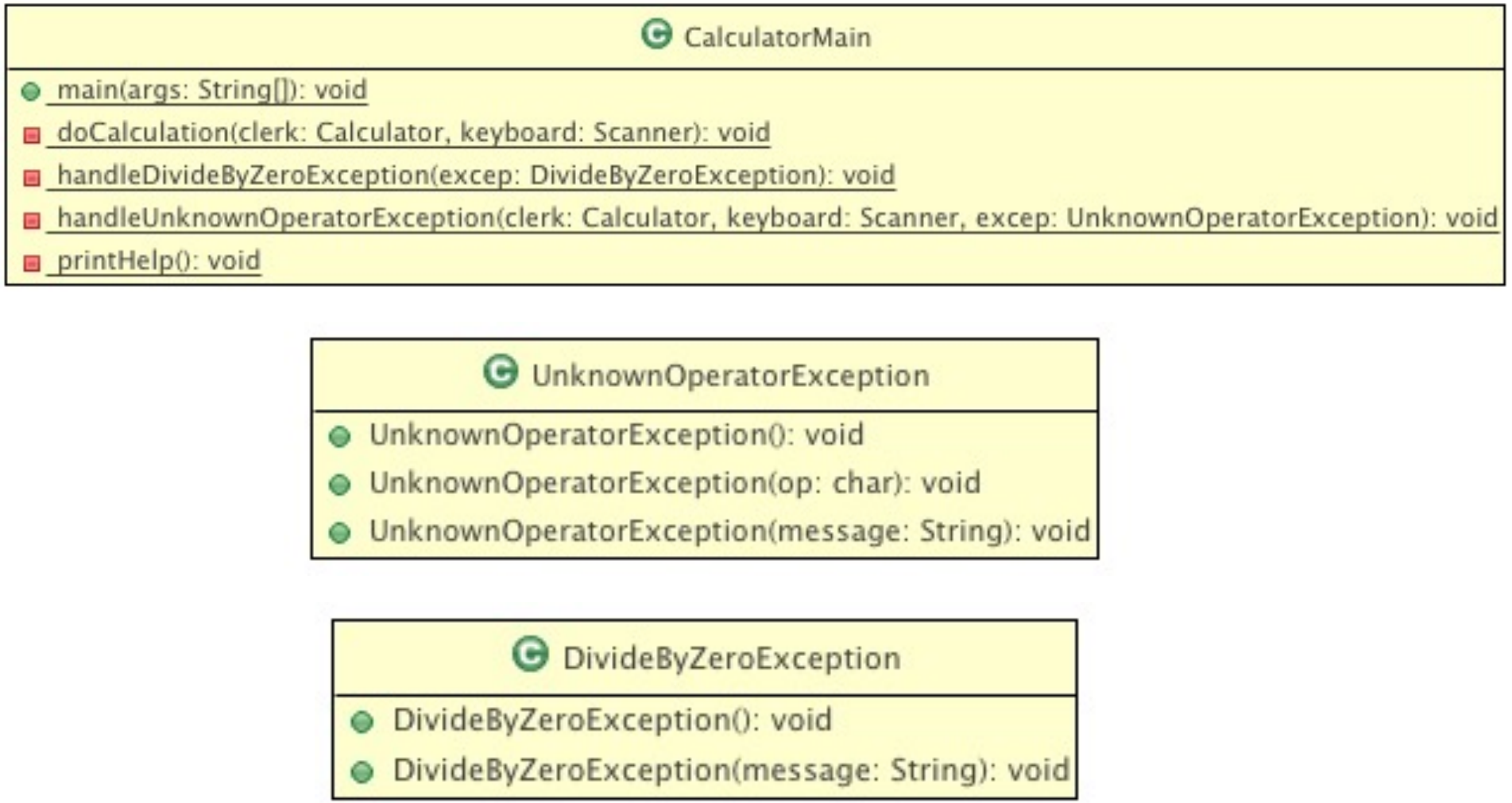


Diagrama de classes



A interface **Calculator**


```
package poo;

public interface Calculator {
   // Numbers this close are treated as if equal to zero
   final double PRECISION = 0.000001;
   final char QUIT = 'Q';

   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 ( Math.abs(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();
    }

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

A classe **UnknownOperatorException**


```
public class UnknownOperatorException extends Exception {
        private String operator;
        public UnknownOperatorException( ) {
                super();
                operator = "";
        }
        public UnknownOperatorException(String message) {
                super(message);
                operator = "";
        }
        public UnknownOperatorException(char op) {
                super();
                operator = op + "";
        }
        public String getOperator() {
                return operator;
        }
}
```

A classe **MainCalculator**


```
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 **MainCalculator**


```
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( );  // may launch an exception !!!
         result = clerk.evaluate(nextOp, result, nextNumber);
         clerk.setResult(result);
         System.out.println("Resultado" + nextOp + nextNumber + "=" + result);
     }
}}
```

A classe **MainCalculator**


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

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.");
  } catch(DivideByZeroException excep3) {
      handleDivideByZeroException(excep3);
  }
}
```

Excepções e pré-condições

- As excepções permitem-nos tratar situações inesperadas no código
  - podemos então tornar os nossos programas mais robustos e não dependentes do "mundo seguro"
- Nos comentários javadoc em vez de especificar pré-condições podemos especificar em que condições é que os métodos levantam excepções

```
/**
 * ...
 * @throws NoMoreElementsException if !hasNext()
 **/
E next() throws NoMoreElementsException;
```