Java - Exceções

BCC 221 - Programação Orientada a Objectos(POO)

Guillermo Cámara-Chávez

Departamento de Computação - UFOP

Tratamento de Exceções

- Uma exceção é uma **indicação de um problema** que ocorre **durante a execução** de um programa
  - **Tratar as exceções permite** que um **programa continue executando** como se não houvesse ocorrido um erro;
  - Programas **robustos e tolerantes a falhas**
- O estilo e os detalhes do tratamento de exceções em Java é baseado parcialmente do encontrado em C++.

Tratamento de Exceções (cont.)

- O exemplo a seguir apresenta um problema comum
  - Divisão por zero.
- Exceções são disparadas, e o programa é incapaz de tratá-las.

Tratamento de Exceções (cont.)

```
import java.util.Scanner;

public class DivideByZeroNoExceptionHandling
{
   //demonstra o disparo de uma exceção quando ocorre uma divisão por zero
   public static int quotient( int numerator, int denominator )
   {
      // possível divisão por zero
      return numerator / denominator;
   }
```

Tratamento de Exceções (cont.)

```
    public static void main( String args[] )
    {
       Scanner scanner = new Scanner( System.in );

       System.out.print( "Please enter an integer numerator: " );
       int numerator = scanner.nextInt();
       System.out.print( "Please enter an integer denominator: " );
       int denominator = scanner.nextInt();

       int result = quotient( numerator, denominator );
       System.out.printf("\nResult: %d / %d = %d\n", numerator,
           denominator, result);
    }
}
```

Tratamento de Exceções (cont.)

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
at DivideByZeroNoExceptionHandling.quotient(
DivideByZeroNoExceptionHandling.java:10)
at DivideByZeroNoExceptionHandling.main(
DivideByZeroNoExceptionHandling.java:22)
```

Tratamento de Exceções (cont.)

- Quando o denominador é nulo, várias linhas de informação são exibidas em resposta à entrada inválida
  - Esta informação é chamada de **Stack Trace**;
  - Inclui o nome da exceção em uma mensagem descritiva que indica o problema ocorrido e também a **cadeia de chamadas aos métodos (method-call stack)** no momento em que ocorreu o erro;
  - O *stack trace* inclui o **caminho da execução que levou a exceção** método por método;

Tratamento de Exceções (cont.)

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at DivideByZeroNoExceptionHandling.main(
                DivideByZeroNoExceptionHandling.java:20)
```

Tratamento de Exceções (cont.)

- Ao informarmos um tipo diferente do esperado, uma exceção **InputMismatchException** é lançada;
- A partir do final do *stack trace*, vemos que a exceção foi detectada na linha 20 do *main*
- Na linha superior, a exceção ocorre no método *nextInt*
  - Ao invés de aparecer o nome do arquivo e número da linha, aparece o texto "*Unknown Source*"

try e *catch*

- O exemplo a seguir utiliza o tratamento de exceções para processar quaisquer exceções *ArithmeticException* e *InputMismatchException*
  - Se o usuário cometer um erro, o **programa captura** e trata a exceção;
  - Neste caso, permite que o usuário informe os dados novamente.

try e *catch* (cont.)

```
import java.util.InputMismatchException;
import java.util.Scanner;

public class DivideByZeroWithExceptionHandling {

    public static int quotient(int num, int den) {
        if (den == 0) throw new ArithmeticException();
        return num / den;
    }
```

try e *catch* (cont.)

```
public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    boolean continueLoop = true;
    do{
        try{
            System.out.println("Digite o numerador:");
            int num = input.nextInt();
            System.out.println("Digite o denominador:");
            int den = input.nextInt();
            int res = quotient(num, den);
            System.out.printf("\nResultado: %d / %d = %d\n", num,
                den, res);
            continueLoop = false;
        }
```

try e *catch* (cont.)

```
            catch(InputMismatchException e){
                System.err.printf("\nExcecao: %s\n", e);
                input.nextLine();
                System.out.println("Inserir numeros inteiros\n");
            }
            catch(ArithmeticException e){
                System.err.printf("\nExcecao: %s\n", e);
                System.out.println("Divisao por zero. Tente novamente\n
                    ");
            }
        }while(continueLoop);
    }

}
```

try e *catch* (cont.)

```
Digite o numerador: 100
Digite o denominador: 0

Excecao: java.lang.ArithmeticException: / by zero
Divisao por zero. Tente novamente

Digite o numerador: 100
Digite o denominador: ola

Excecao: java.util.InputMismatchException
Inserir numeros inteiros

Digite o numerador: 100
Digite o denominador: 7

Resultado: 100 / 7 = 14
```

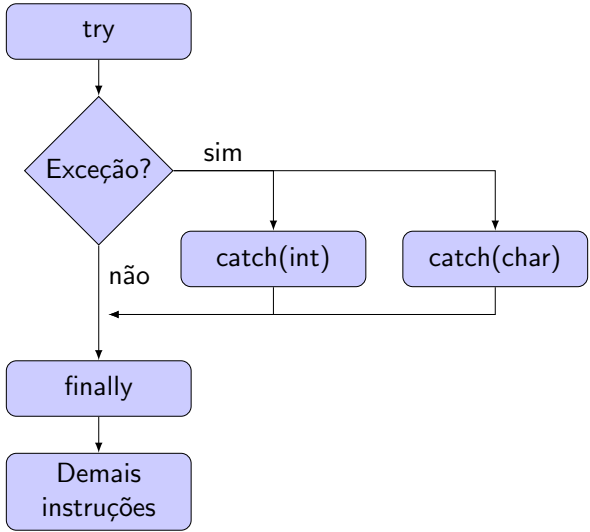
try e *catch* (cont.)

- A classe **ArithmeticException** não precisa ser importada porque está localizada no pacote *java.lang*
- A classe **InputMismatchException** precisa ser importada.
- O bloco *try* neste exemplo é seguido de dois blocos *catch*
  - Um para cada tipo de exceção.

try e *catch* (cont.)

- Um bloco *catch* **captura e trata** uma exceção
  - Começa com a palavra *catch*, seguido por um único parâmetro entre parênteses e um bloco de código entre { e };
  - O parâmetro especifica o tipo da exceção a ser capturada.
- Pelo menos um bloco *catch* ou um bloco *finally* devem seguir imediatamente um bloco *try*
  - O bloco cujo objeto parâmetro seja do mesmo tipo ao da exceção lançada será executado;

Modelo de Terminação



Quando Utilizar Exceções

- Erros síncronos (na execução de uma instrução)
  - Índice de vetor fora dos limites;
  - Overflow aritmético (valor fora dos limites do tipo);
  - Divisão por zero;
  - Parâmetros inválidos;
  - Alocação de memória excessiva ou indisponível.

Quando Utilizar Exceções (cont.)

- Exceções não devem ser utilizadas para erros assíncronos (pararelos à execução do programa)
  - Erros de I/O de disco;
  - Cliques de mouse e pressionamento de teclas.
  - Obviamente, exceções não tratam erros em **tempo de compilação**

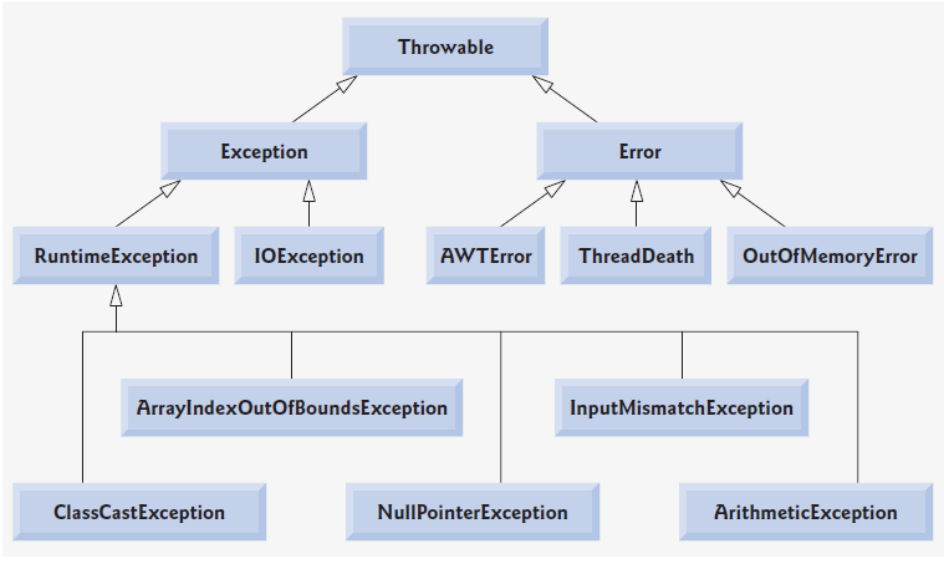
Hierarquia de Exceções Java

- Todas as classes de exceção Java herdam direta ou indiretamente da classe **Exception**
  - É possível estender esta hierarquia para criar nossas próprias classes de exceção;
  - A hierarquia específica é iniciada pela classe **Throwable** (uma subclasse de *Object*)

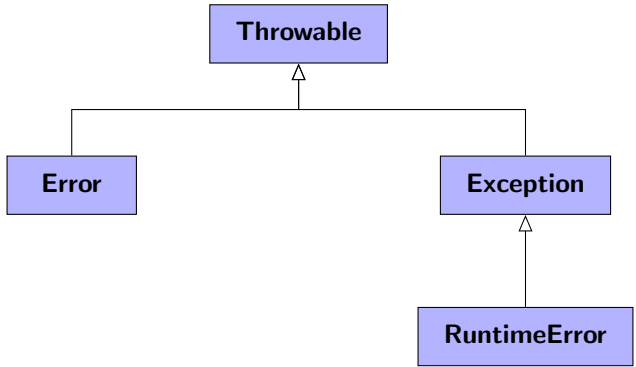
Hierarquia de Exceções Java (cont.)

- Somente objetos *Throwable* podem ser utilizados com o mecanismo de tratamento de exceções.
- A classe **Exception** e suas subclasses representam situações excepcionais que ocorrem em um programa e que podem ser capturadas por ele próprio;
- A classe **Error** e suas subclasses representam situações anormais que podem acontecer na JVM.

Hierarquia de Exceções Java (cont.)



Hierarquia de Exceções Java (cont.)



Hierarquia de Exceções Java (cont.)

- **Erro (Error)**
  - Exceções tão graves que a aplicação não tem como resolver o problema
  - O programa não tem o que fazer para resolver o problema que eles apontam
  - Exemplos de erros são *OutOfMemoryError* que é lançada quando o programa precisa de mais memória

Hierarquia de Exceções Java (cont.)

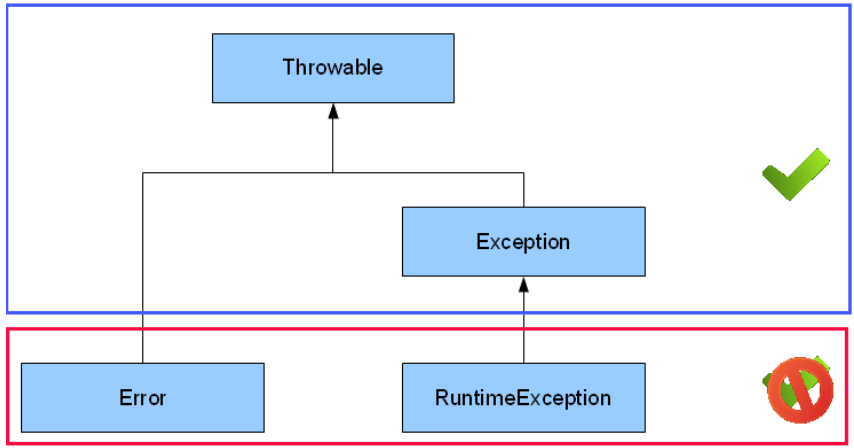
- **Exceção de Contigência (Exception)**
  - São aquelas que a aplicação pode causar ou não, mas que tem que tratar explicitamente
  - O exemplo clássico é a exceção *FileNotFoundException* que significa que o arquivo que estamos tentando ler, não existe
  - Compilador exige que sejam ou capturadas ou declaradas pelo método que potencialmente as provoca

Hierarquia de Exceções Java (cont.)

- **Falha (Runtime EXception)**
  - são exceções que a aplicação causa e pode resolver
  - Representam erros de lógica de programação que devem ser corrigidos
  - Exemplo, *NullPointerExcetion* quando se passa um parâmetro para um método e não pode ser usado pelo método

Hierarquia de Exceções Java (cont.)

Exceções verificadas e não verificadas



catch-or-declare

- Códigos "válidos" em Java deve honrar o requisito *catch-or-declare*
- Códigos que possam lançar certas exceções devem cumprir com uma das opções abaixo:
  - Possuir uma estrutura *try/catch* que manipule a exceção;
  - Declarar que o método correspondente pode lançar exceções, através de uma cláusula *throw*

catch-or-declare (cont.)

- Esta distinção é importante, porque o compilador Java força o **catch-or-declare** para exceções verificadas
- O tipo da exceção determina quando uma exceção é verificada ou não
  - Todas as exceções que herdam direta ou indiretamente da classe *RuntimeException* e *Error* são exceções não verificadas;
  - Todas as exceções que herdam direta ou indiretamente da classe *Exception* mas não da classe *RuntimeException* são exceções verificadas.

Blocos *finally*

- Programas que obtêm certos tipos de recursos devem devolvê-los aos sistema explicitamente para evitar a perda dos mesmos (*resource leaks*);
- O bloco *finally* é opcional, e se presente, é colocado depois do último bloco *catch*
  - Consiste da palavra finally seguida por um bloco de comandos entre { e }

Blocos *finally* (cont.)

- Justamente por quase sempre ser executado, um bloco *finally* contém códigos de liberação de recursos
  - Por exemplo, fechar conexões de rede, arquivos, etc.
- O exemplo a seguir demonstra a execução de um bloco *finally* mesmo uma exceção não sendo lançada no bloco *try* correspondente

Blocos *finally* (cont.)

```
public class UsingExceptions
{
   public static void main( String args[] )
   {
      try
      {
         throwException();
      }
      catch ( Exception exception ) // excecao lancada por throwException
      {
         System.err.println( "Exception handled in main" );
      }
      doesNotThrowException();
   }
```

Blocos *finally* (cont.)

```
// demonstra try...catch...finally
   public static void throwException() throws Exception {
      try { // lanca uma excecao e imediatamente a captura
         System.out.println( "Method throwException" );
         throw new Exception(); // gera a excecao
      }
      catch ( Exception exception ) { // captura a excecao
         System.err.println("Exception handled in method throwException
             " );
         throw exception; // lanca novamente
         // qualquer codigo aqui serua inatingivel
      }
      finally // executa independentemente do que ocorre no try...catch
      {
         System.err.println( "Finally executed in throwException" );
      }
      // qualquer codigo aqui seria inatingivel
   }
```

Blocos *finally* (cont.)

```
// demonstra o finally quando nao ocorre excecao
   public static void doesNotThrowException()    {
      try  // o bloco try nao lanca excecoes
      {
         System.out.println( "Method doesNotThrowException" );
      }
      catch ( Exception exception ) // nao e executado
      {
         System.err.println( exception );
      }
      finally  // executa independentemente do que ocorre no try...catch
      {
         System.err.println( "Finally executed in doesNotThrowException
             " );
      }

      System.out.println( "End of method doesNotThrowException" );
   }
}
```

Blocos *finally* (cont.)

```
Method throwException
Exception handled in method throwException
Finally executed in throwException
Exception handled in main
Method doesNotThrowException
Finally executed in doesNotThrowException
End of method doesNotThrowE
```

Blocos *finally*

- Note o uso de **System.err** para exibir os dados
  - Direciona o conteúdo para a fluxo padrão de erros
  - Se não for redirecionado, os dados serão exibidos no *prompt* de comando

throw

- A instrução *throw* é executada para indicar que ocorreu uma exceção
- Até aqui tratamos exceções lançadas por outros métodos
  - Podemos lançar as nossas próprias;
  - Deve ser especificado um objeto a ser lançado
  - De qualquer classe derivada da classe Throwable.

throw (cont.)

- Exceções podem ser relançadas
  - Quando um bloco *catch* recebe uma exceção, mas é incapaz de processá-la totalmente, ele pode relançá-la para outro bloco *try-catch* mais externo;
  - Blocos *finally* não podem relançar exceções.

Cláusula *throws*

- Uma cláusula *throws* especifica as exceções que um método lança
  - Aparece entre a lista de parâmetros e o corpo do método;
  - As exceções podem ser lançadas explicitamente dentro do próprio método ou por outros métodos chamados dentro do primeiro.

Desfazendo a Pilha

- Quando uma exceção é lançada mas não capturada em um determinado escopo, a pilha de chamadas de métodos é desfeita passo a passo
  - A cada passo, tenta-se capturar a exceção;
  - Este processo é chamado de **stack unwinding**
- O exemplo a seguir demonstra este processo

Desfazendo a Pilha (cont.)

```
public class UsingExceptions
{
   public static void main( String args[] )
   {
      try
      {
         throwException();
      }
      catch ( Exception exception ) // exceção lancada em throwException
      {
         System.err.println( "Exception handled in main" );
      }
   }
```

Desfazendo a Pilha (cont.)

```
// throwException lanca uma excecao que nao e capturada neste metodo
   public static void throwException() throws Exception
   {
      try // lanca uma excecao e a captura no main
      {
         System.out.println( "Method throwException" );
         throw new Exception(); // gera a excecao
      }
      catch ( RuntimeException runtimeException ) // captura o tipo incorreto
      {
         System.err.println( "Exception handled in method
            throwException" );
      }
      finally // sempre sera executado
      {
         System.err.println( "Finally is always executed" );
      }
   }
}
```

Desfazendo a Pilha (cont.)

```
Method throwException
Finally is always executed
Exception handled in main
```

printStackTrace, getStackTrace e getMessage

- A classe **Throwable** fornece três métodos para obtermos informações sobre exceções:
  - *printStackTrace*: exibe a *stack trace* no fluxo de erro padrão;
  - *getStackTrace*: retorna os dados que serão exibidos pelo método anterior;
  - *getMessage*: retorna uma *string* descritiva armazenada na exceção.
- O exemplo a seguir demonstra a utilização destes métodos

printStackTrace, getStackTrace e getMessage (cont.)

```
public class UsingExceptions {
   public static void main( String args[] ) {
      try {
         method1();
      }
      catch ( Exception exception ){ // captura a excecao
         System.err.printf( "%s\n\n", exception.getMessage() );
         exception.printStackTrace(); // imprime o stack trace

         // obtem a informacao do stack trace
         StackTraceElement[] traceElements = exception.getStackTrace();

         System.out.println( "\nStack trace from getStackTrace:" );
         System.out.println( "Class\t\tFile\t\t\tLine\tMethod" );
```

printStackTrace, getStackTrace e getMessage (cont.)

```
        // itera pelos elementos para obter a descricao da excecao
        for ( StackTraceElement element : traceElements )
        {
            System.out.printf( "%s\t", element.getClassName() );
            System.out.printf( "%s\t", element.getFileName() );
            System.out.printf( "%s\t", element.getLineNumber() );
            System.out.printf( "%s\n", element.getMethodName() );
        }
    }
}
```

printStackTrace, getStackTrace e getMessage (cont.)

```
    // lanca a excecao de volta para o main
    public static void method1() throws Exception
    {
        method2();
    }

    // lanca a excecao de volta para o method1
    public static void method2() throws Exception
    {
        method3();
    }

    // lanca a excecao de volta para o method2
    public static void method3() throws Exception
    {
        throw new Exception( "Exception thrown in method3" );
    }
}
```

printStackTrace, getStackTrace e getMessage (cont.)

```
 Exception thrown in method3

java.lang.Exception: Exception thrown in method3
 at UsingExceptions.method3(UsingExceptions.java:49)
 at UsingExceptions.method2(UsingExceptions.java:43)
 at UsingExceptions.method1(UsingExceptions.java:37)
 at UsingExceptions.main(UsingExceptions.java:10)

Stack trace from getStackTrace:
Class            File                 Line Method
UsingExceptions  UsingExceptions.java 49   method3
UsingExceptions  UsingExceptions.java 43   method2
UsingExceptions  UsingExceptions.java 37   method1
UsingExceptions  UsingExceptions.java 10   main
```

printStackTrace, getStackTrace e getMessage (cont.)

- Objetos da classe StackTraceElement armazenam informações da stack trace sobre a exceção;
- Possuem os métodos:
  - **getClassName**: retorna o nome da classe;
  - **getFileName**: retorna o nome do arquivo;
  - **getLineNumber**: retorna o número da linha;
  - **getMethodName**: retorna o nome do método.

Exceções Encadeadas

```
public class UsingChainedExceptions {
   public static void main( String args[] ) {
      try {
         method1();
      }
      catch ( Exception exception ) // excecao lancada por method1
      {
         exception.printStackTrace();
      }
   }
```

Exceções Encadeadas (cont.)

```
// lanca uma excecao de volta ao main
public static void method1() throws Exception {
    try {
        method2();
    }
    catch ( Exception exception ) // excecao lancada por method2
    {
        throw new Exception( "Exception thrown in method1", exception
            );
    }
}
```

Exceções Encadeadas (cont.)

```
// lanca uma excecao de volta ao method1
   public static void method2() throws Exception {
      try {
         method3();
      }
      catch ( Exception exception ) // excecao lancada por method3
      {
         throw new Exception( "Exception thrown in method2", exception
             );
      }
   }

   // lanca uma excecao de volta ao method2
   public static void method3() throws Exception {
      throw new Exception( "Exception thrown in method3" );
   }
}
```

Declarando Novos Tipos de Exceções

- Uma nova classe de exceções deve estender uma classe de exceções já existente
  - Para garantir que ela funcionará com o mecanismo de tratamento de exceções.

Declarando Novos Tipos de Exceções (cont.)

- Como qualquer outra classe, uma classe de exceções contém atributos e métodos
  - Porém, tipicamente contém apenas dois construtores
    - Um que não possui argumentos e informa uma mensagem padrão ao construtor da superclasse;
    - Um que possui recebe uma string com uma mensagem personalizada como argumento e a repassa ao construtor da superclasse.

Declarando Novos Tipos de Exceções (cont.)

- Antes de criar uma nova classe, é necessário analisar a API Java para decidir qual classe deve ser utilizada como superclasse
  - É uma boa prática que seja uma classe relacionada com a natureza da exceção

Declarando Novos Tipos de Exceções (cont.)

- Programadores gastam um bom tempo realizando a manutenção e a depuração de códigos;
- Para facilitar estas tarefas, podemos especificar os estados esperados antes e depois da execução de um método
  - Estes estados são chamados de **pré-condições** e **pós-condições**

Declarando Novos Tipos de Exceções (cont.)

- Por exemplo, o método *charAt* da classe *String*, que recebe um índice como argumento
  - **Pré-condição**: o argumento deve ser maior ou igual a zero, e menor que o comprimento da *string*;
  - **Pós-condição**: retornar o caractere no índice indicado;
  - Caso contrário, o método lançará a exceção **IndexOutOfBoundsException**

Asserções

- Quando implementamos e depuramos uma classe, é útil criarmos certas condições que devem ser verdadeiras em determinados pontos do código
  - Estas condições são chamadas de asserções;
  - Nos ajudam a capturar eventuais *bugs* e a identificar possíveis erros de lógica;
  - Pré-condições e pós-condições são asserções relativas aos métodos.
- Java inclui duas versões da instrução assert para validação de asserções

Asserções (cont.)

- A instrução *assert* avalia uma expressão booleana e determina se a mesma é verdadeira ou falsa; A primeira forma é

```
assert expressao;
```

- Uma exceção **AssertionError** é lançada caso a expressão seja falsa;

- A segunda forma é

```
assert expressao1 : expressao2;
```

- Uma exceção **AssertionError** é lançada caso a primeira expressão seja falsa, com a segunda expressão como mensagem de erro.

Asserções (cont.)

```
import java.util.Scanner;

public class AssertTest
{
   public static void main( String args[] )
   {
      Scanner input = new Scanner( System.in );

      System.out.print( "Enter a number between 0 and 10: " );
      int number = input.nextInt();

      // assercao para verificar que o valor absoluto esta entre 0 e 10
      assert ( number >= 0 && number <= 10 ) :
        "bad number: " + number;

      System.out.printf( "You entered %d\n", number );
   }
}
```

Asserções (cont.)

```
Enter a number between 0 and 10: 5
You entered 5

Enter a number between 0 and 10: 50
Exception in thread "main" java.lang.AssertionError: bad number: 50
        at AssertTest.main(AssertTest.java:15)
```

Asserções (cont.)

- Por padrão, as asserções são desabilitadas quando o programa é executado
  - Reduzem a performance e são inúteis para o usuário final.
- Para habilitar as asserções, é necessário utilizar a opção **–ea** na linha de comando

```
java -ea AssertTest
```

Genéricos

- A programação de genéricos nos permite criar modelos genéricos
  - Métodos genéricos especificam em uma única declaração um **conjunto de métodos relacionados**;
  - Classes genéricas especificam em uma única declaração um **conjunto de tipos relacionados**;

Métodos Genéricos

- A declaração de métodos genéricos começa com a **seção de parâmetro de tipo**, delimitado por $<$ e $>$
  - Antes do tipo de retorno no método;
  - Cada seção contém um ou mais parâmetros de tipo, separados por vírgulas.

Métodos Genéricos (cont.)

- O corpo de um método genérico é declarado de forma semelhante a um método comum;
- Parâmetros de tipo podem somente representar tipos de **referências**
  - Tipos primitivos como *int*, *double* e *char* não;
  - Todo os dados enviados como parâmetros devem ser objetos de classes ou interfaces

Métodos Genéricos (cont.)

```
public class GenericMethodTest {
   //metodo generico
   public static < E > void printArray( E[] inputArray ){
      // exibe os elementos do vetor
      for ( E element : inputArray )
         System.out.printf( "%s ", element );
      System.out.println();
   }

   public static void main( String args[] ) {
      // cria vetores dos tipos Integer, Double e Character
      Integer[] integerArray = {1,2,3,4,5,6};
      Double[] doubleArray = {1.1,2.2,3.3,4.4,5.5,6.6};
     Character[] characterArray ={'H','E','L','L','O'};
```

Métodos Genéricos (cont.)

```
        System.out.println( "\nintegerArray contains:" );
        printArray( integerArray ); // envia um vetor de Integer
        System.out.println( "\ndoubleArray contains:" );
        printArray( doubleArray ); // envia um vetor de Double
        System.out.println("\ncharacterArray contains:");
        // envia um vetor de Character
        printArray(characterArray);
    }
}
```

Métodos Genéricos (cont.)

```
integerArray contains
1 2 3 4 5 6

doubleArray contains
1.1 2.2 3.3 4.4 5.5 6.6

characterArray contains
H E L L O
```

Métodos Genéricos (cont.)

- Quando o compilador traduz o código para *bytecode*, os métodos genéricos têm seus argumentos substituídos por tipos de verdade
  - Por padrão, o tipo Object é utilizado;
  - Diferentemente do que ocorre em C++, em que uma cópia para cada tipo utilizado é criada.
- A seguir é apresentado o equivalente ao método genérico do código anterior depois de compilado

Métodos Genéricos (cont.)

```
public static void printArray( Object[] inputArray )
  {
     // exibe os elementos do vetor
     for (Object element : inputArray )
        System.out.printf( "%s ", element );

     System.out.println();
  }
```

Métodos Genéricos (cont.)

- Os métodos genéricos podem ser sobrecarregados
  - Por outros métodos genéricos;
  - Por métodos específicos
    - Inclusive tendo os mesmos parâmetros;
    - Têm precedência maior em relação ao genérico

Classes Genéricas

- Os **conceitos de estruturas de dados**, como uma pilha, são **independentes dos tipos** dos elementos que elas manipulam
  - Desta forma, podemos **criar uma classe que descreva o comportamento** de uma estrutura de dados, de uma maneira independente
  - Ao instanciarmos esta **classe genérica**, podemos especificar qual é o tipo desejado;
  - **Esta capacidade permite** um grande avanço na **reusabilidade** de código.
- O tratamento dispensado pelo compilador às classes genéricas é semelhante ao dispensado aos métodos genéricos.

Classes Genéricas (cont.)

- Estas classes são conhecidas como **classes parametrizadas**
  - Ou *tipos parametrizados*, uma vez que podem receber um ou mais parâmetros;
  - Tais parâmetros representam apenas tipos de referência
    - Ou seja, uma estrutura não poderia ser instanciada com um tipo **primitivo**;
    - No entanto, podemos utilizar o **autoboxing** para converter tipos primitivos em objetos

Classes Genéricas (cont.)

- **Autoboxing** consiste em atribuir um tipo primitivo a uma variável ou estrutura cujo tipo é uma classe empacotadora (*wrapper class*)
  - A conversão é implícita;
  - Por exemplo, *int* para *Integer*
- O **Auto-Unboxing** é o processo contrário;

Classes Genéricas (cont.)

- Em Java há 8 classes empacotadoras
  - *Byte*, *Short*, *Integer*, *Long*, *Float*, *Double*, *Character* e *Boolean*
  - Todos declaradas no pacote *java.lang*

Classes Genéricas (cont.)

- O exemplo a seguir apresenta a declaração de uma classe que descreve uma pilha genérica
  - O parâmetro *E* representa o tipo dos elementos a serem manipulados pela pilha
    - Uma classe genérica pode possuir mais que um parâmetro de tipo, separados por vírgula.
  - Este parâmetro é utilizado ao longo do código nos trechos em que é necessário indicar o tipo dos elementos

EmptyStackException.java

```
public class EmptyStackException extends RuntimeException{

    public EmptyStackException(){
        this("Stack is empty");
    }

    public EmptyStackException(String msg){
        super(msg);
    }
}
```

FullStackException.java

```
public class FullStackException extends RuntimeException {

    public FullStackException(){
        this("Stack is full");
    }

    public FullStackException(String msg){
        super(msg);
    }
}
```

Stack.java

```
public class Stack< E >
{
   private final int size; // numero de elementos da pilha
   private int top; // indice do topo
   private E[] elements; // vetor para armazenar os elementos
   // o tamanho padrao e 10
   public Stack()
   {
      this( 10 );
   }
   // constroi uma pilha com um tamanho especificado
   public Stack( int s )
   {
      size = s > 0 ? s : 10;
      top = -1; // pilha vazia inicialmente

      elements = ( E[] ) new Object[ size ]; // cria o vetor
   }
```

Stack.java (cont.)

```
    public void push( E pushValue )
    {
       if ( top == size - 1 )
          throw new FullStackException( String.format( "Stack is full,
              cannot push %s", pushValue ) );

       elements[ ++top ] = pushValue;
    }
    public E pop()
    {
       if ( top == -1 )
          throw new EmptyStackException( "Stack is empty, cannot pop" );

       return elements[ top-- ];
    }
}
```

StackTest.java

```
public class StackTest {
   private Double[] doubleElements={1.1, 2.2, 3.3, 4.4, 5.5, 6.6};
   private Integer[] integerElements = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
       11 };

   private Stack< Double > doubleStack; // pilha de Double
   private Stack< Integer > integerStack; // pilha de Integer

   public void testStacks()  {
      doubleStack = new Stack< Double >( 5 );
      integerStack = new Stack< Integer >( 10 );

      testPush("doubleStack", doubleStack, doubleElements);
      testPop("doubleStack", doubleStack);
      testPush("integerStack", integerStack, integerElements);
      testPop("integerStack", integerStack);
   }
```

StackTest.java (cont.)

```
// metodo generico que testa o metodo push da classe generica
   public < T > void testPush( String name,
        Stack< T > stack, T[] elements ) {
      try {
         System.out.printf( "\nPushing elements onto %s\n", name );

         for ( T element : elements ) {
            System.out.printf( "%s ", element );
            stack.push( element );
         }
      }
      catch ( FullStackException fullStackException )
      {
         System.out.println();
         fullStackException.printStackTrace();
      }
   }
```

StackTest.java (cont.)

```
// metodo generico que testa o metodo pop da classe generica
public < T > void testPop( String name, Stack< T > stack ) {
   try {
      System.out.printf( "\nPopping elements from %s\n", name );
      T popValue;

      while ( true ) {
         popValue = stack.pop();
         System.out.printf( "%s ", popValue );
      }
   }
   catch( EmptyStackException emptyStackException ){
      System.out.println();
      emptyStackException.printStackTrace();
   }
}
```

StackTest.java (cont.)

```
    public static void main( String args[] ) {
        StackTest application = new StackTest();
        application.testStacks();
    }
}
```

StackTest.java (cont.)

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5 6.6
javastackgenerico.FullStackException: Stack is full, cannot push 6.6
at javastackgenerico.Stack.push(Stack.java:20)
at javastackgenerico.JavaStackGenerico.testPush(JavaStackGenerico.java
    :36)
at javastackgenerico.JavaStackGenerico.testStacks(JavaStackGenerico.
    java:21)
at javastackgenerico.JavaStackGenerico.main(JavaStackGenerico.java:69)
```

StackTest.java (cont.)

```
Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
javastackgenerico.EmptyStackException: Stack is empty, cannot pop
at javastackgenerico.Stack.pop(Stack.java:26)
at javastackgenerico.JavaStackGenerico.testPop(JavaStackGenerico.java
    :56)
at javastackgenerico.JavaStackGenerico.testStacks(JavaStackGenerico.
    java:22)
at javastackgenerico.JavaStackGenerico.main(JavaStackGenerico.java:69)
```

Tipos "*Crus*"

- Os exemplos anteriores instanciavam a classe *Stack* com tipos *Integer* e *Double* passados por argumento
  - Também é possível instanciar uma classe genérica sem especificar o tipo, como a seguir:

```
// nenhum tipo especificado
Stack objectStack = new Stack( 5 );
```

  - Neste caso, dizermos que o objeto possui um tipo "cru" (*raw type*);
  - O compilador utiliza o tipo *Object* implicitamente, criando uma pilha para qualquer tipo de elemento;
  - Há insegurança quanto ao tipo dos dados armazenados em um tipo cru

Tipos "*Crus*" (cont.)

- Os tipos crus são importante para a compatibilidade das versões antigas do Java
- É possível atribuir uma estrutura de tipo cru a uma estrutura que especifique o tipo, como abaixo:

```
Stack rawTypeStack2 = new Stack< Double >( 5 );
```

Coringas em Métodos Genéricos

- Quando não pudermos determinar a classe específica dos elementos que serão passados a um genérico, podemos utilizar um **coringa** (**wildcard**)
  - Por exemplo, em um método que soma os elementos de um vetor, podemos não saber se tais elementos serão dos tipos *Integer* ou *Double*
  - Podemos então indicar simplesmente que o tipo será o de uma classe que estende a classe *Number*
    - De fato, *Integer* e *Double* são subclasses de *Number*
  - Um parâmetro coringa é indicado por uma **?**, como abaixo:

```
ArrayList< ? extends Number > list
```

Coringas em Métodos Genéricos (cont.)

```
public static double sum( ArrayList< ? extends Number > list )
{
      double total = 0;

       for ( Number element : list )
         total += element.doubleValue();

      return total;
}
```

Coringas em Métodos Genéricos (cont.)

- Uma desvantagem desta sintaxe é que o símbolo ? não pode ser utilizado como o nome de um tipo ao longo do método
  - Por exemplo, no for aprimorado não podemos substituir Number por **?**
- Uma alternativa é declarar o método como a seguir:

```
public static <T extends Number> double sum( ArrayList< T > list )
```

FIM