Tópicos selecionados de programação em Java

Revisado para Java 7

Exceções em Java

Padrões, anti-padrões e boas práticas

Helder da Rocha

2005, 2011

Objetivos

- Este módulo é uma revisão da teoria de exceções na linguagem Java através da análise de práticas comuns (padrões e anti-padrões) de sua aplicação
- Estrutura da apresentação
  - Lista de recomendações (a maioria extraída do livro Effective Java, de Joshua Bloch e Practical Java de P. Haggar)
- Espaço para discussão
  - Nem todas as recomendações se aplicam a qualquer situação
  - Há casos em que uma boa prática pode ser abandonada em nome de outros benefícios
  - Qualquer regra podem ser quebrada, mas, é preciso antes conhecê-las para poder justificar!

Tópicos

1. Conheça a mecânica do fluxo de controle de exceções
2. Nunca ignore ou oculte uma exceção
3. Use finally para evitar perdas de recursos
4. Seja específico e abrangente com a cláusula throws
5. Use exceções somente para condições excepcionais
6. Use exceções checadas para condições recuperáveis e de runtime para erros
7. Evite o uso desnecessário de checked exceptions
8. Favoreça o uso de exceções padrão
9. Documente todas as exceções lançadas por cada método
10. Coloque blocos try-catch fora de loops
11. Não use exceções para condições de erro
12. Gere exceções a partir de construtores
13. Retorne objetos para um estado válido antes de gerar uma exceção
14. Use chained exceptions
15. Teste a ocorrência de exceções

EJ = Effective Java, BJ = Bitter Java, PJ = Practical Java – veja referências no final

O que são exceções

- Exceções são
  - Erros de tempo de execução
  - Objetos criados a partir de classes especiais que são "lançados" quando ocorrem condições excepcionais
- Métodos podem capturar ou deixar passar exceções que ocorrerem em seu corpo
  - É obrigatório, para a maior parte das exceções, que o método declare quaisquer exceções que ele não capturar
- Mecanismo try-catch é usado para tentar capturar exceções enquanto elas passam por métodos

Por que ocorrem exceções?

- Segundo a **JLS**, há três motivos para que ocorra uma exceção
  1. Foi detectada uma condição anormal de execução (síncrona)
  2. Uma exceção assíncrona ocorreu
  3. Uma instrução **throw** foi executada

1. Causas de exceções **síncronas** (ocorrem em lugar determinado)
   - Expressão que viola a semântica da linguagem (ex: divisão por zero)
   - Erro que ocorre ao carregar ou ligar parte do programa (Classloader)
   - Limitação de recursos, como memória
2. Causas de exceções **assíncronas** (são raras)
   - Chamada do método Thread.stop(), que foi descontinuado.
   - Erro interno na JVM
3. Instruções **throw** são explicitamente chamadas por programadores (geralmente de APIs) para reportar uma condição exceptional em relação ao domínio da aplicação

JLS = Java Language Specification

O que diz a especificação?

- No capítulo 11, sobre Exceções:
  - “Durante o processo de lançamento de uma exceção, a JVM abruptamente conclui quaisquer expressões, declarações, chamadas de métodos ou construtores, e expressões de inicialização iniciados mas não terminados no thread atual.
  - O processo continua até que seja encontrado um handler que indique ser capaz de lidar com aquela exceção em particular, ao fornecer o nome da classe da exceção ou de sua superclasse.
  - Se tal handler não for encontrado, o método **uncaughtException** é chamado para o ThreadGroup que é pai do thread atual.”
- Além disso, em relação à sincronização...
  - “O mecanismo de exceções da plataforma Java é integrado com seu modelo de sincronização, de forma que travas são liberadas à medida em que declarações sincronizadas e chamadas de métodos sincronizados terminam abruptamente.”

Três tipos de erros de tempo de execução

- 1. Erros de lógica de programação
  - Ex: limites do vetor ultrapassados, divisão por zero
  - Devem ser corrigidos pelo programador
- 2. Erros devido a condições do ambiente de execução
  - Ex: arquivo não encontrado, rede fora do ar, etc.
  - Fogem do controle do programador mas podem ser contornados em tempo de execução
- 3. Erros graves onde não adianta tentar recuperação
  - Ex: falta de memória, erro interno do JVM
  - Fogem do controle do programador e não podem ser contornados

Como causar uma exceção?

- Uma exceção é um tipo de objeto que sinaliza que uma condição excepcional ocorreu
  - A identificação (nome da classe) é sua parte mais importante
- Precisa ser criada com **new** e depois lançada com **throw**
  - ```
    IllegalArgumentException e = new
            IllegalArgumentException("Erro!");
    throw e;  // exceção foi lançada!
    ```
- A referência é desnecessária. A sintaxe abaixo é mais usual:
  - ```
    throw new IllegalArgumentException("Erro!");
    ```

1. Conheça a mecânica do fluxo de controle de exceções (EJ Praxis 16)

- Uma exceção lançada interrompe o fluxo normal do programa
  - O fluxo do programa segue a exceção
  - Se o método onde ela ocorrer não a capturar, ela será propagada para o método que chamar esse método e assim por diante
  - Se ninguém capturar a exceção, ela irá causar o término da aplicação
  - Se em algum lugar ela for capturada, o controle pode ser recuperado

Captura e declaração de exceções

```
public class RelatorioFinanceiro {
    public void metodoMau() throws ExcecaoContabil {
        if (!dadosCorretos) {
            throw new ExcecaoContabil("Dados Incorretos");
        }
    }
    public void metodoBom() {
        try {
            ... instruções ...
            metodoMau();
            ... instruções ...
        } catch (ExcecaoContabil ex) {
            System.out.println("Erro: " + ex.getMessage());
        }
        ... instruções ...
    }
}
```

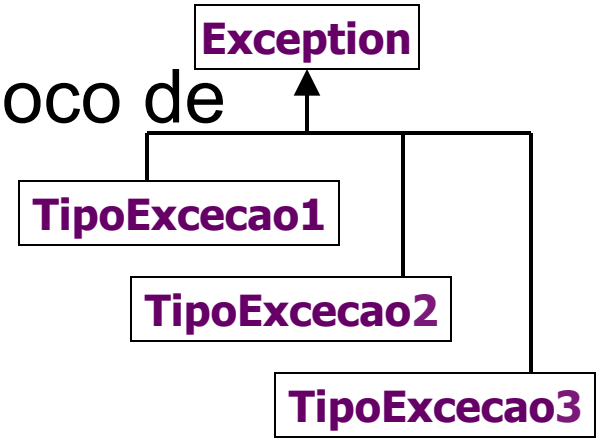
instruções que sempre serão executadas

instruções serão executadas se exceção não ocorrer

instruções serão executadas se exceção não ocorrer ou se ocorrer e for capturada

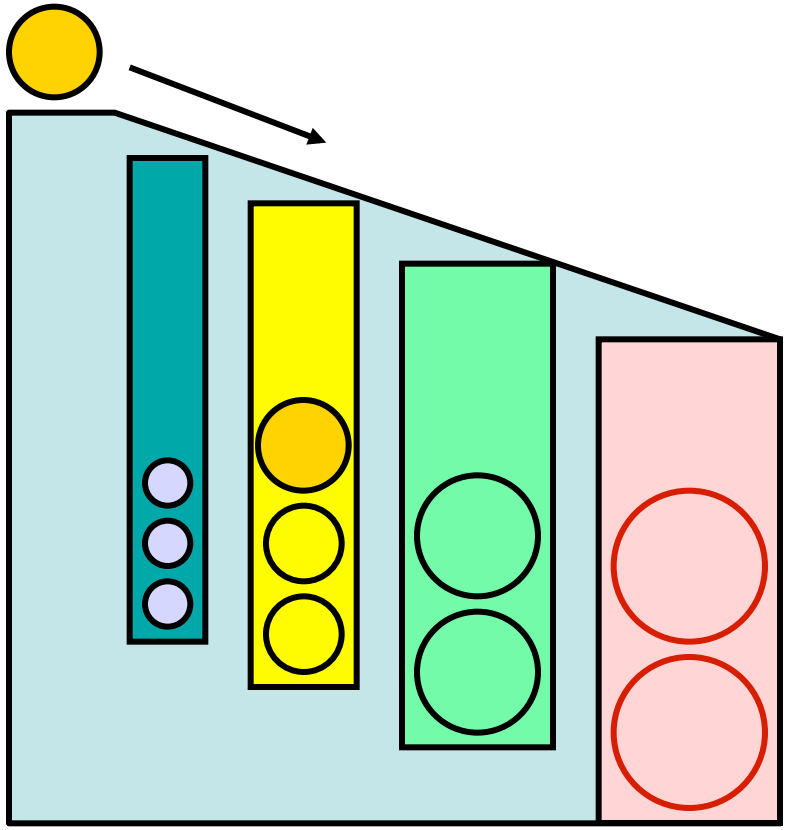
try e catch

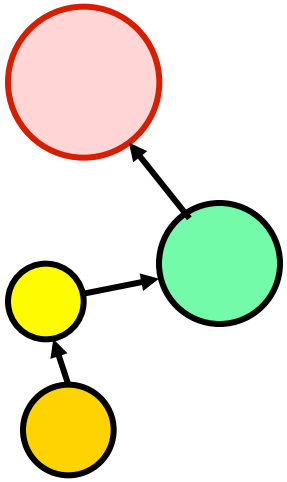
- O bloco try "tenta" executar um bloco de código que pode causar exceção
- Blocos catch recebem tipo de exceção como argumento
  - Se ocorrer uma exceção no try, ela irá descer pelos catch até encontrar um que declare capturar exceções de uma classe ou superclasse da exceção
  - Apenas um dos blocos catch é executado




```
try {
    ... instruções ...
} catch (TipoExcecao1 ex) {
    ... faz alguma coisa ...
} catch (TipoExcecao2 ex) {
    ... faz alguma coisa ...
} catch (Exception ex) {
    ... faz alguma coisa ...
}

... continuação ...
```

try-catch





finally

- O bloco finally contém instruções que devem se executadas independentemente da ocorrência ou não de exceções

```
try {
  // instruções: executa até linha onde ocorrer exceção
} catch (TipoExcecao1 ex) {
   // executa somente se ocorrer TipoExcecao1

} catch (TipoExcecao2 ex) {
   // executa somente se ocorrer TipoExcecao2
} finally {
   // executa sempre ...
}

// executa se exceção for capturada ou se não ocorrer
```

Java 7

- O Java 7 amplia a sintaxe do try-catch suportando:
  - Fechamento automático de recursos declarados e inicializados dentro do bloco try: close() é chamado automaticamente

    ```
    try (FileInputStream out = new FileInputStream()) {...}
    ```

    Mais exemplos veja: *www.theserverside.com/tutorial/OCPJP-Exam-Objective-How-to-Use-Try-With-Resources-to-Clean-Up-Closeable-Resources*
  - Lista de exceções em um único catch (em vez de vários catch)

    ```
    catch(Excecao1 | Excecao2) { ... }
    ```

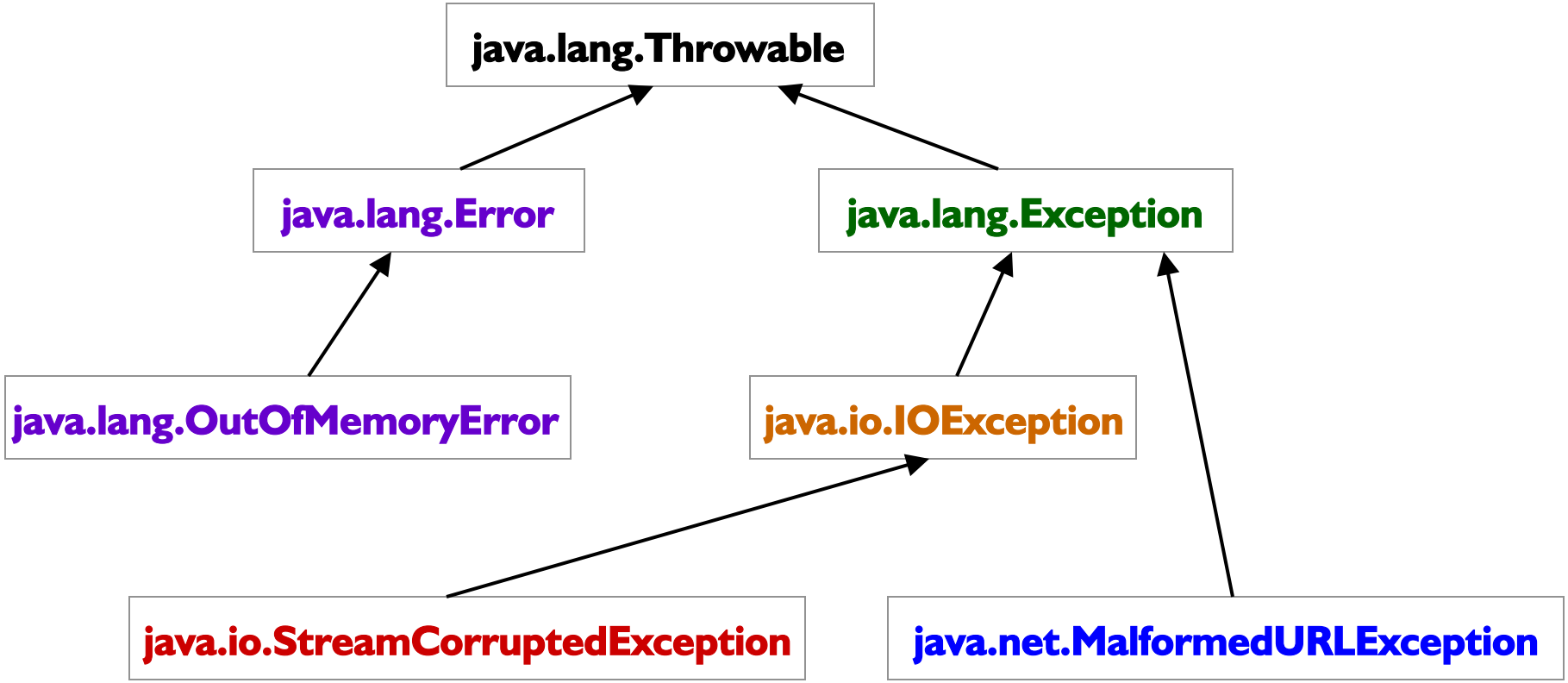
    Mais exemplos veja: *docs.oracle.com/javase/7/docs/technotes/guides/language/catch-multiple.html*

Teste se voce sabe como funcionam as exceções

- Considere o seguinte código ...

```
1.  try {
2.    URL u = new URL(s); // s is a previously defined String
3.    Object o = in.readObject(); // in is valid ObjectInputStream
4.    System.out.println("Success");
5.  }
6.  catch (MalformedURLException e) {
7.    System.out.println("Bad URL");
8.  }
9.  catch (IOException e) {
10.   System.out.println("Bad file contents");
11. }
12. catch (Exception e) {
13.   System.out.println("General exception");
14. }
15. finally {
16.   System.out.println("doing finally part");
17. }
18. System.out.println("Carrying on");
```

... e a seguinte hierarquia



Teste 1

- 1. Que linhas são impressas se os métodos das linhas 2 e 3 completarem com sucesso sem provocar exceções?
  - A. Success
  - B. Bad URL
  - C. Bad File Contents
  - D. General Exception
  - E. Doing finally part
  - F. Carrying on

Teste 2

- 2. Que linhas são impressas se o método da linha 3 provocar um OutOfMemoryError?
  - A. Success
  - B. Bad URL
  - C. Bad File Contents
  - D. General Exception
  - E. Doing finally part
  - F. Carrying on

Teste 3

- 3. Que linhas são impressas se o método da linha 2 provocar uma MalformedURLException?
  - A. Success
  - B. Bad URL
  - C. Bad File Contents
  - D. General Exception
  - E. Doing finally part
  - F. Carrying on

Teste 4

- 4. Que linhas são impressas se o método da linha 3 provocar um StreamCorruptedException?
  - A. Success
  - B. Bad URL
  - C. Bad File Contents
  - D. General Exception
  - E. Doing finally part
  - F. Carrying on

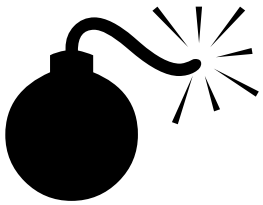
2. Nunca ignore ou oculte uma exceção (EJ Item 47, PJ Praxis 17, 18)

- Quando uma exceção é lançada, pode-se
  1. Capturar a exceção e tratá-la
  2. Capturar a exceção e relançá-la
  3. Capturar a exceção e lançar outra
  4. Não capturá-la e deixar que ela seja propagada
- O método que contiver a exceção terá que conter uma declaração throws em todos os casos exceto o primeiro
- A pior coisa que pode ser feita é
  5. Capturar a exceção e ignorá-la

Ignorando o problema

- A causa da queda do império romano

```
try {
      // risco de InstabilidadePoliticaException
      // risco de InsurreicaoDeBarbarosException
      // risco de SegurancaException
} catch (Exception e) {}
// tudo continua em paz na casa de Cesar
```



Se não souber o que fazer...

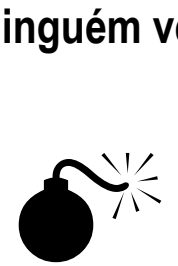
- ... pelo menos documente o risco

```
try {
      // risco de InstabilidadePoliticaException
      // risco de InsurreicaoDeBarbarosException
      // risco de SegurancaException
} catch (Exception e) {
         System.out.println(e + " ocorreu!");
         log.info(e);
}
// tudo continua aqui
```

Ocultando o problema

- Uma exceção é ocultada quando uma outra exceção é lançada de um bloco **catch** ou **finally**, depois que é lançada do bloco principal
- Anti-pattern

```
try {
     throw new Exception("exception 1");
}
catch (Exception e) {
     throw new Exception("exception 2");
}
finally {
     throw new Exception("exception 3");
}
```



Parece pouco realista o exemplo?

Exemplos realistas

```
public void readFile() throws FileNotFoundException, IOException {
    BufferedReader br1, br2;
    FileReader fr;
    try {
        fr = new FileReader("data1.txt"); // FileNotFoundException
        br1 = new BufferedReader(fr);
        int i = br1.read(); // risco de IOException
        fr = new FileReader("data2.txt"); // FileNotFoundException
        br2 = new BufferedReader(fr);
        int j = br2.read(); // risco de IOException
    }
    finally {
        if (br1 != null)
            br1.close(); // IOException
        if (br2 != null)
            br2.close(); // IOException
    }
}
```

3. Use finally para evitar perdas de recursos (PJ Praxis 21)

- O bloco finally executa **sempre**
  - É ideal para manter o estado de um objeto consistente, mesmo depois de exceções
  - Sem finally, não há garantia que seu código vai realmente executar
- **Anti-pattern**: liberação de recursos duvidosa

```
Socket s = new Socket("sol4", 2783);
BufferedReader r = new BufferedReader(
    new InputStreamReader(s.getInputStream()));
try {
     String text = r.readline();
}
catch (IOException e) {
    s.close();
    throw e;
}
s.close();
```

Pode nunca ocorrer

Pode nunca ocorrer

Usando finally

- Não só fica mais seguro como fica mais compacto, claro e fácil de entender
  - Não existe como sair do bloco try sem executar o bloco finally: quando o bloco finally está presente ele sempre executa

```
Socket s = new Socket("sol4", 2783);
BufferedReader r = new BufferedReader(
    new InputStreamReader(s.getInputStream()));

try {
    String text = r.readline();
}
finally {
   s.close();
}
```

Abusando de finally

- O que o programa abaixo vai imprimir?

```
public int m1() {
    try {
        return 2;
    } catch (Exception e) {
        return 3;
    }
}
public int m2() {
    try {
        return 3;
    } finally {
        return 4;
    }
}
public static void main(String[] args) {
    FinallyTest ft = new FinallyTest();
    System.out.println("m1 " + ft.m1());
    System.out.println("m2 " + ft.m2());
}
```

4. Seja específico e abrangente com a cláusula throws

- A cláusula throws serve como documentação
  - Através dela o programador cliente saberá quais exceção terá que tratar
  - A documentação JavaDoc extrai informação da cláusula throws
- **Anti-pattern**: outra forma de ocultar exceções

```
class Exception1 extends Exception {}
class Exception2 extends Exception1 {}
class Exception3 extends Exception2 {}

class Lazy {
    public void foo(int i) throws Exception1 {
        if (i == 1)
            throw new Exception1();
        else if (i == 2)
            throw new Exception2();
        else if (i == 3)
            throw new Exception3();
    }
}
```

É perfeitamente legal, porém tem muito pouca informação!

Como usuário vai ficar sabendo dessas outras exceções?

O problema e a solução

- O problema

```
try {
     Lazy l = new Lazy();
     l.foo(3);
 }
 catch (Exception1 e) {
      // A exceção é Exception3,
      // mas nao tenho como tratar!
 }
```

- A solução é não ter preguiça e declarar todas as exceções que realmente ocorrem no código

```
public void foo(int I)
       throws Exception1, Exception2, Exception3
```

5. Use exceções somente para condições excepcionais (EJ Item 39, PJ Praxis 24)

- **Anti-pattern**: usar

```
// TERRÍVEL!
try {
    int i = 0;
    while(true)
        a[i++].f();
} catch (ArrayIndexOutOfBoundsException e) {}
```

em vez de

```
for (int i = 0; i < a.length; i++)
        a[i].f();
```

com a intenção (equivocada) de melhorar a performance

- Há muitas coisas erradas com esse código
  - É muito, muito mais ineficiente (dezenas de vezes)
  - É pouco legível: é mais difícil saber o que ele faz
  - Pode falhar, pois pode capturar um bug que tenha provocado a exceção em outra parte do código (no método f(), por exemplo)

Exceções nunca devem ser usadas para controlar fluxo de controle!

Implicações no design de APIs

- Uma API bem projetada não deve forçar seu cliente a usar exceções para fluxo normal de controle
  - Classe com um método dependente de estado deve ter outro método somente para teste de estado
- Exemplo: Iterator
  - Possui o método next() , que depende de estado
  - Possui o método hasNext() que testa o estado
  - Isto permite que se use o procedimento padrão

```
for (Iterator i = collection.iterator();
i.hasNext();)
     Foo foo = (Foo)i.next();
```

  - O código abaixo, seria necessário se não existisse o hasNext()

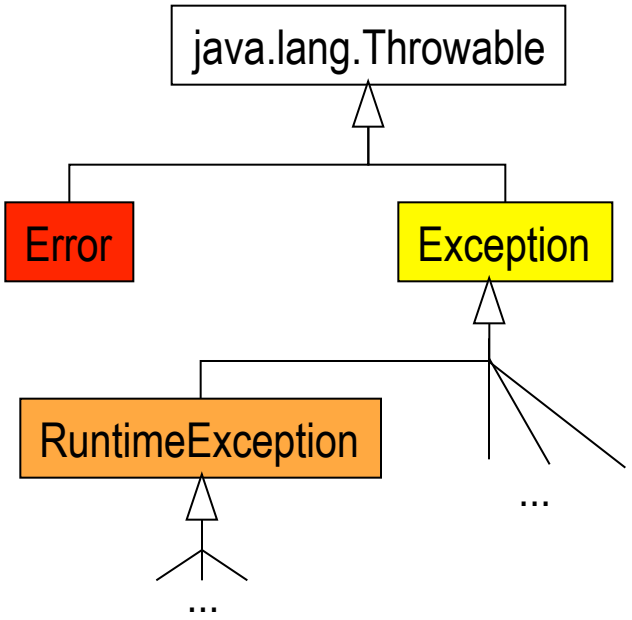
```
try { // Nao use!
     Iterator i = collection.iterator();
     while(true)
          Foo foo = (Foo) i.next();
} catch (NoSuchElementException e) {}
```

Quando o anti-pattern é aceitável

- Em alguns (poucos) casos é aceitável ter o método dependente de estado retornar um valor distinto, como null
  - Isto não vale para o Iterator, pois null é um valor de retorno legítimo para ele
- Razões para fazer isto
  - Acesso concorrente sem sincronização externa (o estado do objeto poderia mudar entre a chamada do método que testa o estado e o método dependente de estado)
  - Performance, se teste de estado for complexo e duplicar o trabalho do método dependente

6. Use corretamente exceções checadas e exceções de runtime (EJ 40)

- A linguagem Java fornece três tipos de objetos Throwable
  - Exceções **checadas** (na compilação): descendentes de **Exception**
  - Exceções de **run-time** (não checadas): descendentes de **RuntimeException**
  - **Erros** (não checados): descendentes da classe **Error**



- Regras gerais
  - Não use erros; nunca crie erros (deixe-os para o sistema)
  - Use exceções de runtime para indicar erros de programação (bugs, violação de contrato da API);
  - Use exceções checadas para condições das quais espere-se que o cliente possa recuperar-se
  - Nunca use Throwable (use suas subclasses)

Guarde informação nas exceções

- Autores de API freqüentemente esquecem que exceções são objetos normais, que podem ter métodos e guardar estado
  - Tais métodos podem guardar informação adicional sobre a causa da exceção que irá facilitar a recuperação
  - Na ausência desses métodos, programadores acabam tentando interpretar a representação do String de uma exceção para descobrir informação adicional
- **Boa prática**: crie métodos para guardar estados importantes utilizáveis pelos clientes que capturarem suas exceções

```
class HttpException extends Exception {
    private String statusCode;
    public HttpException(String m, Throwable e, String status) {
        super(m, e);
        this.statusCode = status;
    }
    public String getStatusCode() {
        return statusCode;
    }
}
```

Código é passado quando exceção for provocada

Código cliente pode obter o código e decidir, com base no código, se tenta de novo ou não

7. Evite o uso desnecessário de checked exceptions (EJ 41, PJ Praxis 29)

- O **lado bom** das exceções checadas é que forçam o programador a lidar com condições excepcionais
- O **lado ruim** é que o seu uso excessivo pode fazer uma API bem menos agradável de usar*
  - O programador tem que ficar capturando ou declarando exceções diversas em seus métodos
- O fardo é justificável se
  - A condição excepcional não pode ser prevenida pelo uso correto da API, **e**
  - O programador que usa a API pode tomar uma atitude útil uma vez que esteja diante da exceção
- A menos que **ambas** as condições sejam verdadeiras, é mais apropriado usar uma exceção não checada

* PJ Praxis 29: considere a desvantagem da cláusula throws

Um teste para ajudar na decisão

- Pergunte-se, numa situação de **catch**: isto é o melhor que eu posso fazer quando a exceção acontece?

```
} catch (TheCheckedException e) {
    assert false : "Nunca deve acontecer!";
}
```

- Ou isto?

```
} catch (TheCheckedException e) {
    e.printStackTrace();  // desisto!
    System.exit(1);
}
```

- Se o programador que usa a API não pode fazer nada melhor, uma exceção não checada seria mais apropriada
- Um exemplo na API Java de uma exceção que não passa esse teste é **CloneNotSupportedException**
  - A natureza obrigatória (checada) da exceção não traz nenhum benefício para o programador, mas requer esforço e complica o código

8. Favoreça o uso das exceções (não checadas) padrão (EJ 42)

- Antes de criar uma nova exceção, veja se já não existe uma exceção padrão que resolve seu problema
- O reuso de exceções pré-existentes tem vários benefícios
  - Torna sua API mais fácil de aprender e usar
  - Usa menos memória (carrega-se menos classes)
- Todas as chamadas incorretas podem ser reduzidas a **argumento** ou **estado** ilegal. Para essas situações genéricas há duas exceções

1. `IllegalArgumentException`
   - Talvez a mais freqüentemente usada
   - Use para acusar valor inadequado para parâmetro de um método ou construtor (ex: valor negativo para um contador)
2. `IllegalStateException`
   - Quando uma chamada é ilegal, dado o estado do objeto
   - Por exemplo, ao chamar um objeto ainda não inicializado

Exceções freqüentemente usadas (2)

- Outras exceções (mais específicas) são usadas para certos tipos de estado ou argumento ilegais

  3. `NullPointerException`

  – Se o cliente passa null quando valores nulos são proibidos

  4. `IndexOutOfBoundsException`

  – Se o cliente passa um valor fora de escopo

- Outras exceções importantes

  5. `ConcurrentModificationException`

  – Quando objeto não treadsafe detecta alteração concorrente

  6. `UnsupportedException`

  – Quando uma implementação não implementa parte da interface

Exceções de APIs padrão

- Exceções checadas de APIs padrão também devem ser reusadas sempre que possível
  - IOException, SQLException, RemoteException
  - Analise sua condição excepcional e avalie se ela é um tipo de alguma exceção já existente
- Benefícios do reuso através de especialização
  - Diminui o número de exceções que o cliente precisa capturar (recomendação EJ41) permitindo que o cliente capture a exceção através de uma superclasse padrão conhecida (ex: FileNotFoundException)
  - Permite que nova exceção herde comportamentos interessantes
  - Facilita o uso da API em sistemas existentes

9. Documente todas as exceções lançadas por cada método (EJ 44)

- A descrição das exceções causadas por um método é parte importante da documentação de uma API
  - É extremamente importante que sejam cuidadosamente documentadas
- Forma básica de documentação
  - Anotação @throws do Javadoc, antes de cada método

```
/** Abre um arquivo.
 *  @throws FileNotFoundException se o
 *          arquivo não for encontrado no
 *          ClassPath da aplicação.
 *  @throws NumberFormatException se o código
 *          contido no arquivo não for um inteiro.
*/
void open(File arquivo)
   throws FileNotFoundException{ ... }
```

Documentação: recomendações básicas (1)

- Sempre declare exceções checadas individualmente
  - Documente precisamente as condições sob as quais cada uma é provocada usando a cláusula **@throws**
  - Seja específico: nunca declare **@throws Exception** quando a exceção é **FileNotFoundException**
  - Várias ferramentas de geração de código geram os comentários corretos (outras geram os comentários genéricos!)
  - Acrescente detalhes sobre as condições que causam a exceção

Documentação: recomendações básicas (2)

- Use a cláusula **@throws** para documentar cada exceção não checada que um método pode provocar, mas não use a palavra-chave **throws** para incluir exceções não checadas na declaração do método
  - Exceções não checadas (runtime exceptions) não devem ser declaradas em Java com **throws**
  - Declarando a exceção não checada na documentação, permite que o programador saiba que seu método causa certas exceções e saberá que são exceções não checadas
- Decida qual o detalhamento ideal
  - Não é realista declarar todas as exceções não checadas que um método causa (principalmente quando vêm de outros métodos)
  - Declare exceções que são causadas em vários métodos pelos mesmos motivos no comentário de classe.

10. Coloque blocos try-catch fora de loops (PJ Praxis 23)

- Gargalo de performance!

```
public void method1(int size) {
    int[] ia = new int[size];
    for (int i = 0; i < size; i++) {
        try {
            ia[i] = i;
        } catch (Exception e) {
        //Exception ignore
        }
    }
}
```

Pelo menos 20 vezes mais lento (JDK 1.2)

Muito mais código executado dentro do loop

```
public void method2(int size) {
    int[] ia = new int[size];
    try {
        for (int i = 0; i < size; i++) {
            ia[i] = i;
        }
    } catch (Exception e) {
    //Exception ignored on purpose
    }
}
```

11. Não use exceções para qualquer condição de erro (PJ Praxis 25)

- Há situações limites onde o controle de fluxo confunde-se com o tratamento de exceções
  - Nesses casos é preferível usar controle de fluxo
  - Avalie cada situação e o benefício que se terá em usar ou não exceções (reuso, propagação, etc.)

Pior

```
int data;
MyInputStream in = new
  MyInputStream("filename.ext");
while (true) {
    try {
        data = in.getData();
    } catch (NoMoreDataException e) {
        break;
    }
}
```

```
int data;
MyInputStream in = new
  MyInputStream("filename.ext");
while (data != 0) {
    try {
        data = in.getData();
    } catch (NoMoreDataException e) {
        break;
    }
}
```

Melhor

12. Gere exceções a partir de construtores (PJ Praxis 26)

- Construtores não retornam valor
  - Tratamento de erros tradicional (pré-Java) buscava alternativas como: realizar a construção em dois estágios ou usar um flag interno para sinalizar erro
  - Em Java, construtores devem provocar exceções
  - Geralmente são mais simples pois se falham, não precisam se preocupar com o estado do objeto
- Algumas regras para melhor coesão
  - Trate as exceções que possam ser tratadas localmente sempre que possível
  - Propague as exceções que possam melhor ser tratadas nas classes clientes (se a criação do objeto falhar, o cliente pode usar outra alternativa)

13. Retorne objetos para um estado válido antes de gerar uma exceção

(EJ Item 46, PJ Praxis 27)

- Uma chamada de método malsucedida deve deixar o objeto no estado em que estava antes da chamada
- Há várias estratégias para alcançar este efeito
  1. Desenvolver **objetos imutáveis**: se o objeto é imutável, não há como ele ficar em um estado inconsistente
  2. Em objetos mutáveis, validar parâmetros **antes** da operação para que a exceção ocorra antes de qualquer alteração no estado do objeto, ou ordenar as operações para que partes que falhem fiquem **antes** das partes que mudam o estado
  3. Rollback: escrever código para **recuperar o estado anterior**
  4. Backup: realizar a operação em uma **cópia temporária** do objeto e fazer a substituição quando concluir

Estratégia 1: Objetos imutáveis

- Para tornar uma classe imutável, é preciso seguir as seguintes regras
  - Não ter métodos que permitam modificar o objeto
  - Garantir que nenhum método possa ser sobreposto (marcar como final ou private)
  - Tornar todos os atributos finais
  - Tornar todos os atributos private
  - Garantir acesso exclusivo a quaisquer componentes mutáveis (referências)
- Veja mais sobre objetos imutáveis no excelente Item 13, do livro Effective Java

Estratégia 2: ordenação (2)

- A classe abaixo é mutável e uma exceção irá deixá-la em estado inconsistente

```
class Foo {
    private int numElements;
    private MyList myList;

    public void add(Object o) throws SomeException {
        numElements++;  //estado alterado!
        if (myList.maxElements() < numElements) {
            //Reallocate myList
            //Copy elements as necessary
            //Could throw exceptions
        }
        myList.addToList(o); //Could throw exception
    }
}
```

Estratégia 2: ordenação

- A solução é deixar as operações que alteram o objeto para o final

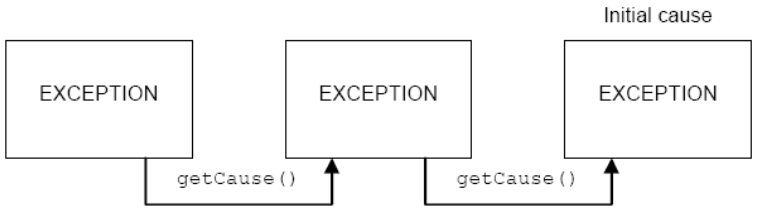
```
class Foo {
    private int numElements;
    private MyList myList;

    public void add(Object o) throws SomeException {
        if (myList.maxElements() < numElements) {
            //Reallocate myList
            //Copy elements as necessary
            //Could throw exceptions
        }
        myList.addToList(o); //Could throw exception
        numElements++;  //estado alterado!

    }
}
```

14. Implemente corretamente suas exceções

- Crie construtores para suas exceções que chamem os construtores da sua superclasse

```
class NovaExcecao extends Exception {
    public NovaExcecao () {}
    public NovaExcecao (String mensagem) {
        super(mensagem);
    }
}
```

- A partir do Java 1.4 é possível encadear exceções (recuperáveis via getCause())



  - Crie construtores que recebam o argumento Throwable
  - Conheça a API

Principais métodos

- Construtores de Exception
  - Exception ()
  - Exception (String message)
  - Exception (String message, Throwable cause)
- Métodos de Exception
  - String getMessage()
    - Retorna mensagem passada pelo construtor
  - Throwable getCause()
    - Retorna exceção que causou esta exceção
  - String toString()
    - Retorna nome da exceção e mensagem
  - void printStackTrace()
    - Imprime detalhes (stack trace) sobre exceção

15. Teste a ocorrência de exceções!

- Escrever testes de unidade é uma prática recomendada
  - Freqüentemente testa-se tudo que pode falhar, menos a falha em si, e quando ela ocorre, pode não causar exceção esperada
- Usando o JUnit* framework
  - Método fail() causa falha do teste
  - É chamado apenas se exceção não for chamada quando devia

```
public void testEntityNotFoundException() {
    resetEntityTable(); // no entities to resolve!
    try {
        // Following method call must cause exception!
        ParameterEntityTag tag = parser.resolveEntity("bogus");
        fail("Should have caused EntityNotFoundException!");
    } catch (EntityNotFoundException e) {
     // success: exception occurred as expected
    }
}
```

* www.junit.org

Fontes de referência

[JLS] James Gosling, Bill Joy, Guy Steele, Java Language Specification second edition, Addison Wesley, 2001

- Capítulo 11
- Seção 14.18 (throws),
- Seção 14.20 (try-catch)

[SDK] Documentação do J2SDK 5.0

[EJ] Joshua Bloch, Effective Java Programming Guide, Addison-Wesley, 2001

- Vários padrões foram extraídos deste livro; veja o item correspondente em cada slide

[PJ] P. Haggar, Practical Java Language Programming Guide, Addison-Wesley, 1999

- Vários padrões foram extraídos deste livro; veja o item (praxis) correspondente em cada slide

[BJ] Bruce Tate, Bitter Java, Manning, 2002

- Vários anti-patterns com refactorings que os consertam, destacando situações onde as regras podem (ou devem) ser quebradas

Exceções em Java

Padrões, anti-padrões e boas práticas

Criado em 31 de julho de 2005
Atualizado em 12 de setembro de 2011