Programação Orientada por Objectos

Tratamento de Erros - Exceções

Prof. Rui César das Neves, Prof. José Cordeiro

Departamento de Sistemas e Informática

Escola Superior de Tecnologia de Setúbal – Instituto Politécnico de Setúbal

2014/2015

Sumário

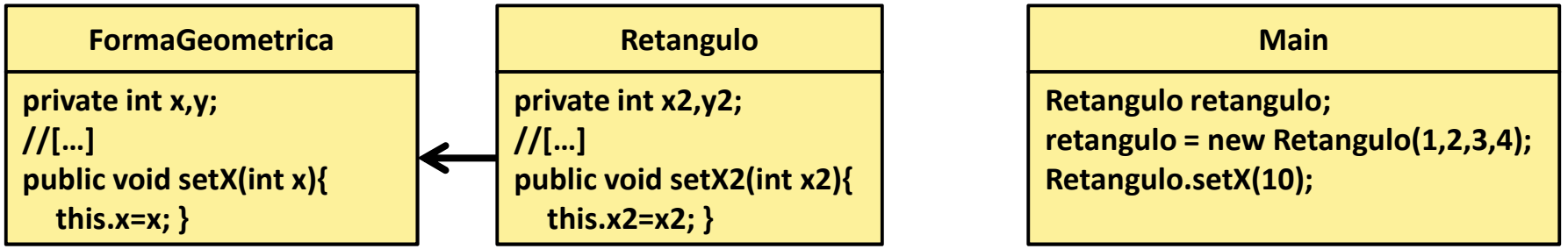
- Motivação: Programação Defensiva
  - Prever possíveis e potenciais erros.
  - Até que ponto deve um "servidor" verificar as chamadas?
  - Como reportar erros?
  - Como é que um cliente pode antecipar uma falha?
  - Como é que um cliente deve lidar com uma falha?

- Exceções
  - O que são
  - Para que servem
  - Lançamento de Exceções (instrução **`throw`**)
  - Tratamento de Exceções (instruções **`try/catch/finally`**)
  - Criação de novas Exceções (por herança da classe **`Exception`** ou **`RunTimeException`**)

Programação Defensiva

- Algumas origens de erros:
  - Implementação incorreta (erros semânticos)
    - Não está de acordo com a especificação.

      - Exemplo: **`setX(int x)`** deve deslocar a origem do retângulo, deslocando assim o retângulo, mas o que está a fazer é a redimensionar o retângulo!



      - Exemplo: O programador pensa que o código funciona de determinada forma mas de facto enganou-se a escrever o código e o programa está a funcionar de forma diferente:

```
// O programador pensa que escreveu:
public Ponto getPosicao() {
    return new Ponto (getX(),getY());
}
```

```
// Mas de facto escreveu:
public Ponto getPosicao() {
    return new Ponto (getX(),getX());
}
```

Programação Defensiva

- Algumas origens de erros:
  - Chamada incorreta de um objeto
    - Por exemplo, um índice inválido.

Main

```
public static void main(String[] args) {
    FormaGeometrica[] formas;
    formas = new FormaGeometrica[4];

    formas[0] = new Circulo(new Ponto(4,8), 3);
    formas[1] = new Circulo(new Ponto(28,10), 4);
    formas[2] = new Retangulo(new Ponto (11,0), new Ponto (23, 6));
    formas[3] = new Retangulo(new Ponto (18,0), new Ponto (21,4));
    double areaTotal = 0;
    for (int i=1; i<10; i++)  //posições 4 a 9 não existem
          areaTotal += formas[i].getArea();
    System.out.println("Area Total: " + areaTotal);
}
```

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 4
        at formas.Main.main(Main.java:22)
Java Result: 1
```

Programação Defensiva

- Algumas origens de erros:
  - Estado de um objeto inconsistente ou inapropriado.
    - Por exemplo tentar aceder a um objeto que não foi criado.

Desenho

```
public class Desenho {
    FormaGeometrica[] formasNoDesenho; // = new FormaGeometrica[1000];
    public FormaGeometrica[] getFormasNoDesenho() {
        return formasNoDesenho; }
    public void setFormasNoDesenho(FormaGeometrica[] formasNoDesenho) {
        this.formasNoDesenho = formasNoDesenho;}
    public void addForma(FormaGeometrica novaForma){
        int i = 0;
        while (formasNoDesenho[i] != null)
            i++;
        formasNoDesenho[i+1] = novaForma;
    }
}
```

TestarDesenho

```
public class TestarDesenho {
    public static void main(String[] args) {
    FormaGeometrica forma;
    forma = new Circulo(new Ponto(4,8), 3);
    Desenho novoDesenho = new Desenho();
    novoDesenho.addForma(forma);
    }
}
```

```
Exception in thread "main" java.lang.NullPointerException
        at formas.Desenho.addForma(Desenho.java:15)
        at TestarDesenho.main(TestarDesenho.java:13)
Java Result: 1
```

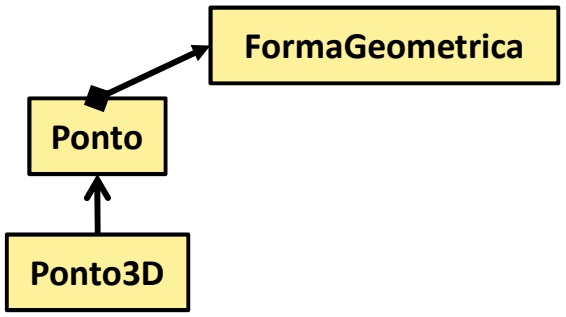
Programação Defensiva

- Nem sempre os erros são do programador:
  - Um URL incorretamente introduzido
  - Uma falha na rede
  - O processamento de ficheiro é particularmente atreito a falhas:
    - Ficheiros inexistentes.
    - Ausência de direitos de acesso.

- Interação “Cliente”-”Servidor”.
  - Deverá um servidor assumir que os clientes são bem comportados?
  - Ou assumir que serão potencialmente hostis?
  - Os tipos de implementação serão radicalmente diferentes.

Programação Defensiva

- O servidor deve verificar os argumentos oriundos do cliente.
  - Os argumentos representam uma das grandes vulnerabilidades dos objetos servidores:
    - Os argumentos dos construtores inicializam o estado do objeto
    - Os argumentos dos métodos muitas vezes influenciam o comportamento
  - A verificação dos argumentos é uma medida defensiva:




```
public FormaGeometrica(Ponto posicao) {
        if ((posicao !=null) && !(posicao instanceof Ponto3D))
                this.posicao = posicao;
        else
                this.posicao = new Ponto(0,0);
}
```

```
public FormaGeometrica(Ponto posicao) {
        if ((posicao !=null) && !(posicao instanceof Ponto3D))
                this.posicao = posicao;
}
```

Programação Defensiva

- Quem notificar sobre argumentos incorretos?
  - O utilizador?
    - Há um utilizador humano?
    - Ele pode resolver o problema?
  - O objeto cliente?
    - Retornar um valor de diagnóstico?
    - Veremos mais à frente que o Java oferece uma solução especificamente para deteção e tratamento de erros
  - Retornar um valor de diagnóstico:

```
public boolean addForma(FormaGeometrica novaForma){
        if (formasNoDesenho == null)
                return false;
        for (FormaGeometrica forma: formasNoDesenho)
                if (forma == null) {
                        forma = novaForma;
                        return true;
                }
        return false;
}
```

TestarDesenho

```
public class TestarDesenho {
    public static void main(String[] args) {
    FormaGeometrica forma;
    forma = new Circulo(new Ponto(4,8), 3);
    Desenho novoDesenho = new Desenho();
    boolean inserida = novoDesenho.addForma(forma);
    System.out.println(inserida);
    }
}
```

Programação Defensiva

- Testar o valor de retorno
  - Tentar recuperar do erro
  - Evitar a falha do programa.
- O valor de retorno é ignorado
  - Perfeitamente possível, não existe mecanismo para o evitar.
  - Provavelmente conduzirá a uma falha do programa.

- As Exceções são preferíveis:
  - São uma característica especial de algumas linguagens de programação
  - Não diminuem o esforço para detetar, reportar e lidar com erros … mas, comparativamente com o teste do valor de retorno, as exceções:
    - Separam claramente o processamento dos dados do código responsável pelo tratamento das situações anómalas.
    - Propagam-se automática e ascendentemente pela pilha de execução dos métodos até encontrarem um objeto que saiba processá-las.
    - Podem ser organizadas em hierarquias de acordo com o tipo de problema.
    - Os erros não podem ser ignorados no cliente porque o fluxo de controlo é interrompido.
    - Encorajam a implementação de medidas de recuperação

Exceções – O que são?

- O que é uma Exceção?
  - É um evento que ocorre durante a execução de um programa e que interfere no fluxo normal das instruções desse programa.
  - Uma Exceção é um sinal gerado pela máquina virtual de Java em tempo de execução do programa, que é comunicado ao programa indicando a ocorrência de um **erro recuperável**.
    - Em Java, a ocorrência de erros durante a execução de um programa não significa necessariamente que o programa termine.
  - É um mecanismo de tratamento de erros comum a várias linguagens de programação.
    - O Java possui um mecanismo próprio, baseado em exceções, para indicar partes críticas num programa e recuperar eventuais erros ocorridos nestas partes.
    - O mecanismo não interrompe a execução do programa, mas passa o controlo da sua execução para o tratamento de erros.

Exceções – Origens e Efeitos

- Quando ocorre uma falha durante a execução de um método:
  - A JVM deteta a falha.
  - A JVM cria um objecto da classe relativa à falha ocorrida
    - O objeto é de uma subclasse de **`Exception`**.
    - Este objecto contém informações sobre a falha ocorrida (o seu tipo e o estado do programa quando ocorreu a falha)
  - A JVM lança o objeto para o programa.
  - A partir deste momento, o mecanismo de tratamento de exceções do Java responsabiliza-se por encontrar o código que trata a falha ocorrida.

Exceções – Origens e Efeitos

- São vários os tipos de falhas que podem gerar uma excepção, nomeadamente:
  - Tentar aceder a uma tabela fora de seus limites,
  - Tentar abrir um arquivo inexistente,
  - Tentar ler um ficheiro para além do fim deste,
  - Tentar abrir um URL inexistente,
  - Tentar dividir por zero,
  - Tentar calcular a raiz quadrada de um número negativo.

- Os efeitos da geração de uma exceção são:
  - O método que a lança termina prematuramente
  - Não é retornado nenhum valor
  - O controlo não passa para o ponto em que método foi chamado
    - Portanto o "cliente" (o método invocador) não pode continuar a executar e ignorar a exceção

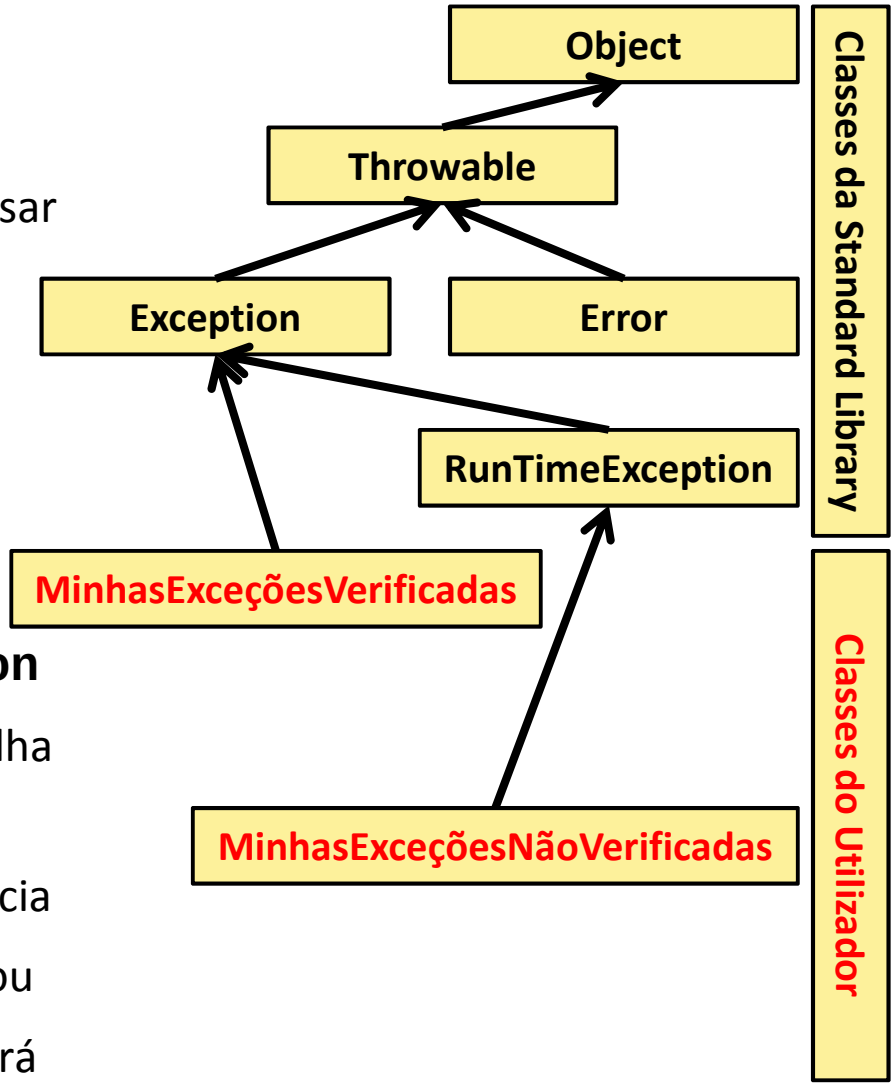
Exceções - Lançamento

- **Lançamento explicito de uma exceção pelo programador:**
  - O programador cria um objeto exceção:
    - `new ExceptionType("...");`
  - O programador lança a exceção:
    - `throw ...`
  - Documentação Javadoc:
    - `@throws ExceptionType texto`
- **Lançamento implícito de uma exceção pela JVM:**
  - Ao deparar com um erro em tempo de execução a JVM lança uma exceção:
    - No exemplo, se o array já estiver completamente preenchido, o método **`addForma`** irá tentar escrever numa posição que excede o limite superior do array levando a que JVM gere uma exceção do tipo: **`ArrayOutOfBoundsException`**.

```
/**
 * Adiciona um forma geometrica ao desenho
 * @param novaForma a FormaGeometrica
 * a adicionar ao desenho.
 * @throws NullPointerException se o array com
 * o desenho ainda não foi criado.
 */
public void addForma(FormaGeometrica novaForma){
      if (formasNoDesenho == null)
            throw new NullPointerException(
                  "array vazio em addForma" );
      int i=-1;
      do
         i++;
      while (formasNoDesenho[i] != null);
      formasNoDesenho[i] = novaForma;
      }
}
```

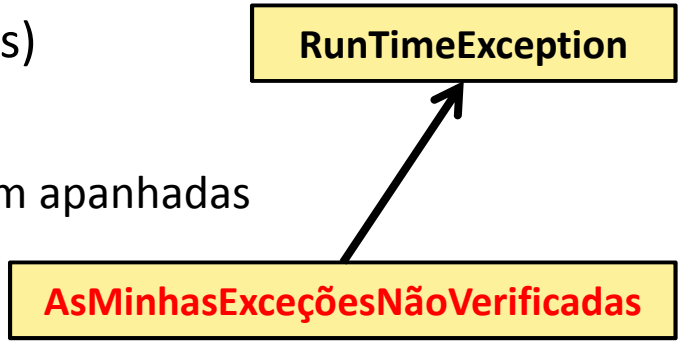
Exceções – Hierarquia de Classes

- **Exceções Verificadas (checked)**
  - São subclasses de **`Exception`**
  - O compilador obriga o programador a usar blocos **`try-catch`** (ver mais à frente).
  - Usam-se para falhas em que é aconselhável (obrigatório?) fazer a verificação do erro.
- **Exceções não verificadas (unchecked)**
  - São subclasses de **`RuntimeException`**
  - O compilador não obriga a verificar a falha
  - O programador pode optar por não verificar a falha ignorando a sua existência
  - Seja porque a falha é pouco frequente ou porque se considera que ela não ocorrerá



Exceções – Não Verificadas

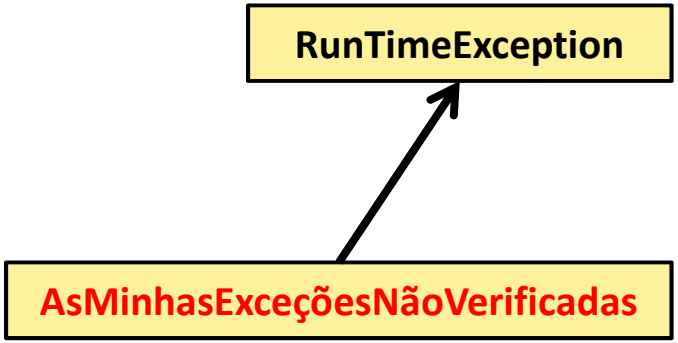
- As exceções não verificadas (unchecked exceptions)
  - Não são verificadas pelo compilador
  - Conduzem ao fim abrupto do programa se não forem apanhadas e tratadas
    - Esta é a prática normal
  - Um exemplo típico é a
    - **`IllegalArgumentException`**




```
public ContactDetails getDetails(String key){
    if(key == null) {
        throw new NullPointerException(
                "null key in getDetails");
    }
    if(key.trim().length() == 0) {
        throw new IllegalArgumentException(
                "Empty key passed to getDetails");
    }
    return book.get(key);
}
```

Exceções – Não Verificadas

- Exceções não verificadas (unchecked exceptions)
  - **`IllegalStateException`**



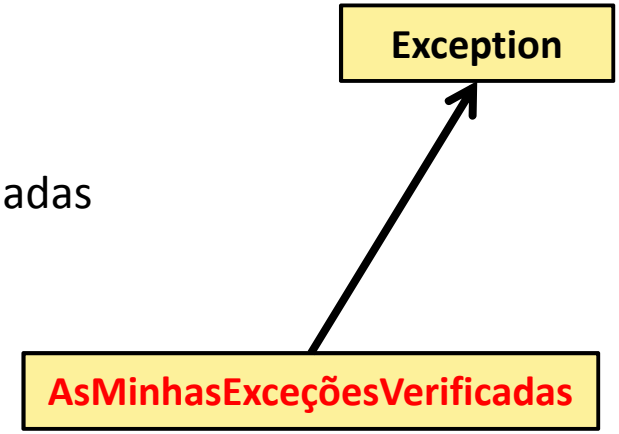

```
public ContactDetails(String name, String phone, String address){
    if(name == null) name = "";
    if(phone == null) phone = "";
    if(address == null) address = "";

    this.name = name.trim();
    this.phone = phone.trim();
    this.address = address.trim();

    if(this.name.length() == 0 && this.phone.length() == 0) {
        throw new IllegalStateException(
                "Either the name or phone must not be blank.");
    }
}
```

Exceções – Verificadas

- As exceções verificadas (checked)
  - Existem para serem apanhadas e tratadas
  - O compilador assegura que são estreitamente controladas
  - Tanto no servidor como no cliente
  - Usadas devidamente podem permitir a recuperação de situações de erro

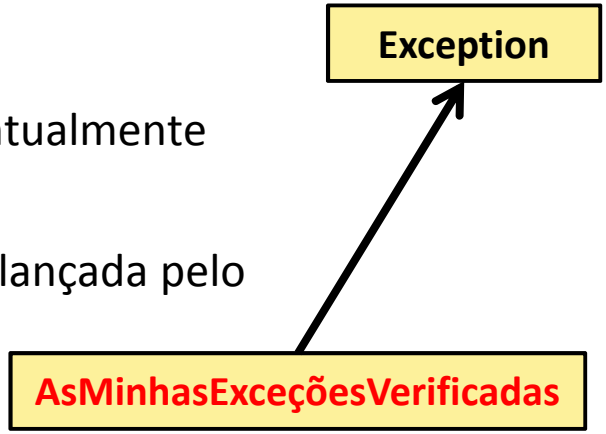


- A cláusula **`throws`**
  - Os métodos que lancem uma exceção verificada têm de incluir uma clausula **`throws`**:

```
public void saveToFile(String destinationFile) throws IOException {
    // […] ímplementação do método saveToFile
}
```

Exceções – blocos try/catch

- Blocos **`try – catch`**
  - A instrução **`try`** verifica a existência de exceções eventualmente lançadas pelo código do bloco entre { }.
  - A instrução **`catch`** captura a exceção eventualmente lançada pelo código do bloco **`try`**.




```
try {
    Proteja uma ou mais instruções aqui
}
catch (Exception e) {
    Reporte ou recupere da exceção aqui
}
```

```
try {
    addressbook.saveToFile(filename);
    tryAgain = false;
}
catch(IOException e) {
    System.out.println("Unable to save to " + filename);
    tryAgain = true;
}
```

1. A exceção é eventualmente lançada aqui

2. Se a exceção for lançada o controlo passa para aqui

Exceções – Apanhar várias exceções

- Blocos **`try`** com **multiplos `catch`**
  - Se o código de um bloco **`try`** pode gerar diferentes exceções ou
  - Se pretendemos dividi-las em subclasses mais especificas
  - Temos de capturar as diferentes exceções com blocos catch para cada tipo de exceção:

```
try {
    ...
    referencia.processo();
    ...
}
catch(EOFException e) {
    // Atuar para uma "end of file exception"
    ...
}
catch(FileNotFoundException e) {
    // Atuar para uma "file not found exception"
    ...
}
```

1. A exceção é eventualmente lançada aqui

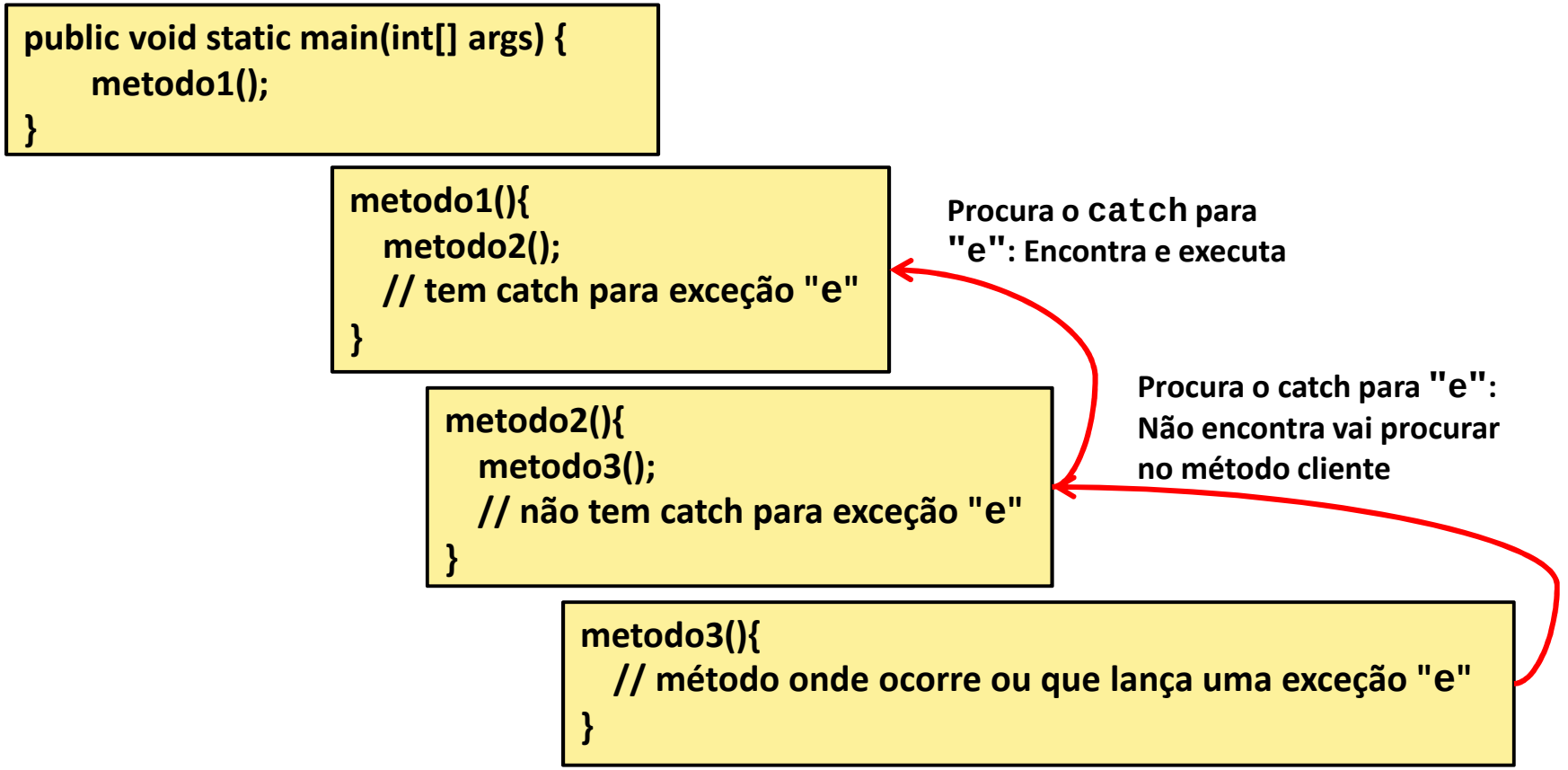
2. Dependendo da exceção que for lançada o controlo passa para o bloco catch correspondente

A partir da versão 7 do Java:

```
catch(EOFException|FileNotFoundException e) {
    //Atuar para uma "end of file exception" e "file not found exception"
    ...
```

Exceções – Procura Ascendente de um Catch

- Quando o método que lança a exceção não tem um **`catch`** para a exceção lançada, a procura do bloco **`catch`** adequado propaga-se ascendentemente pelos métodos servidores até encontrar um **`catch`** para essa exceção ou atingir o método **`main`** e terminar o programa.



Exceções – Procura Ascendente de um Catch

- Quando o método que lança a exceção não tem um **`catch`** para a exceção lançada, a procura do bloco **`catch`** adequado propaga-se ascendentemente pelos métodos servidores até encontrar um **`catch`** para essa exceção ou atingir o método **`main`** e terminar o programa.

  - Um "gestor de exceção" é considerado adequado quando a exceção que ele manipula é do mesmo tipo da exceção lançada.
  - Quando ele é encontrado, recebe o controlo do programa para que possa tratar a falha ocorrida.
    - Diz-se que o método "capturou" a excepção (*catch the exception*).
  - Se a excepção não for tratada e chegar à função main, o programa será interrompido com uma mensagem de erro.
  - Se nenhum dos métodos pesquisados pelo sistema de execução possui um gestor de exceções adequado o programa será interrompido com uma mensagem de erro.

Exceções – bloco finally

- Bloco **`finally`**
  - Destina-se a agrupar instruções que serão sempre executadas quer seja ou não gerada uma exceção.
  - As instruções do bloco **`finally`** são executadas mesmo que a saída dos blocos **`try/catch`** seja feita com **`return`**, **`continue`** ou **`break`**.
  - Uma exceção propagada ou não capturada continua a sair através do bloco **`finally`**

```
try {
    Proteja uma ou mais instruções aqui
}
catch (Exception e) {
    Reporte ou recupere da exceção aqui
}
finally {
    Execute aqui quaisquer ações comuns
    aos casos em que haja ou não lançamento
    de uma exceção.
}
```

O código do bloco `finally` é sempre executado

Exceções – Definição de Novas Exceções

- Para definir novas exceções:
  - Especializamos **`RuntimeException`** para criar uma nova exceção não verificada pelo compilador
  - Especializamos **`Exception`** para uma exceção verificada pelo compilador.

- A definição de novas exceções oferece maior precisão e melhor informação sobre o problema.
  - Inclua sempre a descrição da ocorrência.

```
public class NoMatchingDetailsException
                  extends Exception {

  private String key;

  public NoMatchingDetailsException
              (String message, String key){
    super(message);
    this.key = key;
  }

  public String getKey(){
    return key;
  }

  public String toString(){
    return "No details matching '" + key +
        "' were found.";
  }
}
```

Exceções – Recuperação de Erros

- Os clientes devem tratar das notificações de erro.
  - Verifique os valores retornados.
  - Não ignore as exceções.
- Inclua código para tentar recuperar do erro.
  - Frequentemente isso implica um ciclo.

```
// Tenta guardar a agenda telefónica
boolean sucedido = false;
int tentativas = 0;

do {
        try {
                addressbook.saveToFile(nomeFicheiro);
                sucedido = true;
        }
        catch(IOException e) {
                System.out.println("Incapaz de guardar o ficheiro " +
                        nomeFicheiro);
                tentativas++;
                if (tentativas < MAX_TENTATIVAS)
                        nomeFicheiro = nome alternativo;
        }
} while(!sucedido && tentativas < MAX_ATTEMPTS);

If (!sucedido) {
   Relate o problema e desista;
}
```

Exceções – Mais Comuns

- **`ArithmeticException`**
  Indica falhas no processamento aritmético, tal como uma divisão inteira por 0.
- **`ArrayIndexOutOfBoundsException`**
  indica a tentativa de acesso a um elemento de um array fora dos seus limites: ou o índice é negativo ou maior ou igual ao tamanho do array.
- **`IndexOutOfBoundsException`**
  Indica a tentativa de usar um índice fora dos limite de uma tabela.
- **`ArrayStoreException`**
  Indica a tentativa de armazenamento de um objecto inválido numa tabela.
- **`NegativeArraySizeException`**
  Indica a tentativa de criar uma tabela com dimensão negativa.
- **`StringIndexOutOfBoundsException`**
  Indica a tentativa de usar um índice numa string fora dos seus limites
- **`NumberFormatException`**
  indica a tentativa de conversão de uma string para um formato numérico, mas que o seu conteúdo não representava um número para aquele formato.
- **`NullPointerException`**
  indica que a instrução tentou usar null onde era necessária uma referência a um objeto
- **`IllegalArgumentException`**
  Quando o argumento do método tem um valor impossível
- **`IOException`**
  indica a ocorrência de qualquer tipo de falha em operações de entrada e saída.

Resumindo

- A linguagem Java permite o tratamento de situações de exceção de uma forma normalizada através da utilização de 5 palavras chave correspondentes a cláusulas especiais, a saber:
  - **throws**
  - **throw**
  - **try**
  - **catch**
  - **finally**

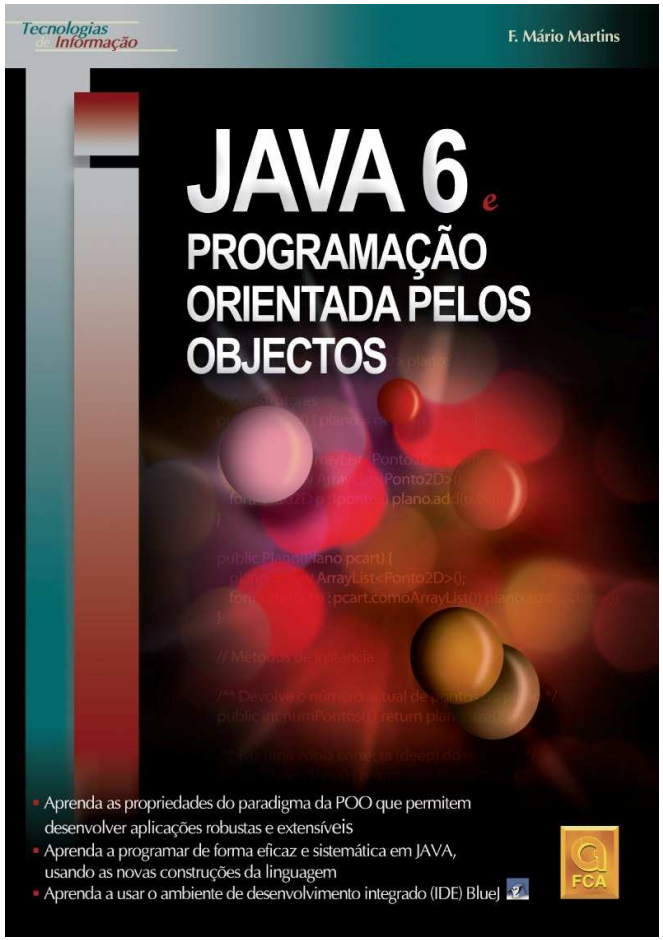
```
public void metodo()throws ExcecaoVerificadaHerdaDeException {

    // […] implementação do método1
    throw new ExcecaoVerificadaHerdaDeException(...);
    // […] mais implementação do método1

    try{
        // […] mais implementação do método1
    }
    catch (ArithmeticException e1){
        // Tratamento da e1 do tipo ArithmeticException
    }
    catch (RuntimeException e2){
        // Tratamento da e2 do tipo RuntimeException
    }
    finally{
        // Bloco opcional se existir é sempre executado
    }
}
```

Leitura Complementar

- Capítulo 9
  - Páginas 371 a 400



Exercicio Prático

1. Crie uma classe **`Data`** com os atributos dia, mês e ano.

```
public class Data {
  private int ano;
  private int mes;
  private int dia;

  public Data(int ano, int mes, int dia){
    this.ano = ano;
    this.mes = mes;
    this.dia = dia;
  }

  public int getAno() {
        return ano;
  }

  public void setAno(int ano) {
    this.ano = ano;
  }

  public int getMes() {
    return mes;
  }

  public void setMes(int mes){
    this.mes = mes;

  public int getDia() {
    return dia;
  }

  public void setDia (int dia)
    this.dia = dia;
  }

  @Override
  public String toString (){
    return ano + "/" + mes + "/" + dia;
  }
}
```

Exercicio Prático

2. Crie uma classe **`ValorInvalidoException`** do tipo **`Exception`**:
   - O construtor deve:
     - receber como argumentos o valor inválido e o nome da variável para a qual o valor foi mal inserido (Mês ou Dia). Ex.: **`ValorInvalidoException(mes, "mês")`**
     - Enviar uma mensagem no formato: **`O valor`** *`valor`* **`não é válido para o`** *`mês`***`.`**

```
public class ValorInvalidoException extends Exception {

   public ValorInvalidoException(int valor, String texto) {
      super("O valor "+ valor + " não é válido para o " + texto + ".");
   }
}
```

Exercicio Prático

3. Altere o código dos modificadores da classe **`Data`** por forma a criarem e lançarem uma **`ValorInvalidoException`** se o argumento passado for inválido.

```
public class Data {
  // atributos
  // construtores
  // acessores

  public void setDia (int dia) throws ValorInvalidoException {
    if ((dia>=1) && (dia <=31))
      this.dia = dia;
    else
      throw new ValorInvalidoException(dia, "dia");
 }

  public void setMes(int mes) throws ValorInvalidoException {
    if ((mes>=1) && (mes <=12))
      this.mes = mes;
    else
      throw new ValorInvalidoException(mes, "mes");
  }
}
```

Exercicio Prático

4. No método **`main`** crie um objeto da classe **`Data (2012,12,31)`** e tente alterar o dia para **`32`** ou o mês para **`14`**.

```
public class Main {

    public static void main(String[] args){
        Data dataNascimento = new Data (2012, 12, 31);
        try{
            dataNascimento.setDia(32);
            //dataNascimento.setMes(14);
        }
        catch (ValorInvalidoException e){
            System.out.println(e.getMessage());
        }
    }

}
```