Centro Federal de Educação Tecnológica do Amazonas

Java Avançado

Prof. Tiago Eugenio de Melo
tiago@comunidadesol.org

Sumário

- Exceção
- Arquivos
- GUI


edit
delete
EXCEÇÕES

Exceção

- O que é exceção?
  - É uma indicação de um problema que ocorre durante a execução de um programa.
- O tratamento de exceção permite a escrita de programas robustos e tolerantes a falhas.
- Java permite o tratamento de exceções.
- Semelhante à forma tratada por C++, Java se utiliza das seguintes instruções: `try`, `catch`, `throw`, `throws` e `finally`.
- Toda exceção deve ser estendida da classe `Throwable`.

Exceção

- Benefícios no uso de exceções:
  - Separação entre o código que trata os erros e o código que é executado normalmente pelo programa.
  - Uma maneira de forçar uma resposta à falha.

Exceção

- Exemplo: DivideByZeroNoExceptionHandling

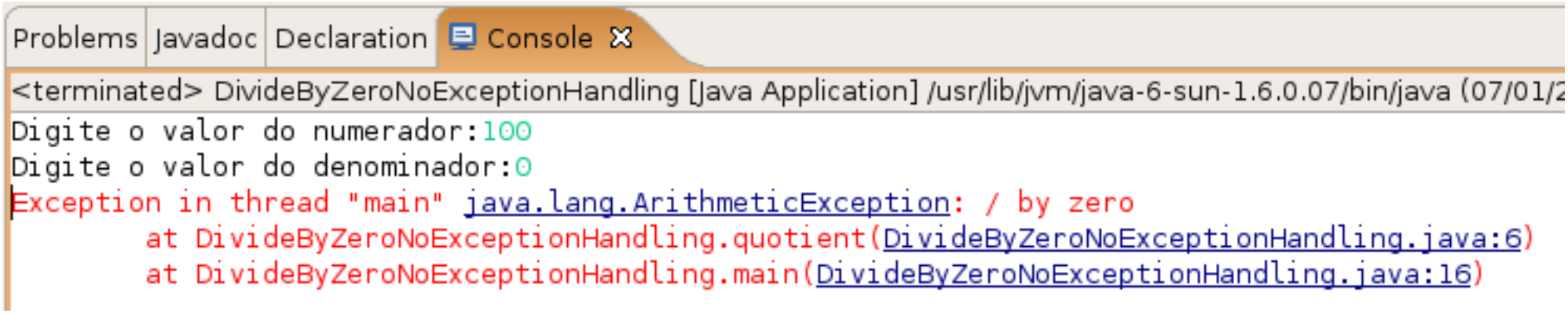
*DivideByZeroNoExceptionHandling.java

```
1  import java.util.Scanner;
2
3  public class DivideByZeroNoExceptionHandling {
4
5      public static int quotient (int numerator, int denominator) {
6          return numerator / denominator;
7      }
8
9      public static void main (String args[]) {
10         Scanner scanner = new Scanner (System.in);
11         System.out.print("Digite o valor do numerador:");
12         int numerator = scanner.nextInt();
13         System.out.print("Digite o valor do denominador:");
14         int denominator = scanner.nextInt();
15
16         int result = quotient (numerator, denominator);
17         System.out.printf("\nResult: %d / %d = %d\n", numerator, denominator, result);
18     }
19 }
```

Programa 1

Exceção

- No Programa 1, é possível rodar o programa com os valores 100 e 2.
- A execução traria a seguinte mensagem: *Result: 100 / 2 = 50*.
- Porém, se os valores de entrada fossem 100 e 0, o resultado seria uma mensagem de exceção.



Exceção

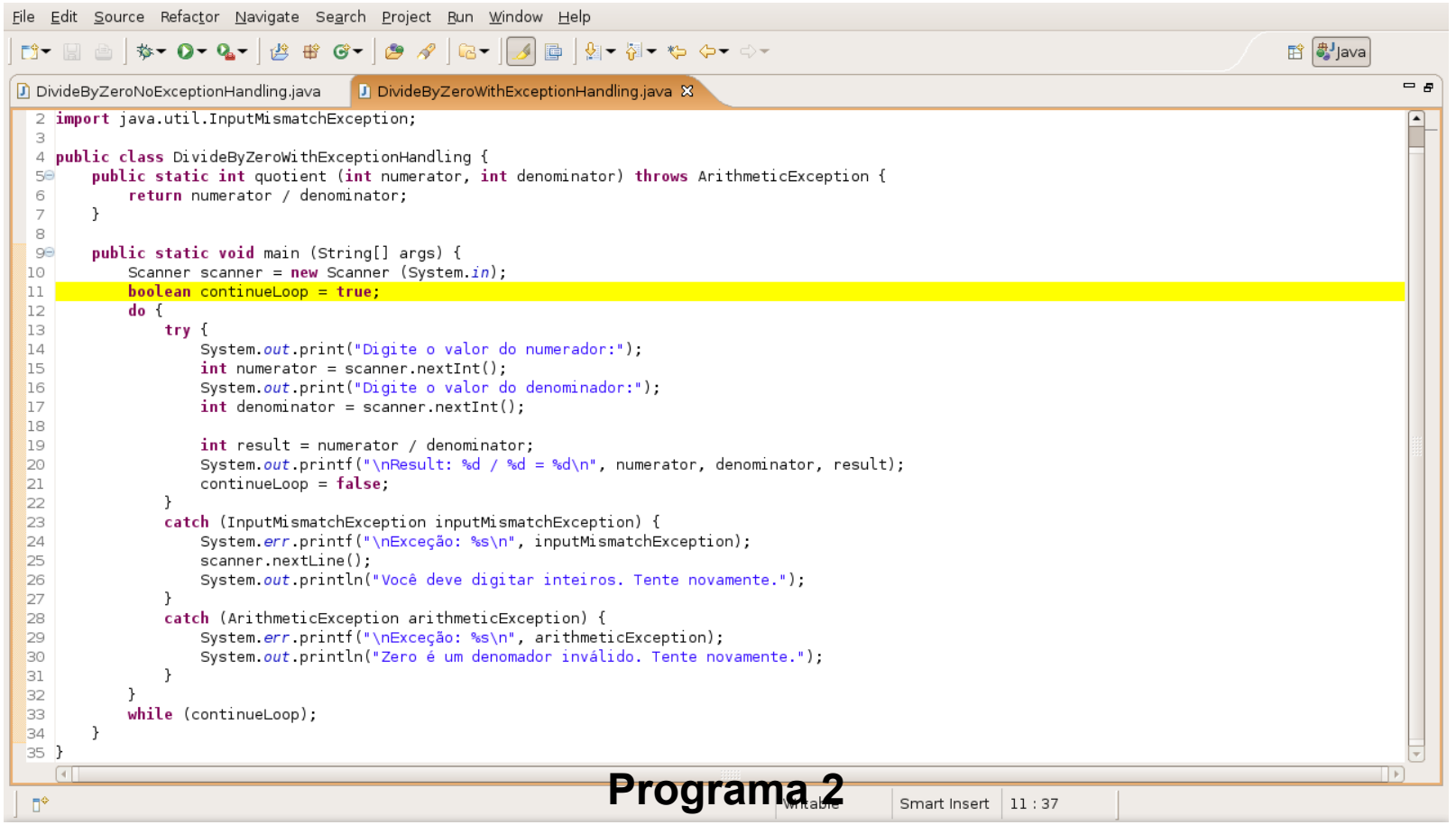
- E o que ocorreria se ao invés de números, fosse digitado uma string?

Exceção

- Nos dois exemplos, ocorreram as exceções ArithmeticException e InputMismatchException.
- Quando as exceções ocorrem os programas também se fecham.
- Porém, em Java, isso nem sempre ocorre, pois às vezes um programa pode continuar mesmo que uma exceção tenha ocorrido e um rastreamento tenha sido impresso. Essa situação é possível no uso de threads.

Exceção

- O tratamento para as duas exceções anteriores pode ser visto no exemplo abaixo:



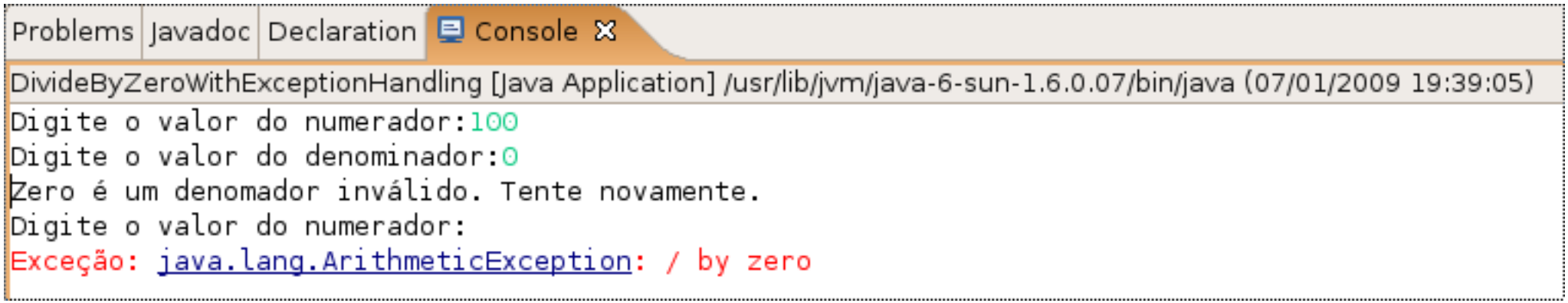
Programa 2

Exceção

- O Programa 2 trata as duas exceções identificadas (ArithmeticException e InputMismatchException).
- O tratamento consiste em verificar a exceção, apresentá-la ao usuário e solicitar que entre com os dados corretamente.

Exceção

- Exemplos:



Exceção

- Sintaxe do tratamento de exceção em Java:

`try` { bloco }

`catch` ( ) { bloco }

...

`catch` ( ) { bloco }

`finally` { bloco } // opcional

- Pelo menos um bloco `catch` ou `finally` deve se seguir imediatamente ao bloco `try`.

Exceção

- É considerado erro de sintaxe colocar código entre um bloco `try` e seus blocos `catch` correspondentes.
- Cada bloco `catch` só poderá ter um único parâmetro.
- É um erro de compilação capturar o mesmo tipo de exceção em dois blocos `catch` diferentes em uma única instrução `try`.

Exceção

- Uma exceção não capturada é uma exceção à qual não há nenhum bloco `catch` correspondente.
- O nome do parâmetro pode ser escolhido pelo programador, porém este deve refletir o tipo de exceção tratada.
- Como ocorrem com qualquer outro bloco de código, quando um bloco `try` terminar, as variáveis locais declaradas no bloco saem de escopo (e são destruídas).

Exceção

- Quando utilizar o tratamento de exceções?
  - O tratamento de exceções é projetado para processar erros síncronos, que ocorrem quando uma instrução é executada.
  - Os exemplos comuns são índice fora do intervalo do array, estouro aritmético, divisão por zero, parâmetros inválidos do método, interrupção de thread e alocação de memória malsucedida.
  - O tratamento de exceções não é projetado para processar problemas associados com eventos assíncronos, que ocorrem paralelamente com o fluxo de controle de programa e independentemente dele.

Exceção

- A estratégia de tratamento de exceções deve ser feito desde o princípio do processo de projeto. Pode ser difícil incluir um tratamento de exceções eficiente depois que um sistema foi implementado.

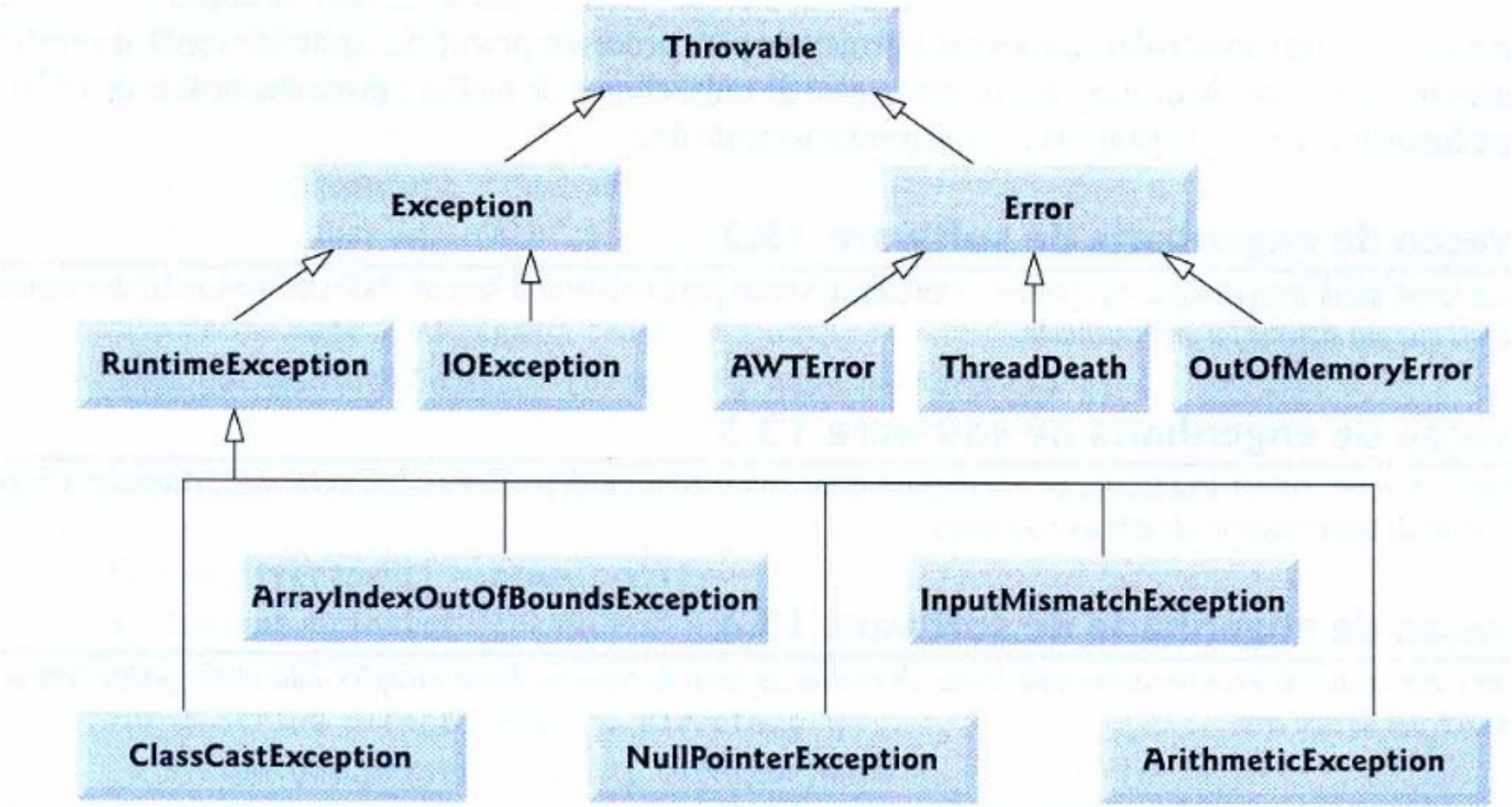
Exceção

- Se uma exceção não é tratada no bloco `try-catch` atual, ela é propagada para o método chamador.
- Se uma exceção volta até o método `main` e não é tratada lá, o programa é terminado de forma anormal.

Hierarquia de exceções

- Todas as classes do Java herdam, direta ou indiretamente, da classe Exception, formando uma hierarquia de herança.
- Os programadores podem estender essa hierarquia para criar suas próprias classes de exceção.

Hierarquia de exceções



Parte da hierarquia de herança da classe `Throwable`.

Hierarquia de exceções

- A figura anterior mostra uma pequena parte da hierarquia da classe *Throwable* (uma subclasse da *Object*), que é uma superclasse da *Exception*.
- Somente objetos *Throwable* podem ser utilizados com o mecanismo de tratamento de exceções.
- A classe *Throwable* tem duas subclasses: *Exception* e *Error*.

Hierarquia de exceções

- A classe *Error* e suas subclasses (por exemplo, *OutOfMemoryError*) representam situações anormais que poderiam acontecer na JVM.
- Erros acontecem raramente e não devem ser capturados pelos aplicativos, pois normalmente não é possível que os programas se recuperem de erros.

Hierarquia de exceções

- Java reconhece duas categorias de exceções:
  - Verificadas (*checked*).
  - Não-verificadas (*unchecked*).
- A diferença é que o compilador impõe um requisito catch-or-declare às exceções verificadas.
- O tipo de uma exceção determina se ela é verificada ou não é verificada.

Hierarquia de exceções

- Todos os tipos de exceção que são subclasses diretas ou indiretas da classe RuntimeException são exceções não verificadas.
- Todas as classes que herdam da classe Exception, mas não da classe RuntimeException são consideradas exceções verificadas.
- As classes que herdam da classe Error são consideradas não verificadas.

Hierarquia de exceções

- O compilador verifica cada chamada de método e declaração de método para determinar se o método lança as exceções verificadas.
- Se lançar, o compilador assegura que a exceção verificada é capturada ou declarada em uma cláusula `throws`.
- A verificação é exigida como uma forma de forçar os programadores a pensar nos problemas que podem ocorrer quando um método que lança exceções verificadas for chamado.

Hierarquia de exceções

- Um erro de compilação ocorre se um método tentar explicitamente lançar uma exceção verificada e ela não estiver listada na cláusula `throws` do método.

Hierarquia de exceções

- Ao contrário das exceções verificadas, o compilador Java não verifica o código para determinar se uma exceção não verificada é capturada ou declarada. Em geral, pode-se impedir a ocorrência de exceções não verificadas pela codificação adequada.
- Por exemplo, a *ArithmeticException* não verificada lançada pelo método quotient pode ser evitada se o método assegurar que o denominador não é zero antes de realizar a divisão.

Hierarquia de exceções

- Várias classes de exceção podem ser derivadas de uma superclasse comum.

Bloco `finally`

- O bloco `finally` é opcional.
- A posição do bloco `finally` é depois do último bloco `catch`.
- Java garante que um bloco `finally` executará se uma exceção for lançada no bloco `try` correspondente ou quaisquer de seus blocos `catch` correspondentes.

Bloco `finally`

- Java também garante que um bloco `finally` executará se um bloco `try` fechar utilizando uma instrução `return`, `break` ou `continue`.
- O bloco `finally` não executará se o aplicativo fechar antes de um bloco `try` chamando o método `System.exit`.
- Como um bloco `finally` quase sempre é executado, ele em geral contém código de liberação de recursos.

Bloco `finally`

- Resumidamente, podemos dizer que as circunstâncias que podem evitar a execução do código em um bloco `finally` são:
  - Uma exceção surgindo no próprio bloco `finally`.
  - A morte da `thread`.
  - O uso de `System.exit( )`.
  - Desligar a energia da CPU.

Bloco `finally`

UsingExceptions.java

```
1  public class UsingExceptions {
2      public static void main (String[] args) {
3          try {
4              throwException();
5          }
6          catch (Exception exception) {
7              System.err.println("Exceção manipulada dentro do main");
8          }
9          doesNotThrowException();
10     }
11     public static void throwException() throws Exception {
12         try {
13             System.out.println("Método throwException");
14             throw new Exception();
15         }
16         catch (Exception exception) {
17             System.err.println("Exceção manipulada dentro do método throwException");
18             throw exception;
19         }
20         finally {
21             System.err.println("Bloco finally executado dentro do throwException");
22         }
23     }
24     public static void doesNotThrowException () {
25         try {
26             System.out.println("Método doesNotThrowException");
27         }
28         catch (Exception exception) {
29             System.err.println(exception);
30         }
31         finally {
32             System.err.println("Bloco finally executado dentro de doesNotThrowException");
33         }
34     }
```

Programa 3

printStackTrace, getStackTrace e getMessage

- Além do tratamento das exceções, Java possui um conjunto de métodos úteis para o processo de teste e depuração dos programas.
- O método printStackTrace envia para o fluxo de erro padrão o rastreamento da pilha.
- O método getStackTrace recupera informações sobre o rastreamento da pilha que podem ser impressas printStackTrace.
- O método getMessage retorna a string descritiva armazenada em uma exceção.

printStackTrace, getStackTrace e getMessage

Tracing.java

```
1  public class Tracing {
2      public static void main(String[] args) {
3          try {
4              method1();
5          }
6          catch (Exception exception) {
7              System.err.printf("%s\n\n", exception.getMessage());
8              exception.printStackTrace(); //imprime o rastreamento da pilha
9
10             StackTraceElement[] traceElements = exception.getStackTrace();
11             System.out.println("\nStack trace from getStackTrace:");
12             System.out.println("Class\t\tFile\t\t\tLine\tMethod");
13
14             for (StackTraceElement element : traceElements) {
15                 System.out.printf("%s\t", element.getClassName());
16                 System.out.printf("%s\t", element.getFileName());
17                 System.out.printf("%s\t", element.getLineNumber());
18                 System.out.printf("%s\n", element.getMethodName());
19             }
20         }
21     }
22
23     public static void method1() throws Exception{
24         method2();
25     }
26
27     public static void method2() throws Exception {
28         method3();
29     }
30
31     public static void method3() throws Exception {
32         throw new Exception("Exception throw in method3");
33     }
34 }
```

Novos tipos de exceção

- Apesar da maioria dos programadores utilizarem do tratamento de exceções da API Java, a linguagem permite que os programadores criem suas próprias classes para lançamento de exceções.
- Porém, por questão de padronização, deve-se utilizar as próprias classes de exceção existentes.

Novos tipos de exceção

- Uma nova classe de exceção deve estender uma classe de exceção existente para assegurar que a classe pode ser utilizada com o mecanismo de tratamento de exceções.
- Como qualquer outra classe, uma classe de exceção pode conter campos e métodos.
- Por convenção, todos os nomes de classes de exceções devem terminar com a palavra Exception.

Assertivas

- Ao implementar e depurar uma classe, às vezes é útil declarar as condições que devem ser verdadeiras em um ponto particular de um método.
- As assertivas ajudam a assegurar a validade de um programa capturando bugs potenciais e identificando possíveis erros de lógica durante o desenvolvimento.
- A instrução assert avalia uma expressão booleana e determina se ela é verdadeira ou falsa.

Assertivas

- A primeira forma das assertivas é:
  - expressao assert;
  - Essa instrução avaliação expressao e lança um AssertionError se a expressao for falsa.
- A segunda forma das assertivas é:
  - assert expressao1:expressao2;
  - Essa instrução avalia expressao1 e lança um AssertionErro com a expressao2 como a mensagem de erro, caso a expressao1 seja falsa.

Assertivas

AssertTest.java

```
1  import java.util.Scanner;
2
3  public class AssertTest {
4      public static void main(String[] args) {
5          Scanner input = new Scanner (System.in);
6
7          System.out.print("Digite um número entre 0 e 10:");
8          int number = input.nextInt();
9
10         assert(number >= 0 && number <=10) : "fora do intervalo " + number;
11
12         System.out.printf ("Você digitou %d\n", number);
13     }
14 }
15
```

Assertivas

- As assertivas são utilizadas principalmente pelo programador para depurar e identificar error de lógica em um aplicativo.
- Por padrão, as assertivas são desativadas ao executar um programa porque reduzem o desempenho e são desnecessárias ao usuário do programa.
- Para ativar as assertivas em tempo de execução, utilize a opção de linha de comando -ea com o comando java.
- java -ea AssertTest

Exemplos

*A.java

```
1  /** Neste exemplo, a classe A tem um metodo f() que pode lançar uma exceção do tipo
2  NumberFormatException, que e' nao verificada. Por esse motivo, o método f() não precisa
3  incluir a terminação "throws NumberFormatException". **/
4
5  public class A {
6
7      public static void main(String[] args) {
8          A obj = new A();
9          obj.f(10);
10     }
11
12     public void f(int a){
13         if (a<20) throw new NumberFormatException();
14         System.out.println("a = "+ a);
15         }
16 }
```

Exemplos

B.java

```
1  import java.io.IOException;
2
3
4  public class B {
5
6      public static void main (String[] args) {
7          B obj = new B();
8          obj.f(10);
9      }
10
11     public void f(int a) throws IOException {
12         if (a < 20)
13             throw new IOException("valor do argumento de f() e' " + a
14                     + " (menor que 20)");
15         System.out.println("a = " + a);
16     }
17 }
```

Exemplos

TesteExc1.java

```java
/* Neste exemplo, a exceção será capturada, e as três mensagens serão exibidas.
 * Ou seja, mesmo depois de finally executar, o restante do método main
 * continua. */
public class TesteExc1{
    public static void main(String[] args){
        try{
            A x = new A();
            int a = 4;
            x.f(a);
        }
        catch(Exception e){
            System.out.println("valor ilegal de a");
        }
        finally{
            System.out.println("fim do bloco try em TesteExc");
        }
        System.out.println("fim do metodo main em TesteExc");
    }
}
```

Exemplos

TesteExc2.java

```
1  /** Neste exemplo, o bloco catch não existe. Portanto, a exceção não será capturada, gerando
2  um stack trace. O bloco finally e' executado, mas não o que segue depois. **/
3  public class TesteExc2 {
4      public static void main(String[] args){
5          try{
6              A x = new A();
7              int a = 30;
8              x.f(a);
9          }
10         finally{
11             System.out.println("fim do bloco try em TesteExc");
12         }
13         System.out.println("fim do metodo main em TesteExc");
14     }
15 }
```

Exemplos

*TesteExc3.java

```
1  /** Neste exemplo, o bloco catch não existe, apenas o try e o finally.
2  Com esse valor de a, a exceção não será lançada.
3  Nesse caso, o código depois do bloco finally também será executado. **/
4  public class TesteExc3 {
5      public static void main(String[] args){
6          try{
7              A x = new A();
8              int a = 34;
9              x.f(a);
10         }
11         finally{
12             System.out.println("fim do bloco try em TesteExc");
13         }
14         System.out.println("fim do metodo main em TesteExc");
15     }
16 }
```

Exemplos

TesteExc4.java

```
1  /** Neste exemplo, como a exceção que pode ser lançada por f() e' não verificada,
2  o compilador não reclama por não haver a cláusula throws no cabeçalho de main.
3  Mas a exceção será lançada, originando um stack trace, e o método main()
4  não continuará após o ponto da chamada de f(). **/
5  public class TesteExc4 {
6      public static void main(String[] args){
7          A obj = new A();
8          int a = 4;
9          obj.f(a); // com esse valor, f() lancara' excecao
10         System.out.println("fim do metodo main em TesteExc");
11     }
12 }
```

Exemplos

TesteExc5.java

```
1  /** Neste exemplo, como a exceção que pode ser lançada por f() e' do tipo "não verificada", o
2  compilador não reclama do fato de main() não informar que pode lançar uma exceção, com
3  "throws NumberFormatException" ou "throws Exception".
4  Como nesse exemplo a exceção não será lançada, o método main irá até o final. **/
5  public class TesteExc5 {
6      public static void main(String[] args){
7          A obj = new A();
8          int a = 10;
9  //      com esse valor, f() nao lancará exceção
10         obj.f(a);
11         System.out.println("fim do metodo main em TesteExc");
12     }
13 }
```

Exemplos

TesteExc6.java

```java
/** Neste exemplo, usa-se a informação contida no objeto exceção para gerar a mensagem de
erro, pois o método f() da classe B cria exceções com uma mensagem informativa. **/
import java.io.IOException;
public class TesteExc6 {
    public static void main(String[] args){
        try{
            B x = new B();
            int a = 4;
            x.f(a);
        }
        catch(IOException e){
            System.out.println(e); // imprime toString(e)
        }
        finally {
            System.out.println("fim do bloco try em TesteExc!");
        }
        System.out.println("fim do metodo main em TesteExc");
    }
}
```

Exemplos

TesteExc7.java

```
1  /** Neste exemplo, o compilador reclama porque a exceção que pode ser lançada por f()
2  é do tipo "verificada" (IOException), e o método main() não tem a clausula "throws
3  IOException" **/
4  import java.io.IOException;
5  public class TesteExc7 {
6      public static void main(String[] args) {
7          B x = new B();
8          int a = 34;
9          try {
10             x.f(a);
11         } catch (IOException e) {
12             // TODO Auto-generated catch block
13             e.printStackTrace();
14         }
15         System.out.println("fim do metodo main em TesteExc");
16     }
17 }
```

Exemplos

TesteExc8.java

```
1  /** Neste exemplo, a exceção que pode ser lançada por f() e' verificada (IOException),
2  e o método main() tem a clausula "throws IOException", compilando OK. **/
3  import java.io.IOException;
4  public class TesteExc8{
5      public static void main(String[] args) {
6          B x = new B();
7          int a = 4;
8          x.f(a);
9          System.out.println("fim do metodo main em TesteExc");
10     }
11 }
```

Questionário

- Execute o Programa 1 e pare a sua execução através do Eclipse. Observe se ocorreu uma exceção. Caso tenha ocorrido, faça um tratamento adequado a essa exceção.
- A ocorrência de uma exceção em Java, obrigatoriamente, finaliza o programa? Justifique a sua resposta.
- O bloco finally é sempre executado? Justifique a sua resposta.

Questionário

- Qual é a diferença entre erros e exceções?
- Qual é a diferença entre os comandos `System.out.println` e `System.err.println`?
- Qual é a diferença entre as palavras-reservadas `throw` e `throws`?
- Se nenhuma exceção é lançada em um bloco try, onde o controle completa a execução?
- Um aplicativo convencional deve capturar objetos `Error`? Justifique a sua resposta.

Referências Bibliográficas

- http://lucabastos.blogspot.com/2007/06/tratamento-de-excees-parte-1-um-pouco.html
- http://java.sun.com/docs/books/tutorial/essential/exceptions/definition.html
- http://www.dcc.ufrj.br/~comp2/TextosJava/Excecoes.pdf


edit
delete


ARQUIVOS

Arquivos

- Os arquivos armazenam dados de forma permanente (dados persistentes) porque eles existem além da duração da execução do programa.

Hierarquia de dados

- Os caracteres em Java são caracteres Unicode compostos de dois bytes e cada byte é composto de oito bits.
- Assim como os caracteres são compostos de bits, campos são compostos de caracteres ou bytes.
- Um campo é um grupo de caracteres ou bytes que transmitem um significado.
- A hierarquia segue assim: bit, caracteres, campos, registros (classes) e arquivos.

Hierarquia de dados

- Para facilitar a recuperação de registros específicos de um arquivo, pelo menos um campo em cada registro é escolhido como uma chave de registro.
- Esse campo em geral é utilizado para pesquisar e classificar registros.
- Há muitas maneiras de organizar os registros de um arquivo. A mais comum é a seqüencial.

Arquivos e Fluxos

- Java vê cada arquivo como um **fluxo** seqüencial de *bytes*.
- Cada sistema operacional fornece um mecanismo para determinar o fim do arquivo.
- Os arquivos podem ser fluxos baseados em bytes (**arquivos binários**) ou em caracteres (**arquivos de texto**).
- Programas em Java realizam o processamento de arquivos utilizando as classes no pacote `java.io`.

Classe `File`

- Os objetos da classe `File` são utilizados frequentemente com objetos de outras classes `java.io` para especificar arquivos ou diretórios a manipular.
- A classe `File` possui quatro construtores:
  - `public File` (`String` nome);
    - nome → especifica o nome de um arquivo ou diretório para associar com o objeto `File`.
    - O nome pode conter informações de caminho, bem como um nome de arquivo ou diretório.
    - O caminho pode ser absoluto ou relativo.

Classe `File`

- `public File` (`String` caminho, `String` nome);
  - Utiliza argumento caminho para localizar o arquivo ou diretório especificado por nome.
- `public File` (`File` diretorio, `String` nome);
  - Utiliza o objeto diretorio para existente para localizar o arquivo ou diretório especificado por nome;
- `public File` (`URI` uri);
  - Utiliza o objeto uri para localizar o arquivo.
  - URI = Uniform Resource Identifier.

Classe `File`

- A classe `File` utiliza o método `isFile( )` para determinar se um objeto `File` representa um arquivo (não um diretório) antes de tentar abrir um arquivo.

Classe `File`

FileDemonstration.java | FileDemonstrationTeste.java

```
1  import java.io.File;
2
3  public class FileDemonstration {
4      public void analyzePath(String path) {
5          File name = new File(path);
6          if (name.exists()) {
7              System.out.printf("%s%s\n%s\n%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s",
8                      name.getName(), "exists", (name.isFile() ? "is a file"
9                              : "is not a file"),
10                     (name.isDirectory() ? "is a directory"
11                             : "is not a directory"),
12                     (name.isAbsolute() ? "is absolute path"
13                             : "is not absolute path"), "Last modified: ", name
14                             .lastModified(), "Length: ", name.length(),
15                     "Path: ", name.getPath(), "Absolute path: ", name
16                             .getAbsolutePath(), "Parent: ", name.getParent());
17             if (name.isDirectory()) {
18                 String directory[] = name.list();
19                 System.out.println("\n\nDirectory contents:\n");
20
21                 for (String directoryName : directory)
22                     System.out.printf("%s\n", directoryName);
23             }
24         } else {
25             System.out.printf("%s %s", path, "does not exist.");
26
27         }
28     }
29 }
```

FileDemonstration.java

Classe `File`

FileDemonstrationTeste.java

```
1  import java.util.Scanner;
2
3  public class FileDemonstrationTeste {
4      public static void main(String[] args) {
5          Scanner input = new Scanner (System.in);
6          FileDemonstration application = new FileDemonstration();
7
8          System.out.print("Enter file or directory name here.");
9          application.analyzePath(input.nextLine());
10     }
11 }
12
```

FileDemonstrationTeste.java

Classe `File`

- Um caracter separador é utilizado para separar diretórios e arquivos no caminho.
- No Windows, o separador é uma barra invertida (\), já no Unix o separador é uma barra normal (/).
- Porém, Java processa esses dois caracteres de maneira idêntica em um nome de caminho.
- Uma forma alternativa é usar o `File.pathSeparator` para obter o caractere separador adequado.

Classe `File`

- Utilizar \ como separador de diretório em vez de \\ em uma literal string é um erro de lógica.

Arquivos de texto de acesso seqüencial

- Java não impõe nenhuma estrutura a um arquivo.
- Os três programas seguintes demonstram como criar um arquivo simples de acesso sequencial que poderia ser utilizado em um sistema de contas a receber para ajudar a monitorar os valores devidos pelos seus clientes a uma empresa.
- O programa assume que o usuário insere os registros em ordem de número de conta.

Arquivos de texto de acesso seqüencial

AccountRecord.java

```
1  public class AccountRecord {
2      private int account;
3
4      private String firstName;
5
6      private String lastName;
7
8      private double balance;
9
10     public AccountRecord() {
11         this(0, "", "", 0.0);
12     }
13
14     public AccountRecord(int acct, String first, String last, double bal) {
15         setAccount(acct);
16         setFirstName(first);
17         setLastName(last);
18         setBalance(bal);
19     }
20
21     public void setAccount(int acct) {
22         account = acct;
23     }
24
25     public int getAccount() {
26         return account;
27     }
28
29     public void setFirstName(String first) {
30         firstName = first;
31     }
32
33     public String getFirstName() {
34         return firstName;
```

AccountRecord.java

Arquivos de texto de acesso seqüencial

CreateTextFile.java

```
1  import java.io.FileNotFoundException;
5
6  public class CreateTextFile {
7      private Formatter output;
8
9      public void openFile() {
10         try {
11             output = new Formatter ("clientes.txt");
12         }
13         catch (SecurityException securityException) {
14             System.err.println("Você não tem permissão de escrita no arquivo.");
15             System.exit(1);
16         }
17         catch (FileNotFoundException filesNotFoundException) {
18             System.err.println("Erro na criação do arquivo.");
19             System.exit(1);
20         }
21     }
22
23     public void addRecords () {
24         AccountRecord record = new AccountRecord ();
25         Scanner input = new Scanner (System.in);
26
27         System.out.printf("%s\n%s\n%s\n%s\n\n", "Para encerrar a entrada de dados, digite o comando para finalizar", "quando você for solicitado para
28
29         System.out.printf("%s\n%s", "Digite quantidade (n>0), primeiro nome, último nome e saldo.", "?");
30
31         while (input.hasNext()){
32             try {
33                 record.setAccount(input.nextInt());
34                 record.setFirstName(input.next());
35                 record.setLastName(input.next());
36                 record.setBalance(input.nextDouble());
37
```

CreateTextFile.java

Arquivos de texto de acesso seqüencial

CreateTextFileTest.java

```
1 public class CreateTextFileTest {
2     public static void main(String[] args) {
3         CreateTextFile application = new CreateTextFile();
4
5         application.openFile();
6         application.addRecords();
7         application.closeFile();
8     }
9 }
```

CreateTextFileTest.java

Arquivos de texto de acesso seqüencial

ReadTextFile.java

```
1  import java.io.File;
5
6  public class ReadTextFile {
7      private Scanner input;
8
9      public void openFile () {
10         try {
11             input = new Scanner(new File ("clientes.txt"));
12         }
13
14         catch (FileNotFoundException fileNotException) {
15             System.err.println("Erro na abertura do arquivo.");
16             System.exit(1);
17         }
18     }
19
20     public void readRecords () {
21         AccountRecord record = new AccountRecord();
22
23         System.out.printf("%-10s%-12s%-12s%10s\n", "Conta", "Primeiro Nome", "Último Nome", "Saldo");
24
25         try {
26             while (input.hasNext()) {
27                 record.setAccount(input.nextInt());
28                 record.setFirstName(input.next());
29                 record.setLastName(input.next());
30                 record.setBalance(input.nextDouble());
31
32                 System.out.printf("%-10d%-12s%-12s%10.2f\n", record.getAccount(), record.getFirstName(), record.getLastName(), record.getBalance());
33             }
34         }
35         catch (NoSuchElementException elementException) {
36             System.err.println("Falha no arquivo.");
37             input.close();
```

ReadTextFile.java

Arquivos de texto de acesso seqüencial

ReadTextFileTest.java

```
1 public class ReadTextFileTest {
2     public static void main(String[] args) {
3         ReadTextFile application = new ReadTextFile();
4         application.openFile();
5         application.readRecords();
6         application.closeFile();
7     }
8 }
```

ReadTextFileTest.java

Arquivos de texto de acesso seqüencial

- Os dados em muitos arquivos sequenciais não podem ser modificados sem o risco de destruir outros dados no arquivo.
- O problema é que a atualização do novo registro pode ser maior que o antigo registro.
- O problema aqui é que os campos em um arquivo de texto – e consequentemente os registros – podem variar de tamanho.

Arquivos de texto de acesso seqüencial

- Portanto, os registros em um arquivo de acesso sequencial normalmente não são atualizados no lugar.
- Em vez disso, o arquivo inteiro é normalmente regravado.

Serialização de objeto

- Quando sabemos exatamente a forma pela qual os dados são armazenados em um arquivo, podemos ler ou gravar um objeto, de um determinado tipo, a partir de um arquivo.
- Java possui esse tipo de recurso, chamado de serialização de objetos.
- Um objeto serializado é um objeto representado como uma sequência de bytes que inclui os dados do objeto, bem como as informações sobre o tipo do objeto e os tipos dos dados armazenados no objeto.

Arquivos de acesso aleatório

- Os arquivos de acesso sequencial não são apropriados para aplicativos em que a operação de busca seja importante.
- Exemplos: sistemas de reserva de passagem aérea, sistemas bancários, sistemas comerciais, caixas automáticos, entre outros.
- O acesso imediato pode ser realizado em arquivos de acesso aleatório e com banco de dados.
- Arquivos de acesso aleatório também são chamados de acesso direto.

Arquivos de acesso aleatório

- Para uso de arquivos de acesso aleatório é necessário especificar o formato desses arquivos.
- Várias técnicas podem ser utilizadas para criar arquivos de acesso aleatório.
- Talvez a mais simples de todas seja exigir que todos os registros em um arquivo tenham o mesmo comprimento fixo

Arquivos de acesso aleatório

- A utilização de registros de largura fixa ajuda o programa a calcular a localização exata de qualquer registro em relação ao início do arquivo.
- Um programa pode inserir dados em um arquivo de acesso aleatório sem destruir os dados nesse arquivo.
- Um `RandomAccessFile` é útil para aplicativos de acesso direto.

Arquivos de acesso aleatório

- Quando um programa associa um objeto da classe `RandomAccessFile` com um arquivo, ele lê ou grava os dados a partir do local no arquivo especificado pelo ponteiro de posição no arquivo e manipula todos os dados como tipos primitivos.
- Por exemplo, quando se grava um valor `int`, quatro bytes são enviados para o arquivo de saída.

Arquivos de acesso aleatório

- Programas de processamento de arquivos de acesso aleatório raramente gravam um único campo em um arquivo.
- Normalmente, eles gravam um objeto por vez, como mostraremos nos exemplos a seguir.

Arquivos de acesso aleatório

RandomAccessAccountRecord.java

```
1  import java.io.IOException;
3
4  public class RandomAccessAccountRecord extends AccountRecord {
5      public static final int SIZE = 72;
6
7      public RandomAccessAccountRecord () {
8          this(0,"","", 0.0);
9      }
10
11     public RandomAccessAccountRecord (int account, String firstName, String lastName, double balance) {
12         super(account, firstName, lastName, balance);
13     }
14
15     public void read(RandomAccessFile file) throws IOException {
16         setAccount (file.readInt());
17         setFirstName(readName(file));
18         setLastName(readName(file));
19         setBalance(file.readDouble());
20     }
21
22     private String readName(RandomAccessFile file) throws IOException {
23         char name[] = new char[15], temp;
24
25         for (int count = 0; count < name.length; count++) {
26             temp = file.readChar();
27             name[count] = temp;
28         }
29
30         return new String(name).replace('\0', ' ');
31     }
32
33     public void write(RandomAccessFile file) throws IOException {
34         file.writeInt(getAccount());
35         writeName(file, getFirstName());
```

RandomAccessAccountRecord.java

Arquivos de acesso aleatório

CreateRandomFile.java

```
1  import java.io.IOException;
3
4  public class CreateRandomFile {
5      private static final int NUMBER_RECORDS = 100;
6
7      public void createFile() {
8          RandomAccessFile file = null;
9
10         try {
11             file = new RandomAccessFile ("cliente.dat", "rw");
12             RandomAccessAccountRecord blankRecord = new RandomAccessAccountRecord();
13
14             for (int count = 0; count < NUMBER_RECORDS; count++)
15                 blankRecord.write(file);
16
17             System.out.println("Criou o arquivo cliente.dat");
18
19             System.exit(0);
20         }
21         catch(IOException ioException){
22             System.err.println("Erro ao processar o arquivo.");
23             System.exit(1);
24         }
25         finally {
26             try {
27                 if (file != null)
28                     file.close();
29             }
30             catch (IOException ioException) {
31                 System.err.println("Erro ao fechar o arquivo.");
32                 System.exit(1);
33             }
34         }
35     }
```

CreateRandomFile.java

Arquivos de acesso aleatório

CreateRandomFileTest.java

```
1 public class CreateRandomFileTest {
2     public static void main(String[] args){
3         CreateRandomFile application = new CreateRandomFile();
4         application.createFile();
5     }
6 }
7
```

CreateRandomFileTest.java

Arquivos de acesso aleatório

- O programa a seguir grava os dados em um arquivo que é aberto com um modo “rw” (para leitura e gravação).
- Ele utiliza o método `seek` de `RandomAccessFile` para posicionar a localização exata no arquivo em um registro das informações é armazenado.

Arquivos de acesso aleatório

- O método `seek` configura o ponteiro de posição no arquivo, como um local específico no arquivo relativo ao começo do arquivo, e o método `RandomAccessAccountRecord` `write` gera saída dados na posição atual no arquivo.
- O programa supõe que o usuário não insere números de conta duplicados.

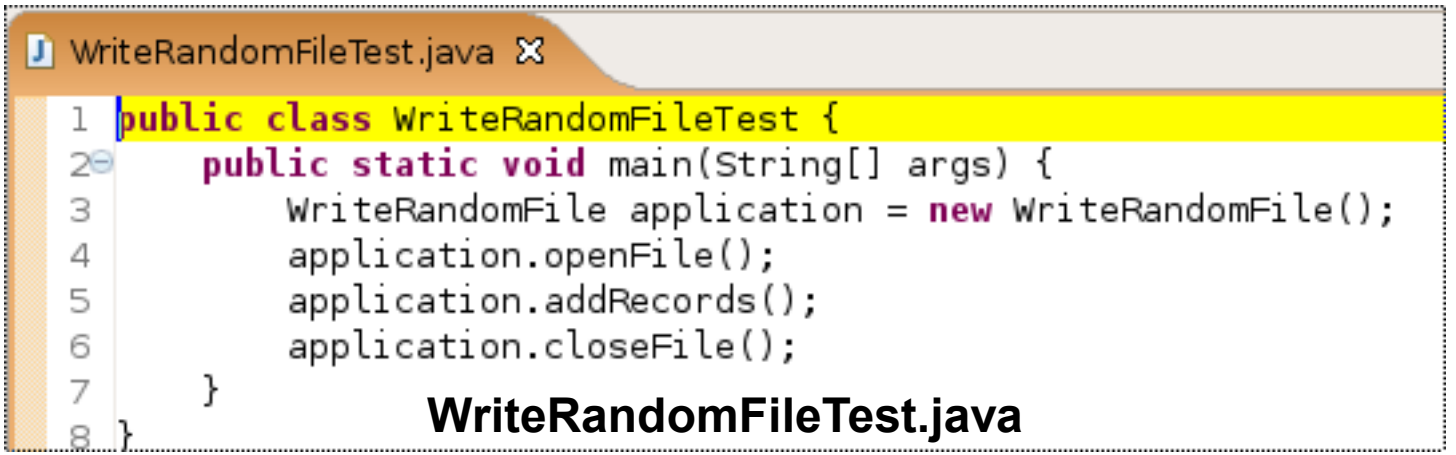
Arquivos de acesso aleatório

WriteRandomFile.java

```
1  import java.io.IOException;
5
6  public class WriteRandomFile {
7      private RandomAccessFile output;
8
9      private static final int NUMBER_RECORDS = 100;
10
11     public void openFile() {
12         try {
13             output = new RandomAccessFile("cliente.dat", "rw");
14         }
15         catch (IOException ioException) {
16             System.err.println("Arquivo não existe.");
17         }
18     }
19
20     public void closeFile() {
21         try {
22             if (output != null)
23                 output.close();
24         }
25         catch (IOException ioException) {
26             System.err.println("Erro ao fechar o arquivo.");
27             System.exit(1);
28         }
29     }
30
31     public void addRecords () {
32         RandomAccessAccountRecord record = new RandomAccessAccountRecord();
33
34         int accountNumber = 0;
35         String firstName;
36         String lastName;
37         double balance;
```

WriteRandomFile.java

Arquivos de acesso aleatório



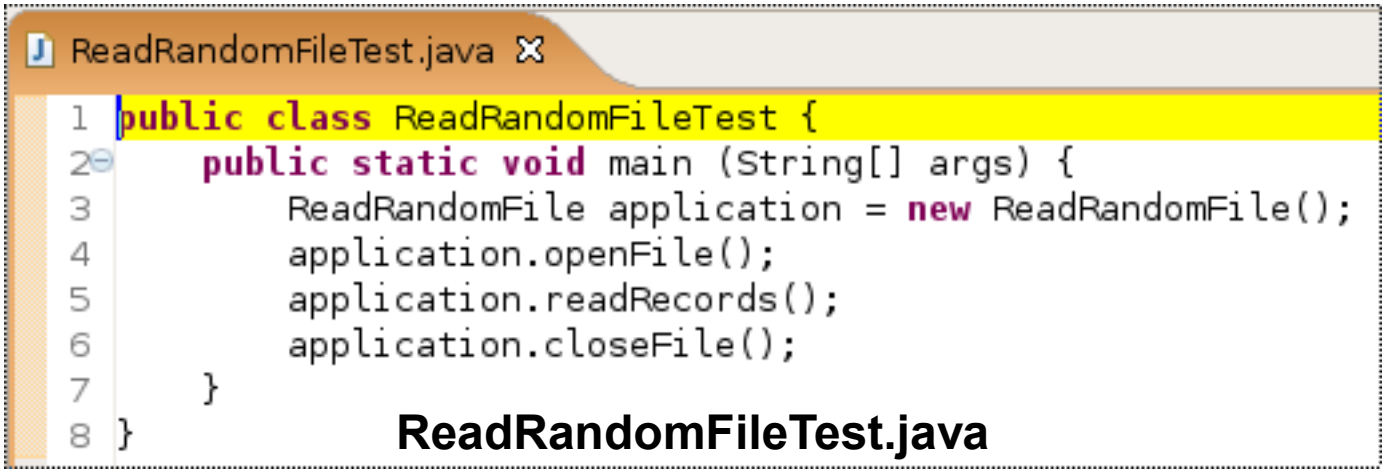
WriteRandomFileTest.java

Arquivos de acesso aleatório

ReadRandomFile.java

```java
import java.io.EOFException;
import java.io.IOException;
import java.io.RandomAccessFile;

public class ReadRandomFile {
    private RandomAccessFile input;

    public void openFile() {
        try {
            input = new RandomAccessFile("clients.dat", "r");
        }
        catch (IOException ioException) {
            System.err.println("Arquivo não existe.");
        }
    }

    public void readRecords() {
        RandomAccessAccountRecord record = new RandomAccessAccountRecord();

        System.out.printf("%-10s%-15s%-15s%10s\n", "Conta", "Primeiro nome", "Último nome", "Saldo");

        try {
            while(true) {
                do {
                    record.read(input);
                } while (record.getAccount() == 0);

                System.out.printf("%-10d%-12s%-12s%10.2f\n", record.getAccount(), record.getFirstName(), record.getLastName(), record.getBalance());
            }
        }
        catch (EOFException eofException) {
            return;
        }
        catch (IOException ioException) {
```

Arquivos de acesso aleatório



ReadRandomFileTest.java

Abrindo arquivos com `JFileChooser`

- Java possui a classe `JFileChooser` que permite exibir uma caixa de diálogo para que os usuários possam selecionar os arquivos mais facilmente.

Abrindo arquivos com `JFileChooser`

FileDemonstrationJFileChooser.java

```
1  import java.awt.BorderLayout;
2  import java.io.File;
3  import javax.swing.JFileChooser;
4  import javax.swing.JFrame;
5  import javax.swing.JOptionPane;
6  import javax.swing.JScrollPane;
7  import javax.swing.JTextArea;
8
9  public class FileDemonstrationJFileChooser extends JFrame {
10     private JTextArea outputArea;
11
12     private JScrollPane scrollPane;
13
14     public FileDemonstrationJFileChooser() {
15         super("Testando a classe File.");
16
17         outputArea = new JTextArea();
18
19         scrollPane = new JScrollPane(outputArea);
20
21         add(scrollPane, BorderLayout.CENTER);
22
23         setSize(400, 400);
24         setVisible(true);
25
26         analyzePath();
27     }
28
29     private File getFile() {
30         JFileChooser fileChooser = new JFileChooser();
31         fileChooser.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);
32
33         int result = fileChooser.showOpenDialog(this);
34
```

FileDemonstrationJFileChooser.java

Abrindo arquivos com `JFileChooser`

FileDemonstrationJFileChooserTest.java

```
1  import javax.swing.JFrame;
2
3  public class FileDemonstrationJFileChooserTest {
4      public static void main(String[] args) {
5          FileDemonstrationJFileChooser application = new FileDemonstrationJFileChooser();
6          application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7      }
8  }
```

FileDemonstrationJFileChooserTest.java

Questionário

- Qual é a diferença entre arquivo binário e arquivo texto?
- Explique o que é hierarquia de dados.
- O que é chave de registro?
- Escreva um programa em Java que faça a leitura de campos pelo teclado. Faça o tratamento adequado das possíveis exceções que possam ocorrer no programa.
- O que é um objeto serializado?

Questionário

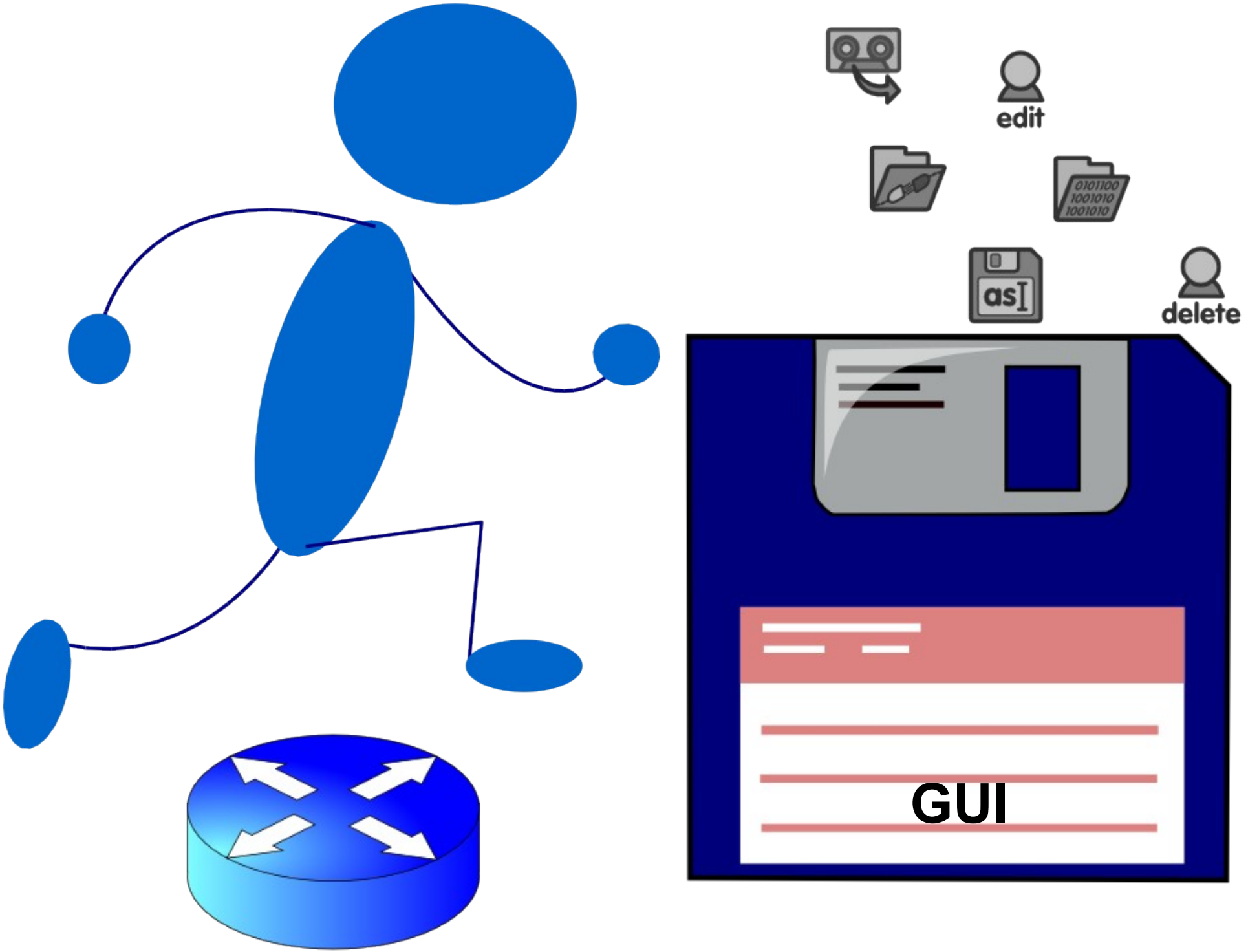
- Quais são as principais vantagens no uso de arquivos de acesso aleatório em relação aos arquivos de acesso seqüencial?
- Implemente um programa de console capaz de armazenar em um arquivo de formato binário 100 números inteiros, gerados aleatoriamente no intervalo [0,100.000] em um arquivo, cujo nome será fornecido como argumento da aplicação. A soma dos valores deverá ser exibida na tela (mas não gravada no arquivo).

Questionário

- Implemente um programa de console capaz de efetuar a leitura de um arquivo binário, cujo nome será fornecido com argumento do programa, contendo 100 valores inteiros. A soma dos valores recuperados deve ser exibida na tela.

Referências

- Deitel, H.M. & Deitel, P.J. **Java – Como Programar**. 6a. Edição. Bookman, 2007.
- Júnior, Peter J. **Java – Guia do Programador**. Novatec Editora, 2007.


edit
delete
GUI

Introdução

- Uma interface gráfica com o usuário (*Graphical User Interface* – GUI) apresenta um mecanismo amigável ao usuário para interagir com um aplicativo.
- As GUIs são construídas a partir de componentes GUI.
- Esses componentes são chamados de controles ou widgets.

Entrada/saída baseada em GUI simples com `JOptionPane`

- A maioria dos aplicativos utiliza caixas de diálogo para que o usuário possa interagir.
- A classe `JOptionPane` do Java (pacote `javax.swing`) fornece caixas de diálogo pré-empacotadas tanto para entrada como para saída de dados.
- Esses diálogos são exibidos invocando métodos estáticos `JOptionPane`.
- A seguir, um exemplo de programa de soma simples.

Entrada/saída baseada em GUI simples com `JOptionPane`

Addition.java

```
1  import javax.swing.JOptionPane;
2
3  public class Addition {
4      public static void main(String[] args) {
5          String firstNumber = JOptionPane.showInputDialog("Entre com o primeiro número:");
6          String secondNumber = JOptionPane.showInputDialog("Entre com o segundo número:");
7
8          int number1 = Integer.parseInt(firstNumber);
9          int number2 = Integer.parseInt(secondNumber);
10
11         int sum = number1 + number2;
12
13         JOptionPane.showMessageDialog(null, "A soma é " + sum, "Soma de dois inteiros", JOptionPane.PLAIN_MESSAGE);
14     }
15 }
16
```

Additon.java

Entrada/saída baseada em GUI simples com `JOptionPane`

- Constantes `JOptionPane static` para diálogos de mensagem:

Tipo de diálogo de mensagem	Ícone	Descrição
`ERROR_MESSAGE`		Um diálogo que indica um erro para o usuário.
`INFORMATION_MESSAGE`		Um diálogo com uma mensagem informativa para o usuário.
`WARNING_MESSAGE`		Um diálogo que adverte o usuário de um problema potencial.
`QUESTION_MESSAGE`		Um diálogo que impõe uma pergunta ao usuário. Normalmente, esse diálogo exige uma resposta, como clicar em um botão Yes ou No.
`PLAIN_MESSAGE`	Nenhum ícone	Um diálogo que contém uma mensagem, mas nenhum ícone.

Visão geral dos componentes Swing

- A maioria dos componentes Swing são componentes Java puro, ou seja, foram completamente escritos, manipulados e exibidos em Java.
- Eles fazem parte da *Java Foundation Classes* (JFC) – bibliotecas de Java para desenvolvimento de GUI para múltiplas plataformas.

Visão geral dos componentes Swing

- Alguns componentes básicos:

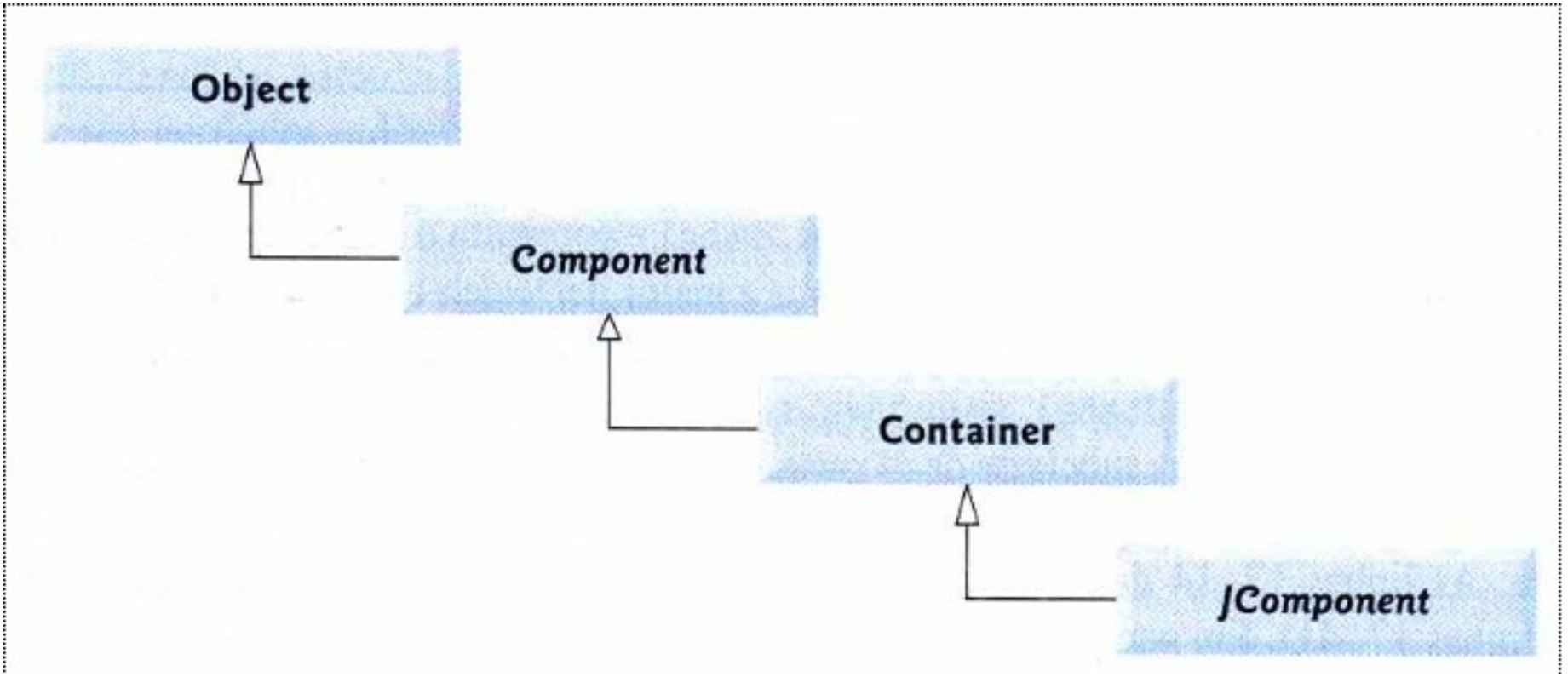
Componente	Descrição
`JLabel`	Exibe texto não editável ou ícones.
`JTextField`	Permite ao usuário inserir dados do teclado. Também pode ser utilizado para exibir texto editável ou não editável.
`JButton`	Desencadeia um evento quando o usuário clicar nele com o mouse.
`JCheckBox`	Especifica uma opção que pode ou não ser selecionada.
`JComboBox`	Fornece uma lista drop-down de itens a partir da qual o usuário pode fazer uma seleção clicando em um item ou possivelmente digitando na caixa.
`JList`	Fornece uma lista de itens a partir da qual o usuário pode fazer uma seleção clicando em qualquer item na lista. Múltiplos elementos podem ser selecionados.
`JPanel`	Fornece uma área em que os componentes podem ser colocados e organizados. Também pode ser utilizado como uma área de desenho para imagens gráficas.

Visão geral dos componentes Swing

- Swing versus AWT
  - Antes da criação do Swing, as GUIs em Java eram construídas com componentes do *Abstract Window Toolkit* (AWT).
  - Quando um aplicativo do AWT GUI é executado em diferentes plataformas Java, os componentes do aplicativo são exibidos de maneira diferente, adaptando-se ao sistema. ↑
  - Os componentes Swing são implementados em Java; desse modo são mais portáteis e flexíveis do que os componentes originais do AWT. ↑

Visão geral dos componentes Swing

- Superclasses comuns de muitos dos componentes do Swing.



Exibição de textos e imagens em uma janela

- A maioria das GUIs são construídas através de uma instância da classe `JFrame` ou como uma subclasse de `JFrame`.
- O componente `JLabel` é conhecido como rótulo e empregado para exibir uma única linha de texto de leitura, uma imagem ou tanto texto como imagem.
- Os aplicativos raramente alteram o conteúdo de um rótulo após a sua criação.
- O programa a seguir apresenta um exemplo de uso do `JLabel`.

Exibição de textos e imagens em uma janela

LabelFrame.java

```java
import java.awt.FlowLayout;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.SwingConstants;

public class LabelFrame extends JFrame{
    private JLabel label1;
    private JLabel label2;
    private JLabel label3;

    public LabelFrame() {
        super ("Testando o JLabel");
        setLayout(new FlowLayout());

        label1 = new JLabel("Label com texto");
        label1.setToolTipText("Este é o label1");
        add(label1);

        Icon bug = new ImageIcon(getClass().getResource("monkey.png"));
        label2 = new JLabel ("Label com texto e ícone", bug, SwingConstants.LEFT);
        label2.setToolTipText("Este é o label2");
        add(label2);

        label3 = new JLabel();
        label3.setText("Label com texto e ícone na parte inferior");
        label3.setIcon(bug);
        label3.setHorizontalTextPosition(SwingConstants.CENTER);
        label3.setVerticalTextPosition(SwingConstants.BOTTOM);
        label3.setToolTipText("Este é o label3");
        add(label3);
    }
}
```

LabelFrame.java

Exibição de textos e imagens em uma janela

LabelTest.java

```
1  import javax.swing.JFrame;
2
3  public class LabelTest {
4      public static void main(String[] args) {
5          LabelFrame labelFrame = new LabelFrame();
6          labelFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7          labelFrame.setSize(300, 350);
8  //         labelFrame.setSize(275, 180);
9          labelFrame.setVisible(true);
10     }
11 }
```

LabelTest.java

Exibição de textos e imagens em uma janela

- Ao construir uma GUI, cada componente deve ser anexado a um contêiner, como uma janela criada com um `JFrame`.
- Além disso, é necessário especificar onde posicionar cada componente da GUI. Isso é chamado de layout dos componentes GUI.
- Java fornece vários gerenciadores de layout que podem ajudar a posicionar os componentes.
- O gerenciador empregado é o `FlowLayout`.

Campos de texto e uma introdução ao tratamento de eventos com classes aninhadas

- As GUIs são baseadas em eventos.
- É através dos eventos que o usuário interage com o programa.
- O código que realiza uma tarefa em resposta a um evento é chamado de handler de evento.
- O processo total de responder a eventos é conhecido como tratamento de evento.

Campos de texto e uma introdução ao tratamento de eventos com classes aninhadas

- A seguir, serão apresentados dois novos componentes GUI que podem gerar eventos
  - `JTextFields`
  - `JPasswordFields`
- Cada um desses componentes é uma área de linha única em que o usuário pode inserir texto pelo teclado.

Campos de texto e uma introdução ao tratamento de eventos com classes aninhadas

TextFieldFrame.java

```
1  import java.awt.FlowLayout;
2  import java.awt.event.ActionEvent;
3  import java.awt.event.ActionListener;
4
5  import javax.swing.JFrame;
6  import javax.swing.JOptionPane;
7  import javax.swing.JPasswordField;
8  import javax.swing.JTextField;
9
10 public class TextFieldFrame extends JFrame{
11     private JTextField textField1;
12     private JTextField textField2;
13     private JTextField textField3;
14     private JPasswordField passwordField;
15
16     public TextFieldFrame() {
17         super("Testando JTextField e JPassordField");
18         setLayout(new FlowLayout());
19
20         textField1 = new JTextField(10);
21         add(textField1);
22
23         textField2 = new JTextField("Digite o texto aqui:");
24         add(textField2);
25
26         textField3 = new JTextField("Campo não editável", 21);
27         textField3.setEditable(false);
28         add(textField3);
29
30         passwordField = new JPasswordField("Texto oculto");
31         add(passwordField);
32
33         TextFieldHandler handler = new TextFieldHandler();
34         textField1.addActionListener(handler);
```

TextFieldFrame.java

Campos de texto e uma introdução ao tratamento de eventos com classes aninhadas

TextFieldTest.java

```
1  import javax.swing.JFrame;
2
3  public class TextFieldTest {
4      public static void main(String[] args) {
5          TextFieldFrame textFieldFrame = new TextFieldFrame();
6          textFieldFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7          textFieldFrame.setSize(325, 100);
8          textFieldFrame.setVisible(true);
9      }
10 }
11
```

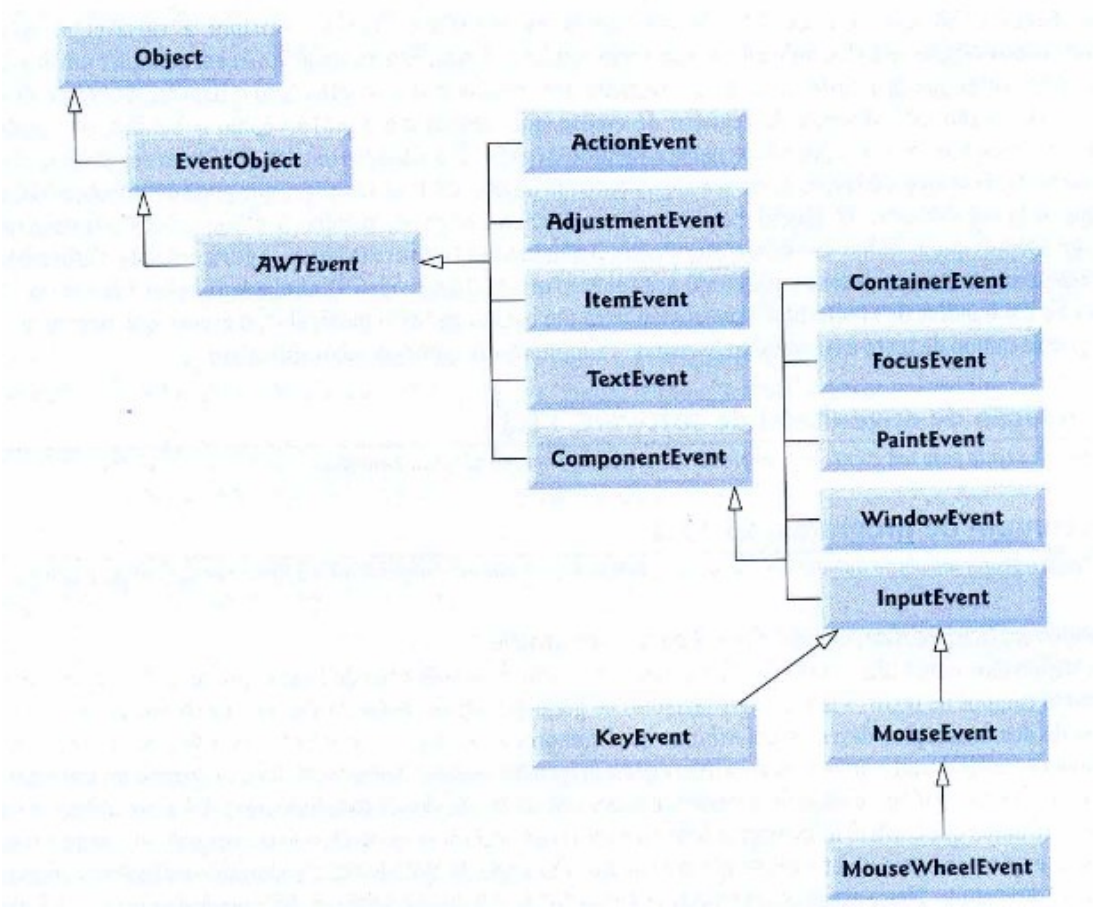
TextFieldTest.java

Tipos comuns de eventos GUI e interfaces ouvintes

- Como visto anteriormente, as informações sobre o evento que ocorre quando o usuário pressiona *Enter* em um campo de texto são armazenadas em um objeto `ActionEvent`.
- Há muitos tipos diferentes de eventos que podem ocorrer quando o usuário interage com uma GUI.
- As informações sobre qualquer evento são armazenadas em um objeto de uma classe que estende `AWTEvent`.

Tipos comuns de eventos GUI e interfaces ouvintes

- Algumas classes de evento do pacote `java.awt.event`.



Tipos comuns de eventos GUI e interfaces ouvintes

- O tratamento de eventos pode ser resumido em três partes: a origem do evento, o objeto do evento e o ouvinte do evento.
- A origem do evento é o componente GUI particular com o qual o usuário interage.
- O objeto de evento encapsula informações sobre o evento que ocorreu.
- O ouvinte de eventos é um objeto que é notificado pela origem de evento quando um evento ocorre.

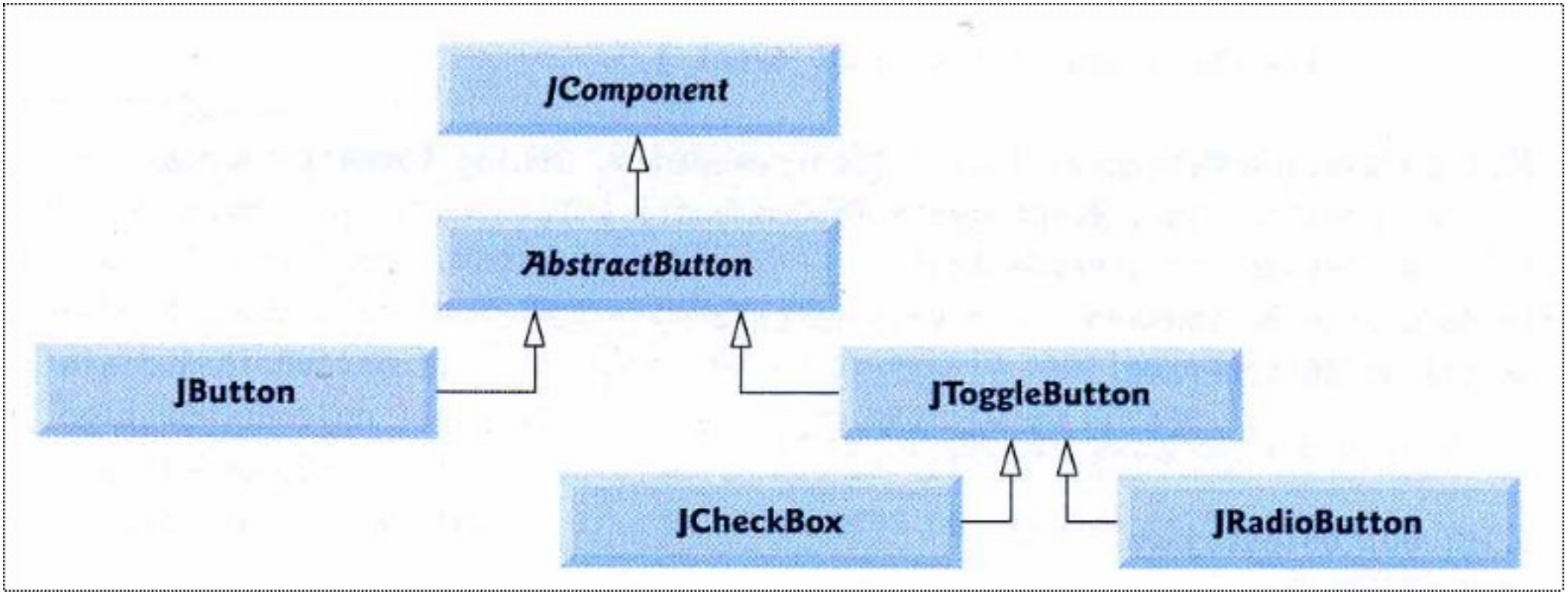
Como o tratamento de evento funciona?

JButton

- Um botão é um componente que o usuário clica para acionar uma ação específica.
- Java possui vários tipos de botões: botões de comando, caixas de seleção, botões de alternação e botões de opção.

JButton

- Hierarquia do botão Swing



JButton

ButtonFrame.java

```
1  import java.awt.FlowLayout;
2  import java.awt.event.ActionListener;
3  import java.awt.event.ActionEvent;
4  import javax.swing.Icon;
5  import javax.swing.ImageIcon;
6  import javax.swing.JButton;
7  import javax.swing.JFrame;
8  import javax.swing.JOptionPane;
9
10 public class ButtonFrame extends JFrame {
11     private JButton plainJButton; //botão apenas com texto
12     private JButton fancyJButton; //botão com ícones
13
14     public ButtonFrame () {
15         super ("Testando os botões Java.");
16
17         setLayout(new FlowLayout()); //configura o layout do frame
18
19         plainJButton = new JButton("Botão simples");
20         add(plainJButton); //adiciona o botão ao JFrame
21
22         Icon bug1 = new ImageIcon(getClass().getResource("monkey.png"));
23         Icon bug2 = new ImageIcon(getClass().getResource("monkey.png"));
24         fancyJButton = new JButton("Botão elaborado", bug1);
25         fancyJButton.setRolloverIcon(bug2);
26         add(fancyJButton);
27
28         // cria novo ButtonHandler para tratamento de evento de botão
29         ButtonHandler handler = new ButtonHandler();
30         fancyJButton.addActionListener(handler);
31         plainJButton.addActionListener(handler);
32     }
33
34     // classe interna para tratamento de evento de botão
```

ButtonFrame.java

JButton

ButtonTest.java

```
1  import javax.swing.JFrame;
2
3  public class ButtonTest {
4      public static void main(String[] args) {
5          ButtonFrame buttonFrame = new ButtonFrame();
6          buttonFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7          buttonFrame.setSize(275, 300);
8          buttonFrame.setVisible(true);
9      }
10 }
```

ButtonTest.java

Botões que mantêm o estado

- Os componentes Swing contêm três tipos de botões de estado - `JToggleButton`, `JCheckBox` e `JRadioButton` - que têm valores ativado/desativo ou verdadeiro/falso.
- As classes `JcheckBox` e `JradioButton` são subclasses de `JtoggleButton`.
- Um botão de comando gera um `ActionEvent` quando o usuário clica no botão.
- Os botões de comando são criados com a classe `JButton`.

Botões que mantêm o estado

- O texto na face de um `JButton` é chamado de rótulo de botão e cada botão deve possuir um rótulo diferente, de forma a evitar a ambiguidade.

Botões que mantêm o estado

ButtonFrame.java

Botões que mantêm o estado

- O próximo programa utiliza dois objetos `JCheckBox` para selecionar o estilo desejado de fonte do texto exibido em um `JTextField`.
- Quando selecionado, aplica o estilo negrito e o outro, o estilo itálico.
- Se ambos são selecionados, o estilo da fonte é negrito e itálico.
- Quando o aplicativo é inicialmente executado, nenhum `JCheckBox` está marcado (isto é, os dois são `false`).

Botões que mantêm o estado

CheckBoxFrame.java

Botões que mantêm o estado

CheckBoxText.java

Botões que mantêm o estado

ButtonTest.java

Botões que mantêm o estado

- Os botões de opção (declarados com a classe `JRadioButton`) são semelhantes a caixas de seleção no sentido de que têm dois estados (selecionados e não selecionados).
- Os botões de opção são utilizados para representar opções mutuamente exclusivas.

Botões que mantêm o estado

RadioButtonFrame.java

Botões que mantêm o estado

RadioButtonTest.java

`JList`

- Uma lista exibe uma série de itens a partir da qual o usuário pode selecionar um ou mais itens.
- As listas são criadas com a classe `JList`, que estende diretamente da classe `JComponent`.
- A classe `JList` suporta listas de uma única seleção ou listas de seleção múltipla.

`JList`

- O programa a seguir cria uma `JList` que contém 13 nomes de cor.
- Ao clicar em nome da cor na `JList`, um `ListSelectionEvent` ocorre e o aplicativo muda a cor de fundo da janela de aplicativo para a cor selecionada.

JList

ListFrame.java

JList

ListTest.java

Tratamento de evento de mouse

- Os eventos de mouse podem ser capturados por qualquer componente GUI que deriva de java.awt.Component.
- Os métodos de interfaces MouseListener e MouseMotionListener são resumidos na próxima tabela.

Tratamento de evento de mouse

Métodos de interface `MouseListener` e `MouseMotionListener`

Métodos de interface `MouseListener`

`public void mousePressed( MouseEvent event )`

Chamado quando um botão do mouse é pressionado enquanto o cursor do mouse estiver sobre um componente.

`public void mouseClicked( MouseEvent event )`

Chamado quando um botão do mouse é pressionado e liberado enquanto o cursor do mouse pairar sobre um componente. Esse evento é sempre precedido por uma chamada para `mousePressed`.

`public void mouseReleased( MouseEvent event )`

Chamado quando um botão do mouse é liberado depois de ser pressionado. Esse evento sempre é precedido por uma chamada para `mousePressed` e uma ou mais chamadas para `mouseDragged`.

`public void mouseEntered( MouseEvent event )`

Chamado quando o cursor do mouse entra nos limites de um componente.

`public void mouseExited( MouseEvent event )`

Chamado quando o cursor do mouse deixa os limites de um componente.

Métodos de interface `MouseMotionListener`

`public void mouseDragged( MouseEvent event )`

Chamado quando o botão do mouse é pressionado enquanto o cursor estiver sobre um componente e quando o mouse é movido enquanto o seu botão permanecer pressionado. Esse evento é sempre precedido por uma chamada para `mousePressed`. Todos os eventos de arrastar são enviados para o componente a partir do qual o usuário começou a arrastar o mouse.

`public void mouseMoved( MouseEvent event )`

Chamado quando o mouse é movido quando o cursor estiver sobre um componente. Todos os eventos de movimento são enviados para o componente sobre o qual o mouse atualmente está posicionado.

Tratamento de evento de mouse

MouseTrackerFrame.java

Tratamento de evento de mouse

MouseTracker.java

Gerenciadores de Layout

- Os gerenciadores de layout são fornecidos a fim de organizar componentes GUI em contêiner para apresentações.
- Esse recurso permite que os programadores se concentrem na aparência e comportamento básicos e deixa os gerenciadores de layout processar a maioria dos detalhes de layout.
- Todos os gerenciadores de layout implementam a interface `LayoutManager` (no pacote `java.awt`).

Gerenciadores de Layout

- Há três maneiras básicas de organizar componentes em uma GUI:
  - Posicionamento absoluto.
    - Fornece o maior nível de controle sobre a aparência de uma GUI.
    - Basta configurar o `Container` como `null`, aí pode-se especificar a posição absoluta de cada componente GUI em relação ao canto superior esquerdo do `Container`.
    - Esta opção pode ser mais demorada que as demais.

Gerenciadores de Layout

- Há três maneiras básicas de organizar componentes em uma GUI:
  - Gerenciadores de layout.
    - Utilizar gerenciadores de layout pode ser mais simples e rápido do que o posicionamento absoluto, porém o programador perde algum controle sobre o tamanho e o posicionamento precisos de componentes GUI.

Gerenciadores de Layout

- Há três maneiras básicas de organizar componentes em uma GUI:
  - Programação visual em IDE.
    - Diversos IDEs fornecem ferramentas que facilitam a criação de GUIs.
    - O Eclipse não é uma boa ferramenta para isso.
    - A sugestão seria o IDE Netbeans.

Gerenciadores de Layout

- Principais gerenciadores de layout:

Gerenciador de layout	Descrição
`FlowLayout`	Padrão para `javax.swing.JPanel`. Coloca os componentes seqüencialmente (da esquerda para a direita) na ordem que foram adicionados. Também é possível especificar a ordem dos componentes utilizando o método `Container` `add`, que aceita um `Component` e uma posição de índice do tipo inteiro como argumentos.
`BorderLayout`	Padrão para `JFrames` (e outras janelas). Organiza os componentes em cinco áreas: `NORTH`, `SOUTH`, `EAST`, `WEST` e `CENTER`.
`GridLayout`	Organiza os componentes nas linhas e colunas.

Questionário

- No desenvolvimento das GUIs, aparência e comportamento têm o mesmo significado? Justifique a sua resposta.
- O que são *widgets*?
- O que você entende por componentes leves e pesados?
- O que fornece a classe `JFrame`?

Referências

- Deitel, H.M. & Deitel, P.J. **Java – Como Programar**. 6a. Edição. Bookman, 2007.
- Júnior, Peter J. **Java – Guia do Programador**. Novatec Editora, 2007.