PROGRAMAÇÃO ORIENTADA PELOS OBJECTOS

Resumo

2

Vamos arrumar ideias...

Resumo de alguns aspectos fundamentais da programação orientada pelos objectos

Mecanismos de suporte à extensibilidade

- Herança
  - também conhecida como herança de classes (*subclassing*) ou herança de implementações (*implementation inheritance*)
  - Torna as classes entidades extensíveis
    - O código da superclasse é reutilizado (e reinterpretado) no contexto da subclasse, sem que tenha de ser reescrito
    - Por vezes é preciso redefinir alguns dos métodos herdados e nesse caso escrevem-se novas implementações desses métodos na subclasse (não se altera o código dos métodos originais)

Mecanismos de suporte à extensibilidade

- Subtipo (*subtyping*)
  - também conhecido como polimorfia de inclusão
  - Permite a escrita de código abstracto, que funciona tão bem com as classes existentes como com novas classes a criar no futuro
- Podemos distinguir herança de subtipos da seguinte maneira:
  - Um tipo é a **especificação** dum comportamento
    - Tanto as classes como as interfaces são tipos
  - Uma classe é a **implementação** dum comportamento
  - Além de ser um tipo, uma classe é também uma implementação concreta desse tipo
    - Para um dado tipo, pode haver um número arbitrariamente grande de implementações diferentes

Recapitulando: mecanismos de suporte à extensibilidade

- Envio de mensagens
  - Enviar uma mensagem a um objecto é sinónimo de chamar uma operação desse objecto
    - Metáfora originária da linguagem OO Smalltalk
  - Incorpora uma análise implícita da verdadeira classe (*runtime type*) dum objecto, feita pela infraestrutura do Java
    - funciona tão bem com as classes existentes como com novas classes a criar no futuro
    - Não depende do tipo da referência para o objecto, que pode ser um super-tipo (no limite, pode ser `java.lang.Object`)
    - O código que executa em resposta à mensagem é o da verdadeira classe do objecto que recebeu a mensagem

Mecanismos de suporte à extensibilidade

- Modularidade
  - Um módulo é uma parte dum sistema conceptualmente independente do sistema
  - Possui uma interface bem definida e os restantes módulos do sistema usam o módulo através dessa interface
    - Um módulo é o equivalente no *software* de módulos duma aparelhagem de alta fidelidade, e.g., amplificador, sintonizador de rádio, colunas de som, leitor de SACDs, de DVDs, de cassetes, etc.
  - Num programa a executar, os módulos são os objectos
  - Quando raciocinamos sobre a estrutura estática dum sistema, os módulos são as classes e interfaces

Mecanismos de suporte à extensibilidade

- A modularidade permite-nos obter diversas vantagens
  - **Abstracção.** Se a interface dum módulo estiver bem concebida, podemos abstraír-mo-nos dos detalhes da sua implementação
    - Apenas temos de aprender os conceitos/detalhes da interface
    - Para essa aprendizagem, a documentação do módulo desempenha um papel importante
  - **Localização.** Todo o código (ou texto fonte) relacionado com um conceito do problema (ou por vezes, da solução de programação, e.g., `FileReader`, `Random`) num **único local**, coerente e não misturado com código relacionado com outras funcionalidades

Mecanismos de suporte à extensibilidade

- A modularidade permite-nos obter diversas vantagens
  - **Reutilização**: certos conjuntos de responsabilidades ou funcionalidades prestam-se para ser usados em múltiplos pontos dum programa e múltiplas situações
    - Essas funcionalidades só podem ser utilizadas facilmente em pontos diferentes se não se encontrarem dispersas por múltiplas partes do programa, i.e., se estiverem **modularizadas**
  - **Desacoplamento.** Se o sistema tiver sido concebido para ser extensível, é mais fácil **acoplar** e **desacoplar** módulos, ou substituir uma implementação por outra
    - recorde as aulas anteriores sobre extensibilidade

Algumas heurísticas

- Usar correctamente os mecanismos de herança, sub-tipo, envio de mensagens e modularidade
- Procurar tornar os nossos módulos **extensíveis**
  - Evitar testar explicitamente a classe concreta dos objectos
  - Descubra os conceitos abstractos no seu problema
    - Abstractos no sentido que são comuns a qualquer implementação
- Apenas o código das classes deve ser concreto
- Cada classe preocupa-se com as suas tarefas e deixa as tarefas das outras classes para as outras classes
  - Este princípio é conhecido como *separation of concerns* (separação de facetas, ou conceitos, ou abstracções)

Recapitulando: o que podemos fazer numa sub-classe?

- Uma sub-classe herda todos os membros públicos e protegidos da sua super-classe, independentemente de estarem ou não declarados no mesmo **`package`**
  - Quando uma sub-classe está no mesmo package que a super-classe, também herda os membros acessíveis às classes do package
  - Podemos usar os membros herdados tal como estão
  - Podemos acrescentar novos membros
  - Podemos substituir a implementação dos métodos herdados
  - Podemos ocultar ou tapar as variáveis herdadas

Recapitulando: o que podemos fazer numa sub-classe?

Membros de dados (variáveis e constantes)

- Os membros de dados herdados podem ser usados directamente, tal como quaisquer outros membros de dados definidos na sub-classe
- É possível declarar um campo na sub-classe com o mesmo nome de um campo na super-classe, assim **tapando ou ocultando** o campo da super-classe
  - Mas embora possível é **mau estilo** e não recomendado!
- É possível declarar novos membros de dados na sub-classe, que não estejam na super-classe

Recapitulando : o que podemos fazer numa sub-classe?

Métodos e construtores

- Os métodos herdados podem ser usados directamente
- Pode-se escrever um novo método de numa sub-classe com a mesma assinatura de um método da super-classe, reimplementando assim o método da super-classe na sub-classe
  - O método da super-classe fica escondido
- Podem-se declarar novos métodos na sub-classe
- Pode-se escrever um construtor na sub-classe que invoque o construtor da super-classe
  - A invocação é feita com a palavra reservada **`super`**

O que significa ser sub-tipo

- Os sub-tipos devem satisfazer o princípio da substituição
  - Regra da assinatura
    - As assinaturas dos métodos têm de ser compatíveis e os sub-tipos têm de disponibilizar todos os métodos dos super-tipos
  - Regra dos métodos
    - As chamadas aos métodos dos sub-tipos têm de provocar um comportamento semelhante ao esperado nos super-tipos
  - Regra das propriedades
    - Os sub-tipos têm de preservar todas as propriedades dos super-tipos

Programa


Metodologia
Especificação de interfaces e respectiva documentação
Desenho com classes
Polimorfia de interfaces
Herança
Polimorfia de Herança
Programação genérica
Tratamento de excepções
Utilização de tipos genéricos
Asserções
Testes
Consolidação

PROGRAMAÇÃO ORIENTADA PELOS OBJECTOS

Excepções

16

Erros em programação

Quando as coisas correm mal

- Erros decorrentes de actividades humanas existirão sempre
- Os erros de sintaxe de programação são detectados durante a compilação e 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
- Erros associados ao ambiente de execução
  - Exemplo: corte da ligação de rede, erro de escrita em disco, etc.
- Erros lógicos e situações inesperadas devem ser detectadas e geridas convenientemente

Exemplo de falha grave de software

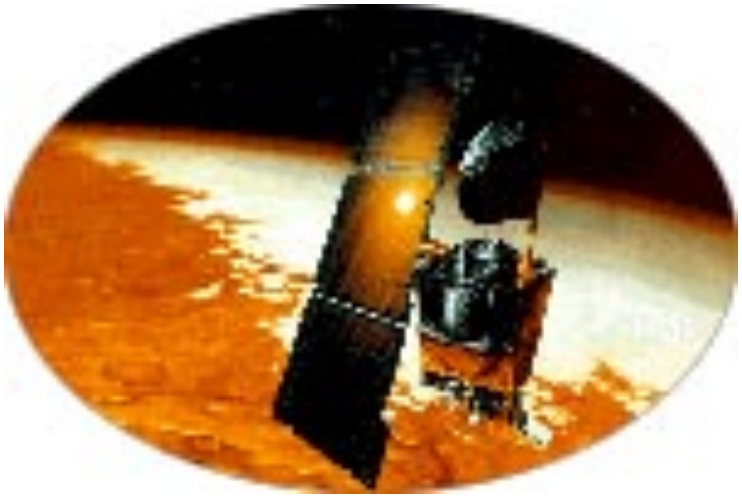
"A Bug and a Crash (by James Gleik)

It took the European Space Agency 10 years and $7 billion to produce Ariane 5, a giant rocket capable of hurling a pair of three-ton satellites into orbit with each launch and intended to give Europe overwhelming supremacy in the commercial space business.

All it took to explode that rocket less than a minute into its maiden voyage last June, scattering fiery rubble across the mangrove swamps of French Guiana, **was a small computer program trying to stuff a 64-bit number into a 16-bit space.**

"

Exemplo de falha grave de software



Mystery of Orbiter Crash Solved **(by Kathy Sawyer)**

NASA's Mars Climate Orbiter was lost in space last week because engineers failed to make a simple conversion from English units to metric, an embarrassing lapse that sent the $125 million craft fatally close to the Martian surface, investigators said yesterday.

…

Mundo seguro vs. mundo inseguro

- Assumindo um "mundo seguro" podemos programar assumindo que o código é usado correctamente
  - por exemplo antes de chamar um método as respectivas pré-condições são validadas
- Contudo muitas vezes temos de assumir um "mundo inseguro" e desenvolver código mais defensivo
  - o desenvolvimento deve antecipar a existência de erros

Erros e qualidade de programação

- 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
  - Retorno válido das operações pode ser confundido com um código de erro e vice-versa
  - Se não analisarmos conveniente tal código de retorno, é possível não detectar 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 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

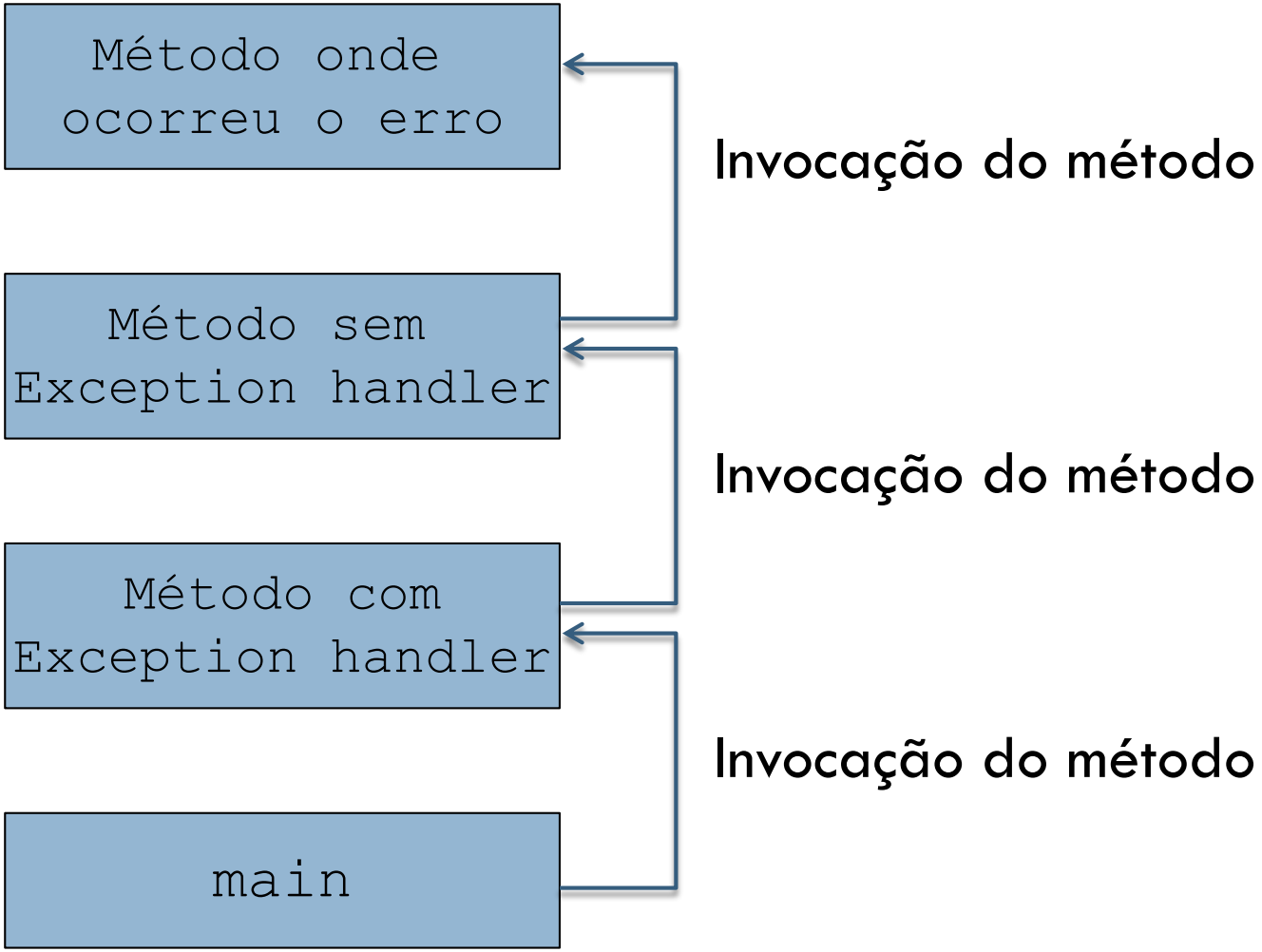
- Uma excepção é um evento pouco frequente, normalmente associado a um erro ou situação anormal, que é detectado por hardware ou software e necessita de algum tipo de processamento especial
  - Exemplos: divisão por zero, fim de ficheiro não esperado, dados inválidos, abertura de um ficheiro que não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

Excepção

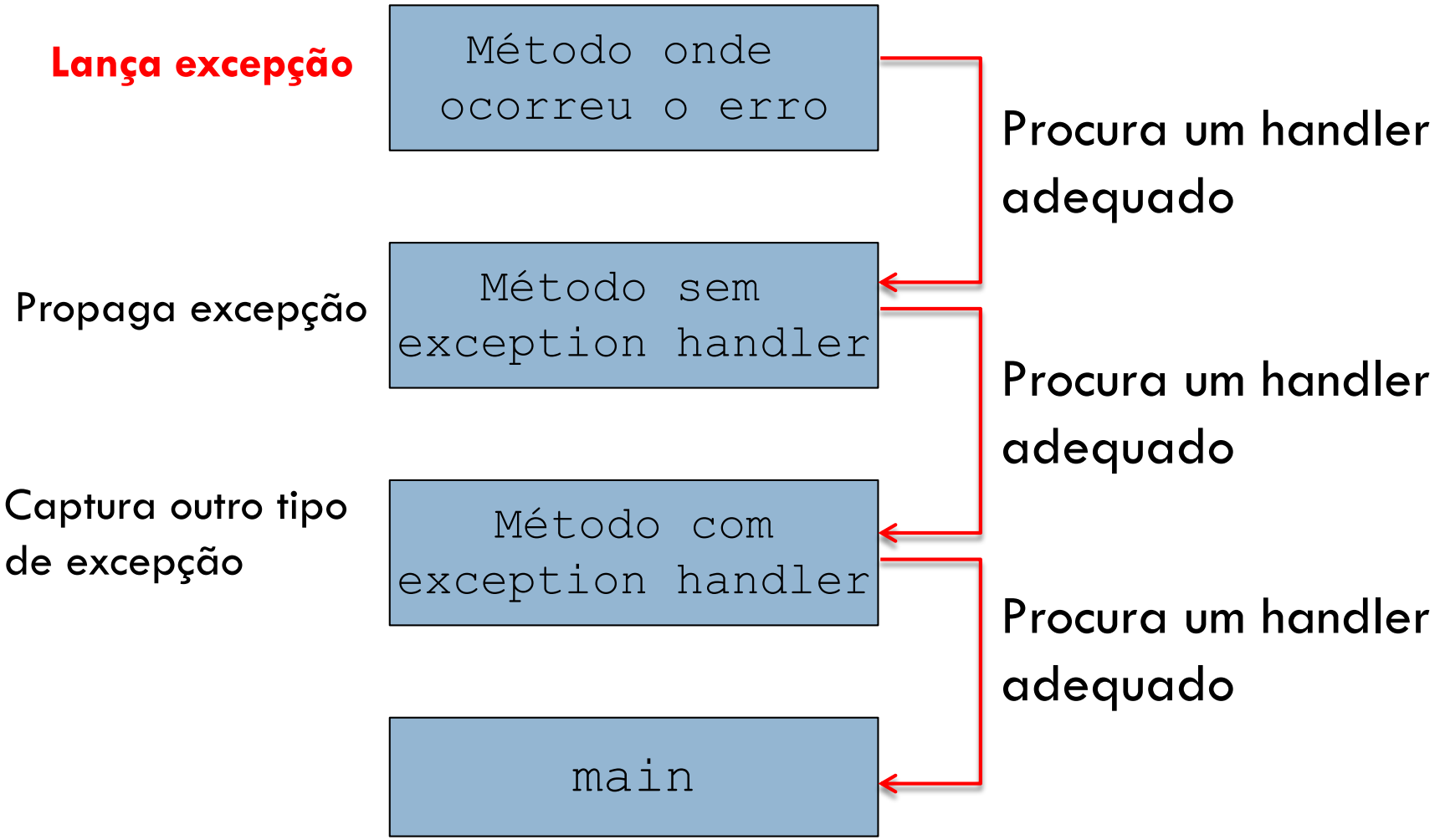
- Uma excepção em Java é um objecto com variáveis e métodos associados
  - Permite avaliar e processar a situação em causa
- Operações associadas a excepções
  - Criação da excepção
  - Lançamento da excepção
    - Programa notifica a ocorrência de um erro
  - Captura e processamento da excepção
    - 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 SaldoInsuficienteException{
   if (amount > balance) {
       throw new SaldoInsuficienteException();
   }
   balance -= amount;
 }
}
```

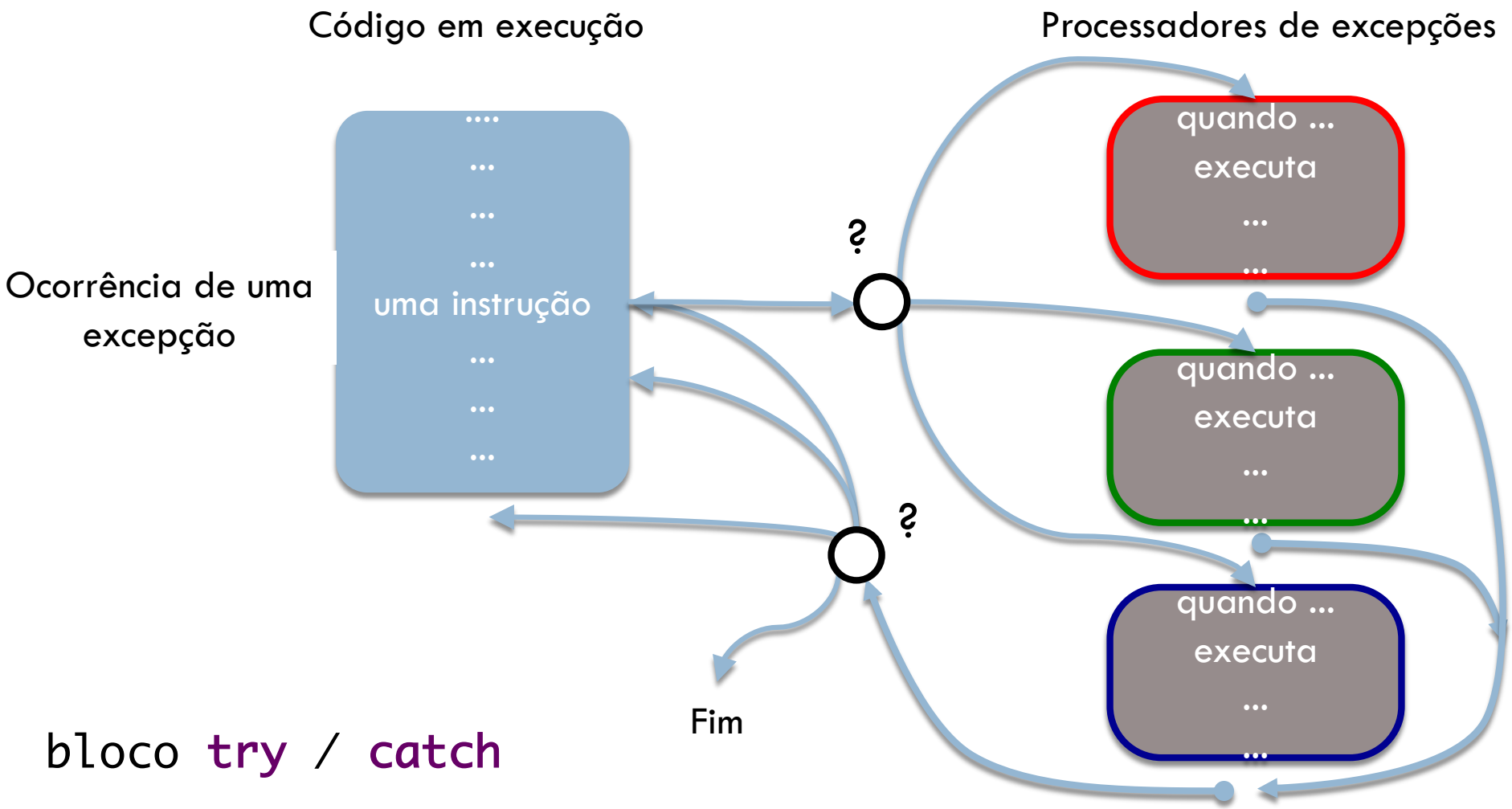
Controlo e processamento de excepções

- Em Java, as excepções são geridas através de um bloco **try**, das cláusulas **catch** e da cláusula **finally**
  - O bloco try termina normalmente ou com a detecção de uma 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 (*default exception handling*)
  - As cláusulas catch são testadas relativamente à excepção gerada, com base na hierarquia de classes de excepções
  - A cláusula finally, se existir, será sempre executada

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
}
```

Controlo de uma excepção



bloco `try` / `catch`

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 desta classe de excepção pelo código 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("My own message");
      . . .
}
```

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`. 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
    - ...
  - ...

- *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

checked
unchecked

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`