CAP 2022

JAVA – CAP 2022

Conteúdo do Edital:

- Linguagem de programação JAVA;
- Bibliotecas de classe do Java;
- Classes e Objetos;
- Instruções de controle;
- Módulos de programa em Java;
- Arrays e Arraylists;
- Programação orientada a objetos;
- Tratamento de exceções;
- Componentes GUI;
- Strings, caracteres e expressões regulares;
- Recursão;
- Applets e Java Web Start;
- Multithreading; e
- Serviços Web.

Livro: DEITEL, Paul; DEITEL, Harvey. JAVA como Programar. 10.ed. [S.l.]: Pearson Prentice Hall, 2016.

JAVA – CAP 2022

Conteúdo Bibliográfico	Capítulo do Livro
Bibliotecas de classe do Java;	Capítulo 2
Classes e Objetos;	Capítulo 3, 6 e 8
Instruções de controle;	Capítulos 4 e 5
Módulos de programa em Java;	Capítulo 2
Arrays e Arraylists;	Capítulo 7
Programação orientada a objetos;	Capítulos 9 e 10
Tratamento de exceções;	Capítulo 11
Componentes GUI;	Capítulo 14
Strings, caracteres e expressões regulares;	Capítulo 16
Recursão;	Capítulo 18
Applets e Java Web Start;	Capítulo 23
Multithreading;	Capítulo 26
Serviços Web.	Capítulo 31

Aula 7 – JAVA

- Strings, caracteres e expressões regulares;

14.2 Fundamentos de caracteres e strings

Os caracteres são os blocos de construção fundamentais dos programas-fonte do Java. Cada programa é composto de uma sequência de caracteres que, quando agrupadas entre si significativamente, são interpretadas pelo compilador Java como uma série de instruções utilizadas para realizar uma tarefa. Um programa pode conter **literais de caractere**. Um literal de caractere é um valor inteiro representado como caractere entre aspas simples. Por exemplo, `'z'` representa o valor inteiro de `z` e `'\t'` representa o valor inteiro de caractere de tabulação. O valor de um literal de caractere é o valor inteiro do caractere no **conjunto de caracteres Unicode**. O Apêndice B apresenta os equivalentes inteiros dos caracteres no conjunto de caracteres ASCII, que é um subconjunto de Unicode (discutido no Apêndice H, em inglês, na Sala Virtual do livro).

A partir da discussão da Seção 2.2, lembre-se de que uma string é uma sequência de caracteres tratada como uma única unidade. Uma string pode incluir letras, dígitos e vários **caracteres especiais**, como `+`,`-`, `*`, `/` e `$`. Uma string é um objeto de classe `String`. As **literais string** (armazenadas na memória como objetos `String`) são escritas como uma sequência de caracteres entre aspas duplas, como em:

```
"John Q. Doe"                (um nome)
"9999 Main Street"           (a rua de um endereço)
"Waltham, Massachusetts"     (uma cidade ou estado)
"(201) 555-1212"             (um número de telefone)
```

Uma string pode ser atribuída a uma referência `String`. A declaração

```
String color = "blue";
```

inicializa a variável `String color` para referir-se a um objeto `String` que contém a string `"blue"`.

Dica de desempenho 14.1

Para economizar a memória, o Java trata todos os literais string com o mesmo conteúdo de um único objeto `String` que tem muitas referências a ele.

14.3 Classe String

A classe `String` é utilizada para representar strings em Java. As próximas várias subseções discutem muitas das capacidades da classe `String`.

14.3.1 Construtores String

A classe `String` fornece construtores para inicializar objetos `String` de uma variedade de maneiras. Quatro dos construtores são demonstrados no método `main` da Figura 14.1.

A linha 12 instancia um novo objeto `String` utilizando o construtor sem argumento da classe `String` e atribui sua referência a `s1`. O novo objeto `String` não contém caracteres (isto é, a **string vazia**, que também pode ser representada como `""`) e tem um comprimento de 0. A linha 13 instancia um novo objeto `String` usando o construtor da classe `String` que recebe um objeto `String` como um argumento e atribui sua referência a `s2`. O novo objeto `String` contém a mesma sequência de caracteres do objeto `String` `s` que é passado como um argumento ao construtor.

Dica de desempenho 14.2

Não é necessário copiar um objeto `String` existente. Objetos `String` são imutáveis, porque a classe `String` não fornece métodos que permitem que o conteúdo de um objeto `String` seja modificado depois que ele é criado.

A linha 14 instancia um novo objeto `String` e atribui sua referência à classe `s3`, que utiliza o construtor de `String` que aceita um array `char` como um argumento. O novo objeto `String` contém uma cópia dos caracteres no array.

A linha 15 instancia um novo objeto `String` e atribui sua referência à classe `s4` utilizando o construtor de `String` que aceita um array de `chars` e dois inteiros como argumentos. O segundo argumento especifica a posição inicial (o *deslocamento*) a partir da qual os caracteres no array são acessados. Lembre-se de que o primeiro caractere está na posição 0. O terceiro argumento especifica o número de caracteres (a contagem) a acessar no array. O novo objeto `String` é formado a partir dos caracteres acessados. Se o deslocamento ou a contagem especificada como um argumento resultar no acesso a um elemento fora dos limites do array de caracteres, uma `StringIndexOutOfBoundsException` é lançada.

```
1  // Figura 14.1: StringConstructors.java
2  // construtores da classe String.
3
4  public class StringConstructors
5  {
6     public static void main(String[] args)
7     {
8        char[] charArray = {'b', 'i', 'r', 't', 'h', ' ', 'd', 'a', 'y'};
9        String s = new String("hello");
10
11       // utiliza os construtores String
12       String s1 = new String();
13       String s2 = new String(s);
14       String s3 = new String(charArray);
15       String s4 = new String(charArray, 6, 3);
16
17       System.out.printf(
18          "s1 = %s%ns2 = %s%ns3 = %s%ns4 = %s%n", s1, s2, s3, s4);
19    }
20 } // fim da classe StringConstructors
```

```
s1 =
s2 = hello
s3 = birth day
s4 = day
```

14.3.2 Métodos String length, charAt e GetChars

Os métodos `length`, `charAt` e `getChars` de `String` retornam o comprimento de uma `String`, obtêm o caractere em uma localização específica em uma `String` e recuperam um conjunto de caracteres de uma `String` como um array `char`, respectivamente.

```
1  // Figura 14.2: StringMiscellaneous.java
2  // Esse aplicativo demonstra os métodos da classe String
3  // length, charAt e getChars.
4
5  public class StringMiscellaneous
6  {
7     public static void main(String[] args)
8     {
9        String s1 = "hello there";
10       char[] charArray = new char[5];
11
12       System.out.printf("s1: %s", s1);
13
14       // testa o método length
15       System.out.printf("%nLength of s1: %d", s1.length());
16
17       // faz loop pelos caracteres em s1 com charAt e os exibe na ordem inversa
18       System.out.printf("%nThe string reversed is: ");
19
20       for (int count = s1.length() - 1; count >= 0; count--)
21          System.out.printf("%c ", s1.charAt(count));
22
23       // copia caracteres a partir de string para charArray
24       s1.getChars(0, 5, charArray, 0);
25       System.out.printf("%nThe character array is: ");
26
27       for (char character : charArray)
28          System.out.print(character);
29
30       System.out.println();
31    }
32 } // fim da classe StringMiscellaneous
```

```
s1: hello there
Length of s1: 11
The string reversed is: e r e h t   o l l e h
The character array is: hello
```

Hello there

0 1 2 3 4 5 6 7 8 9 10

- **public String[ ] split(String regex):**

recebe como parâmetro um *String* cuja expressão será a regra de separação da frase *String* ao qual o método *split* foi aplicado, retornando um *array* de *Strings*, onde cada posição do *array* é o fragmento de frase resultante da divisão da frase original, separado pela expressão.

- **public int compareTo(String outraString):**

compara os caracteres *unicode* de duas *Strings* retornando 0 (zero) se as *Strings* forem iguais e diferente de 0 (zero) se forem diferentes.

- **public int compareToIgnoreCase(String outraString):**

✓ é um método bem interessante principalmente para utilização em pesquisas de nomes em cadastros e aplicações correlatas. Ele compara duas *String*, assim como no método *compareTo*, mas ignorando maiúsculas e minúsculas.

- **public String toUpperCase( ):**

✓ é um método muito conhecido das outras linguagens de programação e em Java executa a mesma operação, ou seja, converte todos os caracteres de uma *String* para maiúsculos.

- **public String toLowerCase( ):**

✓ é um método muito conhecido das outras linguagens de programação e em Java executa a mesma
✓ operação, ou seja, converte todos os caracteres de uma *String* para minúsculos.

- **public char charAt(int index):**

✓ método da classe String que recebe um número inteiro e retorna o caractere naquela posição da *String*.
✓ Vale lembrar que os índices dos caracteres de uma *String* se iniciam em 0 (zero).

- **public int length( ):**

✓ retorna o comprimento de uma *String*, que é igual ao seus número de caracteres. Lembre-se de que
✓ espaço vale como caracter de uma *String*. Além disso, cuidado para não confundir o índice de um caracter
✓ numa *String* começa com 0 (zero), mas o tamanho de uma *String* com um único caractere é 1 (um), mesmo
✓ que ele seja somente um espaço.

- **public String substring(int beginIndex):**

✓ método que recebe um número inteiro que representa o índice do caractere inicial do trecho a ser dividido e
✓ retorna uma *String* que é um fragmento da *String* original a partir do índice inserido como parâmetro, inclusive
✓ este caractere.

- **public String trim( ):**

✓ método que retorna a *String* original retirados os espaços em branco no início e no fim de uma sequência
✓ de caracteres. Método muito interessante de ser utilizado no caso de captura de dados por meio de um
✓ formulário, em que o usuário pode equivocadamente inserir espaços antes ou depois de um campo.

- **public int indexOf(char ch):**

✓ método que recebe um caracter de uma *String* e retorna o índice (número inteiro) correspondente à
✓ posição da primeira incidência daquele caracter no *String*.

- **public int lastIndexOf(char ch):**

✓ método que recebe um caracter de uma *String* e retorna o índice (número inteiro) correspondente à
✓ posição da última incidência daquele caractere no *String*.

- **public boolean isEmpty( ):**

✓ método que retorna *true* caso o comprimento da *String* seja 0 (zero) e *false* caso contrário.

- **public boolean contains(SequenceChar s):**

✓ método que retorna *true* caso a *String* contenha a sequência de caracteres inserida como parâmetro do
✓ método. É um método para fazer uma busca de uma substring dentro de uma *String*.

- **public String replace(char oldChar, char newChar):**

✓ método que retorna a *String* original substituindo o caracter *oldChar* pelo caracter *newChar* passados como
✓ parâmetros do método.

Tratamento de exceções: processando a resposta incorreta

Uma exceção indica um problema que ocorre quando um programa é executado. O nome “exceção” sugere que o problema ocorre com pouca frequência — se a “regra” é que uma instrução normalmente executa corretamente, então o problema representa a “exceção à regra”. O tratamento de exceção ajuda a criar programas tolerantes a falhas que podem resolver (ou tratar) exceções. Em muitos casos, isso permite que um programa continue a executar como se nenhum problema fosse encontrado.
Quando a Java Virtual Machine ou um método detecta um problema, como um índice de array inválido ou um argumento de método inválido, ele lança uma exceção.

A instrução try

Para lidar com uma exceção, coloque qualquer código que possa lançar uma exceção em uma instrução try.
O bloco try contém o código que pode lançar uma exceção, e o bloco catch (linhas 21 a 26) contém o código que trata a exceção se uma ocorrer. Podem haver muitos blocos catch para tratar com diferentes tipos de exceções que podem ser lançadas no bloco try correspondente. As chaves que delimitam o corpo dos blocos try e catch são obrigatórias.

Tratamento de exceções: processando a resposta incorreta

Executando o bloco catch

Quando o programa encontra o valor inválido no array, ele tenta dizer que está fora dos limites do array . Como a verificação dos limites de array é executada em tempo de execução, a JVM gera uma exceção — especificamente a linha e lança ArrayIndexOutOfBoundsException para notificar o programa sobre esse problema. Nesse ponto, o bloco try termina e o bloco catch começa a executar — se você declarou quaisquer variáveis locais no bloco try, agora elas estarão fora do escopo (e não mais existirão), assim elas não estarão acessíveis no bloco catch. O bloco catch declara um parâmetro de exceção (e) do tipo IndexOutOfRangeException. O bloco catch pode lidar com exceções do tipo especificado. Dentro do bloco catch, você pode usar o identificador do parâmetro para interagir com um objeto que capturou a exceção.

Dica de prevenção de erro 7.1

Ao escrever o código para acessar um elemento do array, certifique-se de que o índice de array permanece maior ou igual a 0 e menor que o comprimento do array. Isso evitaria `ArrayIndexOutOfBoundsException` *se seu programa estiver correto.*

JAVA – CAP 2022

Tratamento de exceções: processando a resposta incorreta

11.2 Exemplo: divisão por zero sem tratamento de exceção

Primeiro demonstramos o que acontece quando surgem erros em um aplicativo que não utiliza tratamento de exceção. A Figura 11.2 solicita ao usuário dois inteiros e os passa para o método `quotient`, que calcula o quociente inteiro e retorna um resultado `int`. Nesse exemplo, veremos que as exceções são **lançadas** (isto é, a exceção ocorre) quando um método detecta um problema e é incapaz de tratá-lo.

```
1  // Figura 11.2: DivideByZeroNoExceptionHandling.java
2  // Divisão de inteiro sem tratamento de exceção.
3  import java.util.Scanner;
4
5  public class DivideByZeroNoExceptionHandling
6  {
7     // demonstra o lançamento de uma exceção quando ocorre uma divisão por zero
8     public static int quotient(int numerator, int denominator)
9     {
10       return numerator / denominator; // possível divisão por zero
11    }
12
13    public static void main(String[] args)
14    {
15       Scanner scanner = new Scanner(System.in);
16
17       System.out.print("Please enter an integer numerator: ");
18       int numerator = scanner.nextInt();
19       System.out.print("Please enter an integer denominator: ");
20       int denominator = scanner.nextInt();
21
22       int result = quotient(numerator, denominator);
23       System.out.printf(
24          "%nResult: %d / %d = %d%n", numerator, denominator, result);
25    }
26 } // fim da classe DivideByZeroNoExceptionHandling
```

JAVA – CAP 2022

Tratamento de exceções: processando a resposta incorreta

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
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)
```

continua

continuação

```
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)
```

Figura 11.2 | Divisão de inteiro sem tratamento de exceção.

Rastreamento de pilha

A primeira execução de exemplo na Figura 11.2 mostra uma divisão bem-sucedida. Na segunda execução, o usuário insere o valor `0` como o denominador. Várias linhas de informação são exibidas em resposta a essa entrada inválida. Essas informações são conhecidas como **rastreamento de pilha**, que incluem o nome da exceção (`java.lang.ArithmeticException`) em uma mensagem descritiva que indica o problema que ocorreu e a pilha de chamadas de método (isto é, a cadeia de chamadas) no momento em que ela ocorreu. O rastreamento de pilha inclui o caminho de execução que resultou na exceção método por método. Isso ajuda a depurar o programa.

Rastreamento de pilha para uma `ArithmeticException`

A primeira linha especifica a ocorrência de uma `ArithmeticException`. O texto depois do nome da exceção ("`/by zero`") indica que essa exceção ocorreu como resultado de uma tentativa de dividir por zero. O Java não permite divisão por zero na aritmética de inteiros. Quando isso ocorre, o Java lança uma `ArithmeticException`. `ArithmeticExceptions` podem surgir a partir de alguns diferentes problemas, assim os dados adicionais ("`/by zero`") fornecem informações mais específicas. O Java *realmente* permite a divisão por zero com valores de ponto flutuante. Um cálculo como esse resulta no valor positivo ou negativo infinito, que é representado em Java como um valor de ponto flutuante (mas aparece como a string `Infinity` ou `-Infinity`). Se 0.0 for dividido por 0.0, o resultado é `NaN` (não um número), que também é representado no Java como um valor de ponto flutuante (mas é exibido como `NaN`). Se você precisa comparar um valor de ponto flutuante com `NaN`, use o método `isNaN` de classe `Float` (para obter valores `Float`) ou a classe `Double` (para obter valores `Double`). As classes `Float` e `Double` estão no pacote `java.lang`.

Iniciando da última linha do rastreamento de pilha, vemos que a exceção foi detectada na linha 22 do método `main`. Cada linha do rastreamento de pilha contém o nome de classe e o método (por exemplo, `DivideByZeroNoExceptionHandling.main`) seguido pelo nome de arquivo e número da linha (por exemplo, `DivideByZeroNoExceptionHandling.java:22`). Subindo o rastreamento de pilha, vemos que a exceção ocorre na linha 10, no método `quotient`. A linha superior da cadeia de chamadas indica o **ponto de lançamento** — o ponto inicial em que a exceção ocorreu. O ponto de lançamento dessa exceção está na linha 10 do método `quotient`.

Rastreamento de pilha para uma `InputMismatchException`

Na terceira execução, o usuário insere a string `"hello"` como o denominador. Observe novamente que um rastreamento de pilha é exibido. Isso informa a ocorrência de uma `InputMismatchException` (pacote `java.util`). Nossos exemplos anteriores que inseriram os valores numéricos supunham que o usuário inseriria um valor inteiro adequado. Entretanto, às vezes, os usuários cometem erros e inserem valores não inteiros. Uma `InputMismatchException` ocorre quando o método `Scanner` `nextInt` recebe uma `string` que não representa um inteiro válido. Iniciando do fim do rastreamento de pilha, vemos que a exceção foi detectada na linha 20 do método `main`. Subindo o rastreamento de pilha, vemos que a exceção ocorreu no método `nextInt`. Observe que no lugar do nome de arquivo e número da linha, o texto `Unknown Source` é fornecido. Isso significa que os chamados *símbolos de depuração* que fornecem informações sobre o nome de arquivo e número de linha para a classe desse método não estavam disponíveis para a JVM — esse é tipicamente o caso para as classes da API Java. Muitos IDEs têm acesso ao código-fonte da API Java e exibirão os nomes de arquivo e números de linha em rastreamentos de pilha.

Término do programa

Nas execuções de exemplo da Figura 11.2, quando exceções ocorrem e rastreamentos de pilha são exibidos, o programa também se *fecha*. Isso nem sempre ocorre no Java. Às vezes um programa pode continuar mesmo que uma exceção tenha ocorrido e um rastreamento de pilha tenha sido impresso. Nesses casos, o aplicativo pode produzir resultados inesperados. Por exemplo, um aplicativo com interface gráfica do usuário (GUI) muitas vezes continuará a executar. Na Figura 11.2 os dois tipos de exceção foram detectados no método `main`. No próximo exemplo, veremos como *tratar* essas exceções de modo que você possa permitir que o programa conclua sua execução normalmente.

JAVA – CAP 2022

sposta incorreta

```
3  import java.util.InputMismatchException;
4  import java.util.Scanner;
5
6  public class DivideByZeroWithExceptionHandling
7  {
8     // demonstra o lançamento de uma exceção quando ocorre uma divisão por zero
9     public static int quotient(int numerator, int denominator)
10       throws ArithmeticException
11    {
12       return numerator / denominator; // possível divisão por zero
13    }
14
15    public static void main(String[] args)
16    {
17       Scanner scanner = new Scanner(System.in);
18       boolean continueLoop = true; // determina se mais entradas são necessárias
19
20       do
21       {
22          try // lê dois números e calcula o quociente
23          {
24             System.out.print("Please enter an integer numerator: ");
25             int numerator = scanner.nextInt();
26             System.out.print("Please enter an integer denominator: ");
27             int denominator = scanner.nextInt();
28
29             int result = quotient(numerator, denominator);
30             System.out.printf("%nResult: %d / %d = %d%n", numerator,
31                denominator, result);
32             continueLoop = false; // entrada bem-sucedida; fim do loop
33          }
34          catch (InputMismatchException inputMismatchException)
35          {
36             System.err.printf("%nException: %s%n",
37                inputMismatchException);
38             scanner.nextLine(); // descarta entrada para o usuário tentar de novo
39             System.out.printf(
40                "You must enter integers. Please try again.%n%n");
41          }
42          catch (ArithmeticException arithmeticException)
43          {
44             System.err.printf("%nException: %s%n", arithmeticException);
45             System.out.printf(
46                "Zero is an invalid denominator. Please try again.%n%n");
47          }
48       } while (continueLoop);
49    }
50 } // fim da classe DivideByZeroWithExceptionHandling
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14

Please enter an integer numerator: 100
Please enter an integer denominator: 0

Exception: java.lang.ArithmeticException: / by zero
Zero is an invalid denominator. Please try again.
```

continuação

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello

Exception: java.util.InputMismatchException
You must enter integers. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

Figura 11.3 | Tratando `ArithmeticExceptions` e `InputMismatchExceptions`.

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7
```

Incluindo código em um bloco `try`

As linhas 22 a 33 contêm um **bloco `try`**, que inclui o código que *pode* lançar (`throw`) uma exceção e o código que *não* deve ser executado se ocorrer uma exceção (isto é, se ocorrer uma exceção, o código restante no bloco `try` será pulado). Um bloco `try` consiste na palavra-chave `try` seguida por um bloco de código entre chaves (`{}`). [*Observação:* o termo "bloco `try`" às vezes só se refere ao bloco de código que segue a palavra-chave `try` (não incluindo a própria palavra-chave `try`). Para simplificar, utilizamos o termo "bloco `try`" para nos referirmos ao bloco de código que se segue à palavra-chave `try`, bem como a palavra-chave `try`.] Todas as instruções que leem os inteiros a partir do teclado (linhas 25 e 27) utilizam o método `nextInt` para ler um valor `int`. O método `nextInt` lança uma `InputMismatchException` se o valor lido *não* for um número inteiro.

A divisão pode fazer com que uma `ArithmeticException` não seja realizada no bloco `try`. Em vez disso, a chamada para o método `quotient` (linha 29) invoca o código que tenta a divisão (linha 12); a JVM *lança* um objeto `ArithmeticException` quando o denominador for zero.

Observação de engenharia de software 11.1

As exceções emergem pelo código explicitamente mencionado em um bloco `try`, por chamadas de método profundamente aninhadas iniciado pelo código em um bloco `try` ou a partir da Java Virtual Machine à medida que ela executa os bytecodes do Java.

Capturando exceções

O bloco `try` nesse exemplo é seguido por dois blocos `catch` — um que trata uma `InputMismatchException` (linhas 34 a 41) e um que trata uma `ArithmeticException` (linhas 42 a 47). Um **bloco `catch`** (também chamado de **cláusula `catch`** ou **rotina de tratamento de exceção**) *captura* (isto é, recebe) e *trata* uma exceção. Um bloco `catch` inicia com a palavra-chave `catch` seguido por um parâmetro entre parênteses (chamado *parâmetro de exceção*, discutido em breve) e um bloco de código entre chaves. [*Observação:* o termo "cláusula `catch`" é às vezes utilizado para referir-se à palavra-chave `catch` seguida por um bloco de código, em que o termo "bloco `catch`" refere-se apenas ao bloco de código que se segue à palavra-chave `catch`, mas que não a inclui. Para simplificar, utilizamos o termo "bloco `catch`" para nos referirmos ao bloco de código que se segue à palavra-chave `catch`, bem como à própria palavra-chave.]

Pelo menos um bloco `catch` ou um **bloco `finally`** (discutidos na Seção 11.6) *deve* se seguir imediatamente ao bloco `try`. Cada bloco `catch` especifica entre parênteses um **parâmetro de exceção** que identifica o tipo de exceção que a rotina de tratamento pode processar. Quando ocorrer uma exceção em um bloco `try`, o bloco `catch` que é executado é o *primeiro* cujo tipo corresponde ao tipo da exceção que ocorreu (isto é, o tipo no bloco `catch` corresponde exatamente ao tipo de exceção lançado ou é uma superclasse direta ou indireta dele). O nome do parâmetro de exceção permite ao bloco `catch` interagir com um objeto de exceção capturado —

Multi-catch

É relativamente comum que um bloco `try` seja seguido por vários blocos `catch` para tratar vários tipos de exceção. Se os corpos dos vários blocos `catch` forem idênticos, use o recurso **multi-catch** (introduzido no Java SE 7) para capturar esses tipos de exceção em uma *única rotina de tratamento* `catch` e realizar a mesma tarefa. A sintaxe para um *multi-catch* é:

```
catch (Tipo1 | Tipo2 | Tipo3 e)
```

Cada tipo de exceção é separado do seguinte por uma barra vertical (|). A linha anterior do código indica que *qualquer um* dos tipos (ou suas subclasses) pode ser capturado na rotina de tratamento de exceção. Quaisquer tipos `Throwable` podem ser especificados em um multi-`catch`.

Exceções não capturadas

Uma **exceção não capturada** é uma exceção para a qual não existem blocos `catch` correspondentes. Vimos exceções não capturadas na segunda e terceira saídas da Figura 11.2. Lembre-se de que quando ocorreram exceções nesse exemplo, o aplicativo terminou precocemente (após exibir o *rastreamento de pilha* da exceção). Isso nem sempre ocorre como um resultado de exceções não capturadas. O Java usa um modelo "multiencadeado" (ou *multithreaded*) de execução de programas — cada **thread** é uma *atividade concorrente*. Um programa pode ter muitas threads. Se um programa tiver apenas *uma* thread, uma exceção não capturada fará com que ele seja encerrado. Se um programa tiver *múltiplas* threads, uma exceção não capturada encerrará *apenas* a thread em que ocorreu a exceção. Nesses programas, porém, certas threads podem contar com outras e se uma thread for encerrada por causa de uma exceção não capturada, pode haver efeitos adversos no restante do programa. O Capítulo 23, "Concorrência", discute essas questões em profundidade.

Modelo de terminação do tratamento de exceção

Se ocorrer uma exceção em um bloco `try` (como uma `InputMismatchException` sendo lançada como resultado do código na linha 25 da Figura 11.3), o bloco `try` *termina* imediatamente e o controle do programa é transferido para o *primeiro* dos blocos `catch` seguintes em que o tipo do parâmetro de exceção corresponde ao tipo da exceção lançada. Na Figura 11.3, o primeiro bloco `catch` captura `InputMismatchExceptions` (que ocorrem se uma entrada inválida for fornecida), e o segundo bloco `catch` captura `ArithmeticExceptions` (que ocorrem se houver uma tentativa de dividir por zero). Depois que a exceção é tratada, o controle do programa *não retorna* ao ponto de lançamento, porque o bloco `try` *expirou* (e suas *variáveis locais* foram *perdidas*). Em vez disso, o controle retoma depois do último bloco `catch`. Isso é conhecido como o **modelo de terminação do tratamento de exceção**. Algumas linguagens utilizam o **modelo de retomada do tratamento de exceção**, em que, após uma exceção ser tratada, o controle é retomado logo depois do *ponto de lançamento*.

Observe que nomeamos nossos parâmetros de exceção (`inputMismatchException` e `arithmeticException`) com base em seu tipo. Os programadores Java costumam simplesmente utilizar a letra `e` como o nome de parâmetros de exceção.

Depois de executar um bloco `catch`, o fluxo de controle desse programa prossegue para a primeira instrução depois do último bloco `catch` (linha 48 nesse caso). A condição na instrução `do...while` é `true` (a variável `continueLoop` contém seu valor inicial de `true`), então o controle retorna ao começo do loop e uma entrada é novamente solicitada ao usuário. Essa instrução de controle fará loop até que a entrada *válida* seja inserida. Nesse ponto, o controle de programa alcança a linha 32, que atribui `false` à variável `continueLoop`. O bloco `try` então *termina*. Se nenhuma exceção for lançada no bloco `try`, os blocos `catch` são *ignorados* e o controle continua com a primeira instrução depois dos blocos `catch` (veremos outra possibilidade ao discutir o bloco `finally` na Seção 11.6). Agora a condição para o loop `do...while` é `false` e o método `main` é encerrado.

O bloco `try` e seus blocos `catch` e/ou `finally` correspondentes formam uma **instrução `try`**. Não confunda os termos "bloco `try`" e "instrução `try`" — a última inclui o bloco `try`, bem como os seguintes blocos `catch` e/ou bloco `finally`.

Como acontece com qualquer outro bloco de código, quando um bloco `try` termina, *as variáveis locais* declaradas no bloco *saem de escopo* e não estão mais acessíveis; assim, as variáveis locais de um bloco `try` não são acessíveis nos blocos `catch` correspondentes. Quando um bloco `catch` *termina*, as *variáveis locais* declaradas dentro do bloco `catch` (incluindo o parâmetro de exceção desse bloco `catch`) também *saem de escopo* e são *destruídas*. Quaisquer blocos `catch` restantes na instrução `try` são *ignorados*, e a execução é retomada na primeira linha de código depois da sequência `try...catch` — essa será um bloco `finally`, se houver um presente.

Utilizando a cláusula `throws`

No método `quotient` (Figura 11.3, linhas 9 a 13), a linha 10 é conhecida como cláusula `throws`. Ela especifica as exceções que o método *pode* lançar se ocorrerem problemas. Essa cláusula, que deve aparecer após a lista de parâmetros e antes do corpo do método, contém uma lista separada por vírgulas dos tipos de exceção. Essas exceções podem ser lançadas por instruções no corpo do método ou por métodos chamados a partir daí. Adicionamos a cláusula `throws` a esse aplicativo para indicar que o método pode lançar uma `ArithmeticException`. Os chamadores do método `quotient` são assim informados de que o método pode lançar uma `ArithmeticException`. Alguns tipos de exceção, como `ArithmeticException`, não precisam ser listados na cláusula `throws`. Para aqueles que precisam, o método pode lançar exceções que têm o relacionamento *é um* com as classes listadas na cláusula `throws`. Você aprenderá mais sobre isso na Seção 11.5.

11.4 Quando utilizar o tratamento de exceção

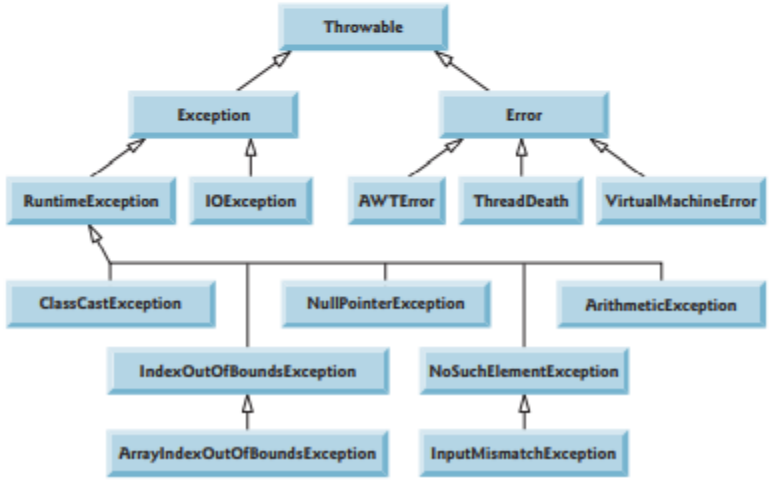
O tratamento de exceção é projetado para processar **erros síncronos**, que ocorrem quando uma instrução executa. Os exemplos mais comuns que veremos ao longo do livro são *índices de array fora dos limites*, *estouro aritmético* (isto é, um valor fora do intervalo representável dos valores), *divisão por zero*, *parâmetros de método inválidos* e *interrupção de thread* (como veremos no Capítulo 23). O tratamento de exceção não é projetado para processar problemas associados com os **eventos assíncronos** (por exemplo, conclusões de E/S de disco, chegadas de mensagem de rede, cliques de mouse e pressionamentos de tecla), que ocorrem paralelamente com fluxo de controle do programa e *independentemente* dele.

11.5 Hierarquia de exceção Java

Todas as classes de exceção do Java herdam direta ou indiretamente da classe `Exception`, formando uma *hierarquia de herança*. Você pode estender essa hierarquia com suas próprias classes de exceção.

A Figura 11.4 mostra uma pequena parte da hierarquia de herança da classe `Throwable` (uma subclasse de `Object`), que é a superclasse da classe `Exception`. Somente objetos `Throwable` podem ser utilizados com o mecanismo de tratamento de exceção. A classe `Throwable` tem duas subclasses diretas: `Exception` e `Error`. A classe `Exception` e suas subclasses — por exemplo, `RuntimeException` (pacote `java.lang`) e `IOException` (pacote `java.io`) — representam situações excepcionais que podem ocorrer em um programa Java e que podem ser capturadas pelo aplicativo. A classe `Error` e suas subclasses representam *situações anormais* que acontecem na JVM. A maioria dos `Errors` *acontece com pouca frequência e não deve ser capturada pelos aplicativos — geralmente os aplicativos não podem se recuperar de* `Errors`.

A hierarquia de exceções do Java contém centenas de classes. As informações sobre classes de exceção do Java podem ser localizadas por toda a Java API. Você pode visualizar a documentação `Throwable` em `docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html`. A partir daí, você pode examinar as subclasses dessa classe para obter informações adicionais sobre `Exceptions` e `Errors` do Java.



11.6 Bloco `finally`

Programas que obtêm certos recursos devem retorná-los ao sistema para evitar os assim chamados **vazamentos de recurso**. Em linguagens de programação como C e C++, o tipo mais comum de vazamento de recurso é um *vazamento de memória*. O Java realiza *coleta automática de lixo* de memória não mais utilizada por programas, evitando assim a maioria dos vazamentos de memória. Entretanto, outros tipos de vazamentos de recurso podem ocorrer. Por exemplo, arquivos, conexões de banco de dados e conexões de rede que não são fechadas adequadamente depois que não são mais necessárias talvez não estejam disponíveis para uso em outros programas.

Dica de prevenção de erro 11.4

Uma questão sutil é que o Java não elimina inteiramente os vazamentos de memória. O Java não efetuará coleta de lixo de um objeto até que não haja nenhuma referência a ele. Portanto, se você mantiver erroneamente referências a objetos indesejáveis, vazamentos de memória podem ocorrer.

O bloco `finally` (que consiste na palavra-chave `finally`, seguida pelo código entre chaves), às vezes referido como a **cláusula `finally`**, é opcional. Se estiver presente, ele é colocado depois do último bloco `catch`. Se não houver blocos `catch`, o bloco `finally`, se presente, segue imediatamente o bloco `try`.

Quando o bloco `finally` é executado

O `finally` será executado se uma exceção *for ou não* lançada no bloco `try` correspondente. O bloco `finally` também será executado se um bloco `try` for fechado usando uma instrução `return`, `break` ou `continue` ou simplesmente quando alcança a chave de fechamento direita. O caso em que o bloco `finally` *não* executará é se o aplicativo *sair precocemente* do bloco `try` chamando o método `System.exit`. Esse método, que demonstraremos no Capítulo 15, termina *imediatamente* um aplicativo.

Se uma exceção que ocorre em um bloco `try` não puder ser capturada por rotinas de tratamento `catch` desse bloco `try`, o programa pula o restante do bloco `try` e o controle prossegue para o bloco `finally`. Então, o programa passa a exceção para o próximo bloco `try` externo — normalmente no método chamador — onde um bloco `catch` associado pode capturá-lo. Esse processo pode ocorrer pelos muitos níveis de blocos `try`. Além disso, a exceção talvez *não seja capturada* (como discutido na Seção 11.3).

Se um bloco `catch` lançar uma exceção, o bloco `finally` ainda executará. Então, a exceção é passada para o próximo bloco `try` externo — novamente, em geral no método chamador.

JAVA – CAP 2022

Considere a seguinte Classe Excecao implementada em Java:

```
public class Excecao {

    public static void main(String[] args) {
        int num[] = new int[2];
        try {
            num[0] = 3;
            num[1] = 4;
            num[2] = 6;
            System.out.println( "sucesso" );
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println( "erro" );
        }finally{
            System.out.println( "final" );
        }
    }
}
```

Quais strings serão impressas no console?

a) sucesso, erro, final
b) sucesso, final
c) erro, final
d) final

JAVA – CAP 2022

Considere a seguinte Classe Excecao implementada em Java:

```
public class Excecao {

    public static void main(String[] args) {
        int num[] = new int[2];
        try {
            num[0] = 3;
            num[1] = 4;
            num[2] = 6;
            System.out.println( "sucesso" );
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println( "erro" );
        }finally{
            System.out.println( "final" );
        }
    }
}
```

Quais strings serão impressas no console?

a) sucesso, erro, final
b) sucesso, final
c) erro, final
d) final

JAVA
Interface Gráfica

Uma interface gráfica com usuário (Graphical User Interface -GUI) apresenta um mecanismo amigável ao usuário para interagir com um aplicativo. Uma GUI dá ao aplicativo uma aparência e comportamento distintos. Fornecer aos diferentes aplicativos componentes de interface com o usuário consistentes e intuitivos permite que os usuários se familiarizem com um novo aplicativo, para que possam aprendê-lo mais rapidamente e utilizá-lo mais produtivamente.
As GUIs são construídas a partir de componentes GUI. Esses são às vezes chamados controles ou widgets - abreviação de window gadgets. Um componente GUI é um objeto com que o usuário interage via mouse, teclado ou outro formulário de entrada, como reconhecimento de voz.

Desde a atualização 10 do Java SE 6, o Java vem com uma nova interface, elegante e compatível com várias plataformas conhecida como NIMBUS.

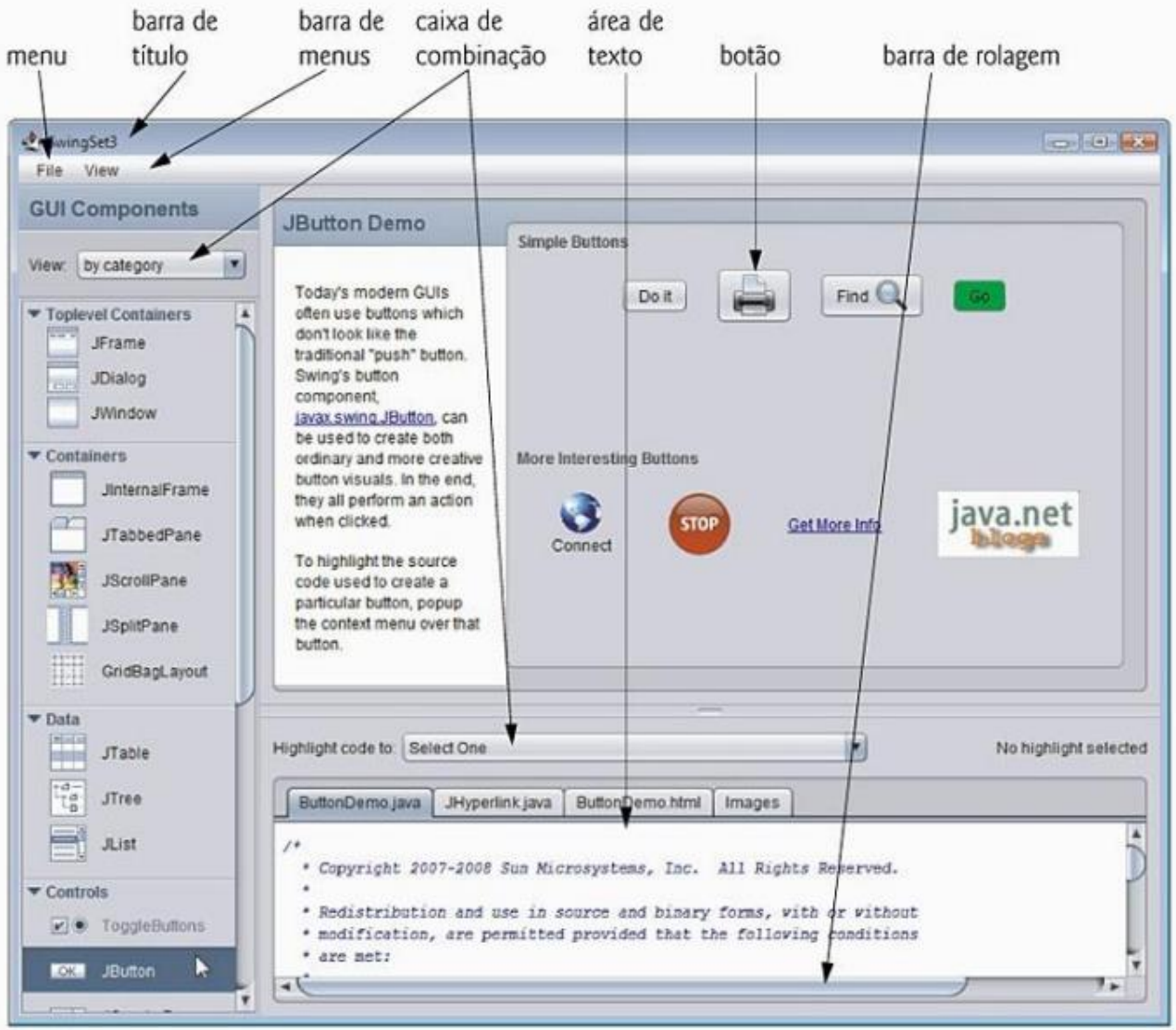


Figura 14.1 | Aplicativo SwingSet3 demonstra muitos dos componentes Swing GUI do Java.

JAVA – CAP 2022

14.3 Entrada/saída baseada em GUI simples com JOptionPane

Os aplicativos nos capítulos 2 a 10 exibem o texto na janela de comando e obtêm a entrada a partir da janela de comando. A maioria dos aplicativos que você utiliza diariamente utiliza janelas ou **caixas de diálogo** (também chamadas de **diálogos**) para interagir com o usuário. Por exemplo, um programa de email permite digitar e ler mensagens em uma janela fornecida pelo programa. Em geral, as caixas de diálogo são janelas em que programas exibem mensagens importantes para o usuário ou obtêm informações do usuário. A classe `JOptionPane` do Java (pacote `javax.swing`) fornece caixas de diálogo pré-construídas tanto para entrada como para saída. Esses diálogos são exibidos invocando métodos `JOptionPane static`. A Figura 14.2 apresenta um aplicativo de adição simples que utiliza dois **diálogos de entrada** para obter inteiros do usuário e um **diálogo de mensagem** para exibir a soma dos inteiros que o usuário insere.

```
1  // Figura 14.2: Addition.java
2  // Programa de adição que utiliza JOptionPane para entrada e saída.
3  import javax.swing.JOptionPane; // programa utiliza JOptionPane
4
5  public class Addition
6  {
7     public static void main( String[] args )
8     {
9        // obtém a entrada de usuário a partir dos diálogos de entrada JOptionPane
10       String firstNumber =
11          JOptionPane.showInputDialog( "Enter first integer" );
12       String secondNumber =
13           JOptionPane.showInputDialog( "Enter second integer" );
14
15       // converte String em valores int para utilização em um cálculo
16       int number1 = Integer.parseInt( firstNumber );
17       int number2 = Integer.parseInt( secondNumber );
18
19       int sum = number1 + number2; // soma os números
20
21       // exibe o resultado em um diálogo de mensagem JOptionPane
22       JOptionPane.showMessageDialog( null, "The sum is " + sum,
23          "Sum of Two Integers", JOptionPane.PLAIN_MESSAGE );
24    } // fim do método main
25 } // fim da classe Addition
```

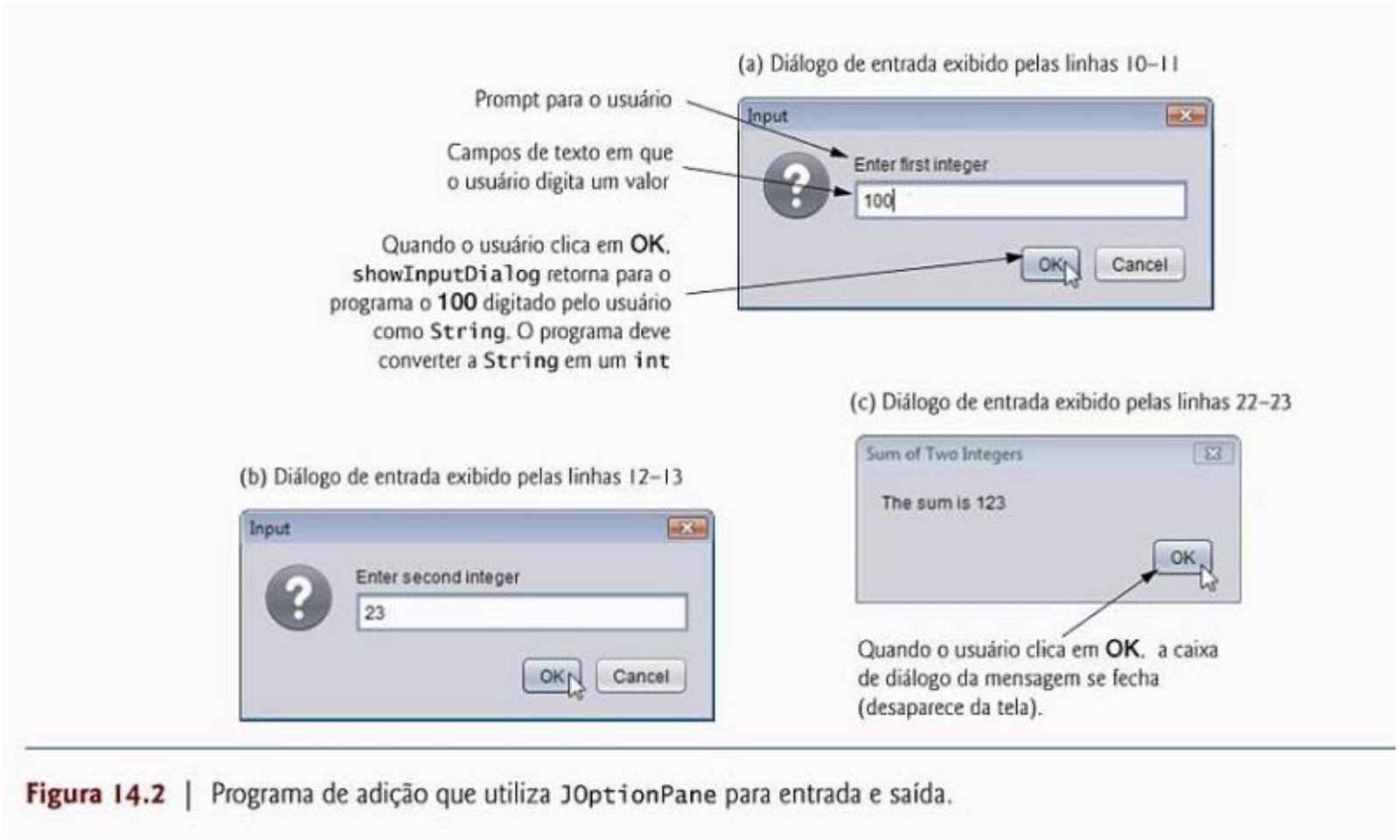


Figura 14.2 | Programa de adição que utiliza `JOptionPane` para entrada e saída.

Constantes de diálogo de mensagem `JOptionPane`

As constantes que representam os tipos de diálogo de mensagem são mostradas na Figura 14.3. Todos os tipos de diálogo de mensagem exceto `PLAIN_MESSAGE` exibem um ícone à esquerda da mensagem. Esses ícones fornecem uma indicação visual da importância da mensagem para o usuário. Observe que um ícone `QUESTION_MESSAGE` é o ícone padrão de uma caixa de diálogo de entrada (ver Figura 14.2).

Tipo de diálogo de mensagem	Ícone	Descrição
`ERROR_MESSAGE`		Indica um erro ao usuário.
`INFORMATION_MESSAGE`		Indica uma mensagem informativa ao usuário.
`WARNING_MESSAGE`		Alerta o usuário de um potencial problema.
`QUESTION_MESSAGE`		Propõe uma questão 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.

Figura 14.3 | Constantes `JOptionPane` `static` para diálogo de mensagem.

14.4 Visão geral de componentes Swing

Embora seja possível realizar entrada e saída utilizando os diálogos `JOptionPane` apresentados na Seção 14.3, a maioria dos aplicativos GUI exige interfaces com o usuário mais elaboradas e personalizadas. O restante deste capítulo discute muitos componentes GUI que permitem aos desenvolvedores de aplicações criar GUIs robustas. A Figura 14.4 lista vários **componentes GUI Swing** do pacote `javax.swing` que são utilizados para construir GUIs Java. A maioria dos componentes Swing são componentes Java puros — eles são completamente escritos, manipulados e exibidos em Java. Eles fazem parte das **Java Foundation Classes (JFC)** — bibliotecas do Java para desenvolvimento de GUI para múltiplas plataformas. Visite `java.sun.com/javase/technologies/desktop/` para obter mais informações sobre tecnologias desktop JFC e Java.

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 ser ou não 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.

Figura 14.4 | Alguns componentes GUI básicos.

Swing versus AWT

Há realmente dois conjuntos de componentes GUI no Java. Antes de o Swing ter sido introduzido no J2SE 1.2, as Java GUIs eram construídas com componentes do **Abstract Window Toolkit (AWT)** no pacote `java.awt`. Quando um aplicativo Java com uma AWT GUI executa em diferentes plataformas Java, os componentes GUI do aplicativo exibem diferentemente em cada plataforma. Considere um aplicativo que exibe um objeto do tipo `Button` (pacote `java.awt`). Em um computador que executa o sistema operacional do Microsoft Windows, `Button` terá a mesma aparência dos botões de outros aplicativos Windows. De maneira semelhante, em um computador que executa o sistema operacional Mac OS X da Apple, `Button` terá a mesma aparência e comportamento dos botões em outros aplicativos Macintosh. Às vezes, a maneira como um usuário pode interagir com um componente AWT particular difere entre plataformas.

Rotulando componentes GUI

Uma GUI típica consiste em muitos componentes. Em uma grande GUI, pode ser difícil identificar o propósito de cada componente. Portanto, os projetistas de GUIs costumam fornecer um texto que declara a finalidade de cada componente. Tal texto é conhecido como **rótulo** e é criado com a classe `JLabel` — uma subclasse de `JComponent`. Um `JLabel` exibe texto somente de leitura, uma imagem ou tanto texto como imagem. Os aplicativos raramente alteram o conteúdo de um rótulo depois de criá-lo.

Observações sobre a aparência e comportamento 14.5

Normalmente, o texto em um `JLabel` emprega maiúsculas e minúsculas no estilo de frases.

O aplicativo das figuras 14.6 e 14.7 demonstra vários recursos `JLabel` e apresenta o framework que utilizamos na maioria de nossos exemplos de GUIs. Não destacamos o código nesse exemplo, visto que a maior parte dele é nova. [*Nota:* há muito mais recursos para cada componente GUI que podemos abranger em nossos exemplos. Para aprender os detalhes completos de cada componente GUI, visite sua página na documentação on-line. Para a classe `JLabel`, visite `java.sun.com/javase/6/docs/api/javax/swing/JLabel.html`.]

```
1  // Figura 14.6: LabelFrame.java
2  // Demonstrando a classe JLabel.
3  import java.awt.FlowLayout; // especifica como os componentes são organizados
4  import javax.swing.JFrame; // fornece recursos básicos de janela
5  import javax.swing.JLabel; // exibe texto e imagens
6  import javax.swing.SwingConstants; // constantes comuns utilizadas com Swing
7  import javax.swing.Icon; // interface utilizada para manipular imagens
8  import javax.swing.ImageIcon; // carrega imagens
9
10 public class LabelFrame extends JFrame
11 {
12    private JLabel label1; // JLabel apenas com texto
13    private JLabel label2; // JLabel construído com texto e ícone
14    private JLabel label3; // JLabel com texto e ícone adicionados
15
16    // construtor LabelFrame adiciona JLabels a JFrame
17    public LabelFrame()
18    {
19       super( "Testing JLabel" );
```

JAVA – CAP 2022

```
20          setLayout( new FlowLayout() ); // configura o layout de frame
21
22          // Construtor JLabel com um argumento de string
23          label1 = new JLabel( "Label with text" );
24          label1.setToolTipText( "This is label1" );
25          add( label1 ); // adiciona o label1 ao JFrame
26
27          // construtor JLabel com string, Icon e argumentos de alinhamento
28          Icon bug = new ImageIcon( getClass().getResource(  "bug1.png" ) );
29          label2 = new JLabel( "Label with text and icon", bug,
30             SwingConstants.LEFT );
31          label2.setToolTipText( "This is label2" );
32          add( label2 ); // adiciona label2 ao JFrame
33
34          label3 = new JLabel(); // Construtor JLabel sem argumentos
35          label3.setText( "Label with icon and text at bottom" );
36          label3.setIcon( bug ); // adiciona o ícone ao JLabel
37          label3.setHorizontalTextPosition( SwingConstants.CENTER );
38          label3.setVerticalTextPosition( SwingConstants.BOTTOM );
39          label3.setToolTipText( "This is label3" );
40          add( label3 ); // adiciona label3 ao JFrame
41       } // fim do construtor LabelFrame
42    } // fim da classe LabelFrame
```

JAVA – CAP 2022

```
1  // Figura 14.7: LabelTest.java
2  // Testando LabelFrame.
3  import javax.swing.JFrame;
4
5  public class LabelTest
6  {
7     public static void main( String[] args )
8     {
9        LabelFrame labelFrame = new LabelFrame(); // cria LabelFrame
10       labelFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       labelFrame.setSize( 260, 180 ); // configura o tamanho do frame
12       labelFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe LabelTest
```

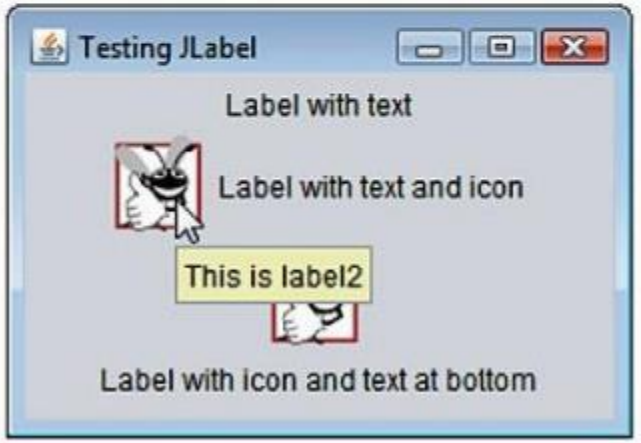


Figura 14.7 | Classe de teste para LabelFrame.

Constante	Descrição
Constantes de posição horizontal	
`SwingConstants.LEFT`	Coloca o texto à esquerda.
`SwingConstants.CENTER`	Coloca o texto no centro.
`SwingConstants.RIGHT`	Coloca o texto à direita.
Constantes de posição vertical	
`SwingConstants.TOP`	Coloca o texto na parte superior.
`SwingConstants.CENTER`	Coloca o texto no centro.
`SwingConstants.BOTTOM`	Coloca o texto na parte inferior.

Figura 14.8 | Posicionando constantes.

Campos de Texto

```
1  // Figura 14.9: TextFieldFrame java
2  // Demonstrando a classe JTextField.
3  import java.awt.FlowLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JTextField;
8  import javax.swing.JPasswordField;
9  import javax.swing.JOptionPane;
10
11 public class TextFieldFrame extends JFrame
12 {
13    private JTextField textField1; // campo de texto com tamanho configurado
14    private JTextField textField2; // campo de texto construído com texto
15    private JTextField textField3; // campo de texto com texto e tamanho
16    private JPasswordField passwordField; // campo de senha com texto
17
18    // construtor TextFieldFrame adiciona JTextFields a JFrame
19    public TextFieldFrame()
20    {
21       super( "Testing JTextField and JPasswordField" );
22       setLayout( new FlowLayout() ); // configura o layout de frame
23
24       // constrói textfield com 10 colunas
25       textField1 = new JTextField( 10 );
26       add( textField1 ); // adiciona textField1 ao JFrame
27
28       // constrói campo de texto com texto padrão
29       textField2 = new JTextField( "Enter text here" );
30       add( textField2 ); // adiciona textField2 ao JFrame
31
32       // constrói textfield com o texto padrão e 21 colunas
33       textField3 = new JTextField( "Uneditable text field", 21 );
34       textField3.setEditable( false ); // desativa a edição
35       add( textField3 ); // adiciona textField3 ao JFrame
36
37       // constrói passwordfield com o texto padrão
38       passwordField = new JPasswordField( "Hidden text" );
39       add( passwordField ); // adiciona passwordField ao JFrame
40
```

Campos de Texto

```
41          // handlers de evento registradores
42          TextFieldHandler handler = new TextFieldHandler();
43          textField1.addActionListener( handler );
44          textField2.addActionListener( handler );
45          textField3.addActionListener( handler );
46          passwordField.addActionListener( handler );
47       } // fim do construtor TextFieldFrame
48
49       // classe interna private para tratamento de evento
50       private class TextFieldHandler implements ActionListener
51       {
52          // processa eventos de campo de texto
53          public void actionPerformed( ActionEvent event )
54          {
55             String string = ""; // declara string a ser exibida
56
57             // usuário pressionou Enter no JTextField textField1
58             if (event.getSource() == textField1 )
59                string = String.format( "textField1: %s",
60                   event.getActionCommand() );
61
62             // usuário pressionou Enter no JTextField textField2
63             else if (event.getSource() == textField2 )
64                string = String.format( "textField2: %s",
65                   event.getActionCommand() );
66
```

Campos de Texto

```
66
67          // usuário pressionou Enter no JTextField textField3
68          else if (event.getSource() == textField3 )
69             string = String.format( "textField3: %s",
70                event.getActionCommand() );
71
72          // usuário pressionou Enter no JTextField passwordFiel
73          else if (event.getSource() == passwordField )
74             string = String.format( "passwordField: %s",
75                event.getActionCommand() );
76
77          // exibe o conteúdo de JTextField
78          JOptionPane.showMessageDialog( null, string );
79       } // fim do método actionPerformed
80    } // fim da classe TextFieldHandler interna private
81 } // fim da classe TextFieldFrame
```

Figura 14.9 | JTextFields e JPasswordFields.

Campos de Texto

```
1  // Figura 14.10: TextFieldTest.java
2  // Testando TextFieldFrame.
3  import javax.swing.JFrame;
4
5  public class TextFieldTest
6  {
7     public static void main( String[] args )
8     {
9        TextFieldFrame textFieldFrame = new TextFieldFrame();
10       textFieldFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       textFieldFrame.setSize( 350, 100 ); // configura o tamanho do frame
12       textFieldFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe TextFieldTest
```

14.7 Tipos comuns de eventos GUI e interfaces ouvintes

Na Seção 14.6, você aprendeu que as informações sobre o evento que ocorre quando o usuário pressiona *Enter* em um campo de texto são armazenadas em um objeto `ActionEvent`. Muitos tipos diferentes de eventos podem ocorrer quando o usuário interage com uma GUI. As informações de evento são armazenadas em um objeto de uma classe que estende `AWTEvent` (do pacote `java.awt`). A Figura 14.11 ilustra uma hierarquia que contém muitas classes de evento do pacote **`java.awt.event`**. Alguns desses são discutidos neste capítulo e no Capítulo 25. Esses tipos de evento são utilizados tanto com componentes AWT como com Swing. Tipos adicionais de evento que são específicos dos componentes Swing GUI são declarados no pacote **`javax.swing.event`**.

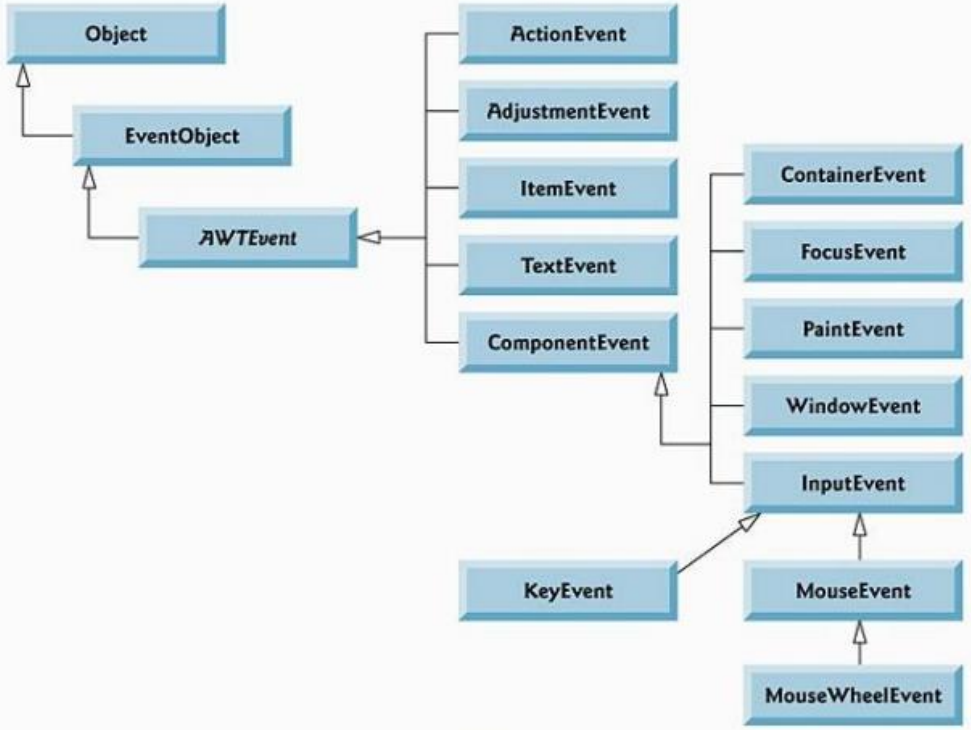


Figura 14.11 | Algumas classes de evento do pacote `java.awt.event`.

Vamos resumir as três partes para o mecanismo de tratamento de evento que você viu na Seção 14.6 — a origem do evento, o objeto do evento e o ouvinte de evento. A origem do evento é o componente GUI com o qual o usuário interage. O objeto do evento encapsula informações sobre o evento que ocorreu, como uma referência à origem do evento e quaisquer informações específicas do evento que podem ser exigidas pelo ouvinte de evento para tratar o evento. O ouvinte de evento é um objeto que é notificado pela origem de evento quando um evento ocorre; de fato, ele "ouve" um evento e um de seus métodos executa em resposta ao evento. Um método do ouvinte de evento recebe um objeto do evento quando o ouvinte de evento é notificado do evento. O ouvinte de evento então utiliza o objeto do evento para responder ao evento. O modelo de tratamento de evento descrito aqui é conhecido como **modelo de delegação de evento** — o processamento de um evento é delegado a um objeto particular (o ouvinte de evento) no aplicativo.

Para cada tipo de objeto de evento, há em geral uma interface listener de eventos correspondentes. Um ouvinte de evento para um evento GUI é um objeto de uma classe que implementa uma ou mais das interfaces ouvintes de evento dos pacotes `java.awt.event` e `javax.swing.event`. Muitos dos tipos de ouvinte de evento são comuns aos componentes Swing e AWT. Esses tipos são declarados no pacote `java.awt.event`, e alguns deles são mostrados na Figura 14.12. Os tipos de ouvinte de evento adicionais que são específicos dos componentes Swing são declarados no pacote `javax.swing.event`.

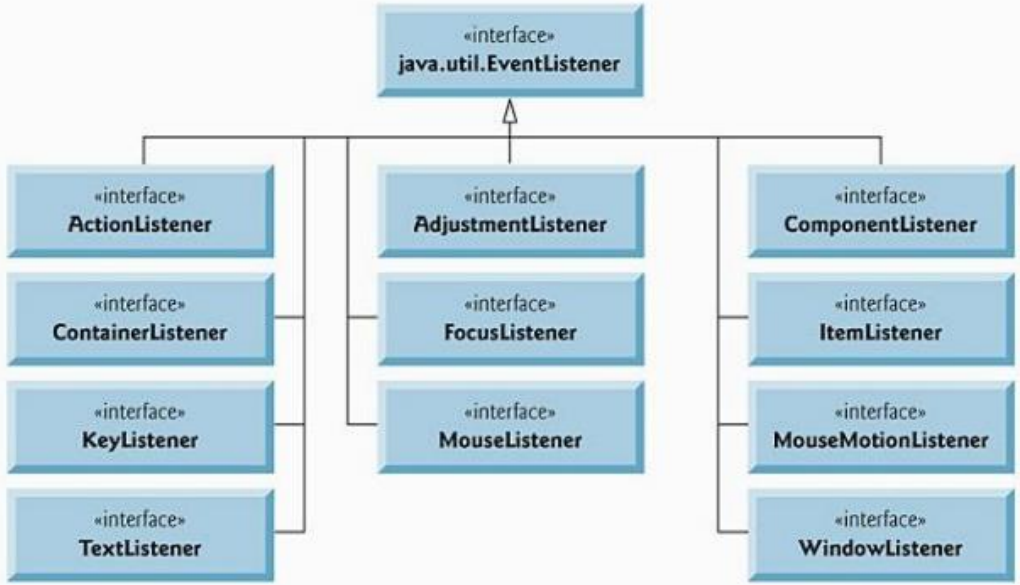


Figura 14.12 | Algumas interfaces listener de eventos comuns do pacote `java.awt.event`.

14.8 Como o tratamento de evento funciona

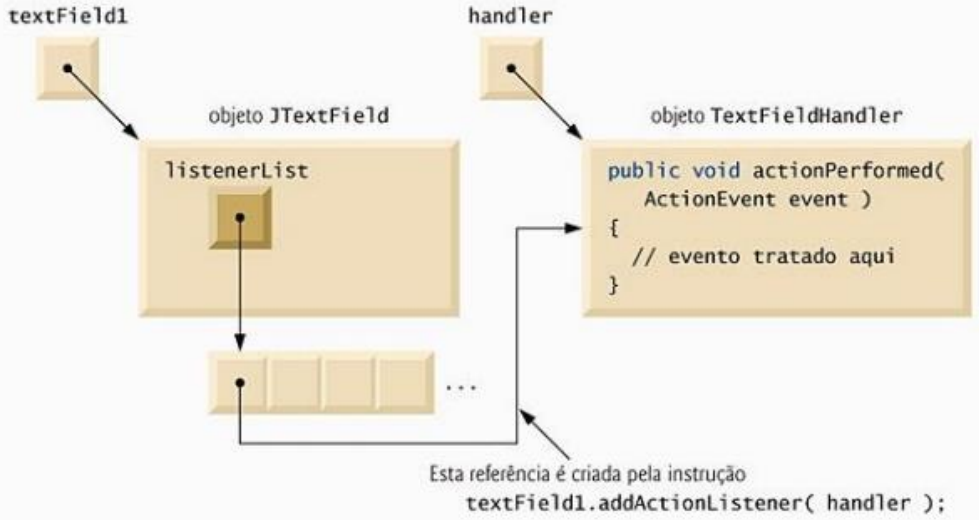
Vamos ilustrar como o mecanismo de tratamento de evento funciona, utilizando `textField1` do exemplo da Figura 14.9. Temos duas perguntas abertas restantes da Seção 14.7:

1. como o handler de evento é registrado?
2. como o componente GUI sabe chamar `actionPerformed` em vez de algum outro método de tratamento de evento?

A primeira pergunta é respondida pelo registro de evento realizado nas linhas 43–46 da Figura 14.9. A Figura 14.13 diagrama a variável `JTextField textField1`, variável `TextFieldHandler handler` e os objetos aos quais elas se referem.

Registrando eventos

Cada `Jcomponent` tem uma variável de instância chamada `listenerList` que referencia um objeto de classe **`EventListenerList`** (pacote `javax.swing.event`). Cada objeto de uma subclasse `JComponent` mantém referências a seus ouvintes (listeners) registrados na `listenerList`. Por uma questão de simplicidade, diagramamos `listenerList` como um array abaixo do objeto `JTextField` na Figura 14.13.



14.9 JButton

Um **botão** é um componente em que o usuário clica para acionar uma ação específica. Um aplicativo Java pode usar vários tipos de botões, incluindo **botões de comando**, **caixas de seleção**, **botões de alternação** e **botões de opção**. A Figura 14.14 mostra a hierarquia de herança dos botões do Swing que abordaremos neste capítulo. Como você pode ver, todos os tipos de botão são subclasses de `AbstractButton` (pacote `javax.swing`), que declara os recursos comuns de botões Swing. Nesta seção, concentramos-nos nos botões que são em geral utilizados para iniciar um comando.

14.10 Botões que mantêm o estado

Os componentes Swing GUI contêm três tipos de **botões de estado** — `JToggleButton`, `JCheckBox` e `JRadioButton` — isso tem valores ativado/desativado ou verdadeiro/falso. As classes `JCheckBox` e `JRadioButton` são subclasses de `JToggleButton` (Figura 14.14). Um `JRadioButton` é diferente de um `JCheckBox` no sentido de que normalmente vários `JRadioButtons` estão agrupados e são mutuamente exclusivos — apenas um no grupo pode ser selecionado de cada vez, assim como os botões em um rádio de carro de antigamente. Primeiro discutimos a classe `JCheckBox`.

14.10.1 JCheckBox

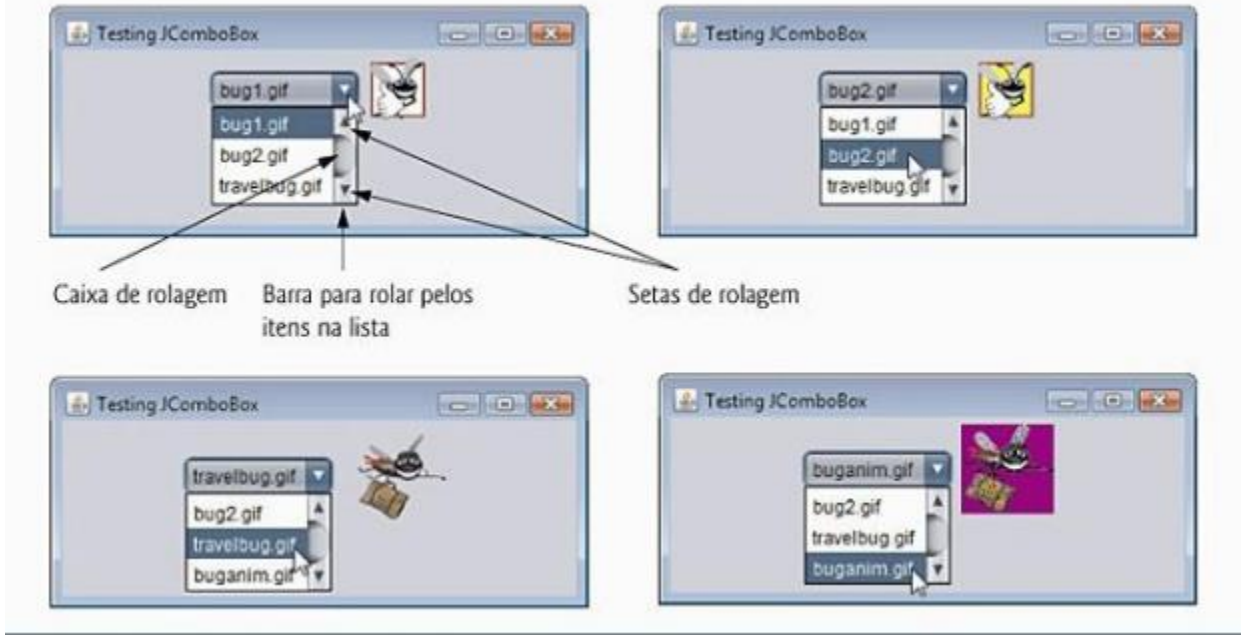
O aplicativo das figuras 14.17 e 14.18 usa dois `JcheckBoxes` para selecionar o estilo desejado de fonte do texto exibido em um `JTextField`. Quando selecionado, um aplica o estilo negrito e o outro, o estilo itálico. Se ambos forem selecionados, o estilo é negrito e itálico. Quando o aplicativo é inicialmente executado, nenhuma `JCheckBox` está marcada (isto é, ambas são `false`), então a fonte é simples. A classe `CheckBoxTest` (Figura 14.18) contém o método `main` que executa esse aplicativo.

14.10.2 JRadioButton

Os **botões de opção** (declarados com a classe `JRadioButton`) são semelhantes a caixas de seleção porque têm dois estados — selecionado e não selecionado. Entretanto, os botões de opção normalmente aparecem como um **grupo** em que apenas um botão pode ser selecionado por vez (ver a saída da Figura 14.20). Selecionar um botão de opção diferente força a remoção da seleção de todos os outros que estão selecionados. Os botões de opção são utilizados para representar **opções mutuamente exclusivas** (isto é, não é possível selecionar múltiplas opções no grupo ao mesmo tempo). O relacionamento lógico entre botões de opção é mantido por um objeto **`ButtonGroup`** (pacote `javax.swing`), que em si não é um componente GUI. Um objeto `ButtonGroup` organiza um grupo de botões e ele mesmo não é exibido em uma interface com o usuário. Em seu lugar, os objetos individuais `JRadioButton` do grupo são exibidos na GUI.

14.11 JComboBox e uso de uma classe interna anônima para tratamento de evento

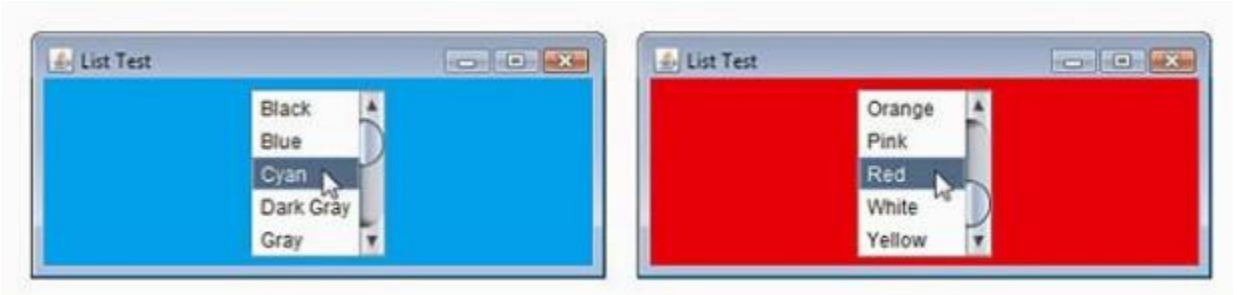
Uma caixa de combinação (às vezes chamada de **lista drop-down**) permite ao usuário selecionar um item de uma lista (Figura 14.22). As caixas de combinação são implementadas com a classe **JComboBox**, que estende a classe `JComponent`. `JComboBoxes` geram `ItemEvents` como `JCheckBoxes` e `JRadioButtons`. Esse exemplo também demonstra uma forma especial de classe interna que costuma ser utilizada no tratamento de evento. O aplicativo (figuras 14.21–14.22) utiliza uma `JComboBox` para fornecer uma lista de quatro nomes de arquivo de imagem a partir da qual o usuário pode selecionar uma imagem para ser exibida. Quando o usuário seleciona um nome, o aplicativo exibe a imagem correspondente como um `Icon` em um `Jlabel`. A classe `ComboBoxTest` (Figura 14.22) contém o método `main` que executa esse aplicativo. As capturas de tela para esse aplicativo mostram a lista `JComboBox` depois que a seleção foi feita para ilustrar qual nome de arquivo de imagem foi selecionado.



14.12 JList

Uma lista exibe uma série de itens a partir da qual o usuário pode selecionar um ou mais itens (ver a saída da Figura 14.24). As listas são criadas com a classe `JList`, que estende diretamente a classe `JComponent`. A classe `JList` suporta **listas de uma única seleção** (que permitem que apenas um item seja selecionado por vez) e **listas de seleção múltipla** (que permitem que qualquer número de itens seja selecionado). Nesta seção, discutimos listas de uma única seleção.

O aplicativo das figuras 14.23–14.24 cria uma `JList` contendo 13 nomes de cores. Quando se clica em um nome de cor na `JList`, ocorre um `ListSelectionEvent` e o aplicativo muda a cor de fundo da janela de aplicativo para a cor selecionada. A classe `ListTest` (Figura 14.24) contém o método `main` que executa esse aplicativo.



14.14 Tratamento de evento de mouse

Esta seção apresenta as interfaces ouvintes de evento `MouseListener` e `MouseMotionListener` para tratar **eventos 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 Figura 14.27 O pacote `javax.swing.event` contém a interface **`MouseInputListener`**, que estende as interfaces `MouseListener` e `MouseMotionListener` para criar uma única interface que contém todos os métodos `MouseListener` e `MouseMotionListener`. Os métodos `MouseListener` e `MouseMotionListener` são chamados quando o mouse interage com um `Component` se objetos listeners de evento apropriados forem registrados para esse `Component`.

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

`public void mouseMoved( MouseEvent event )`

Chamado quando o mouse é movido (sem pressionamentos de botões do mouse) quando o cursor do mouse está sobre um componente. Todos os eventos de movimento são enviados para o componente sobre o qual o mouse atualmente está posicionado.

Método InputEvent	Descrição
`isMetaDown()`	Retorna `true` quando o usuário clica no botão direito do mouse em um mouse com dois ou três botões. Para simular um clique de botão direito com um mouse de um botão, o usuário pode manter pressionada a tecla *Meta* no teclado e clicar no botão do mouse.
`isAltDown()`	Retorna `true` quando o usuário clica no botão do mouse do meio em um mouse com três botões. Para simular um clique com o botão do meio do mouse em um mouse com um ou dois botões, o usuário pode pressionar a tecla *Alt* e clicar no único botão ou no botão esquerdo do mouse, respectivamente.

Figura 14.33 | Métodos `InputEvent` que ajudam a distinguir entre os cliques do botão esquerdo, do centro e direito do mouse.

Gerenciador de layout	Descrição
`FlowLayout`	Padrão para `javax.swing.Jpanel`. Coloca os componentes sequencialmente (da esquerda para a direita) na ordem em 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.

Figura 14.38 | Gerenciadores de layout.

JAVA
Serviços WEB

Um serviço Web é um componente de software armazenado em um computador que pode ser acessado por um aplicativo (ou outro componente de software) em outro computador por uma rede. Serviços Web se comunicam utilizando tecnologias como XML, JSON e HTTP.

JAX-WS baseada em Simple Object Access Protocol - SOAP - Um protocolo baseado em XML que permite a serviços Web e clientes se comunicarem , mesmo que o cliente e o serviço Web sejam escritos em linguagens diferentes.

JAX-RS, utiliza o Representation State Transfer - REST - uma arquitetura de rede que utiliza mecanismos de solicitação/resposta tradicionais da Web como solicitações GET e POST.

Transações de empresa para empresa

Em vez de contar com aplicativos proprietários, as empresas podem conduzir transações via serviços Web padronizados, amplamente disponíveis. Isso tem implicações importantes para **transações business-to-business (B2B)**. Serviços Web são independentes de plataforma e linguagem, permitindo que empresas colaborem sem se preocupar com a compatibilidade de hardwares, softwares e tecnologias de comunicação. Empresas, como a Amazon, Google, eBay, PayPal e muitas outras tiram proveito disponibilizando seus aplicativos no lado do servidor a parceiros via serviços Web.

Comprando alguns serviços Web e utilizando outros gratuitos que são relevantes para seus negócios, o desenvolvimento de aplicativos para as empresas pode ser mais rápido e elas podem criar novos serviços que são mais inovadores. Sites de comércio eletrônico, por exemplo, podem fornecer aos seus clientes experiências de compra aprimoradas. Considere uma loja de músicas on-line. O site Web da loja oferece links para informações sobre vários CDs, permitindo aos usuários comprá-los, conhecer artistas, localizar mais títulos desses artistas, encontrar músicas de outros artistas que talvez eles possam gostar etc. O site Web da loja também pode oferecer links para o site de uma empresa que vende entradas para shows e fornecer um serviço Web que exibe datas de futuros concertos para vários artistas, permitindo aos usuários comprar entradas. Consumindo o serviço Web de entrada para shows no seu site, a loja de músicas on-line pode fornecer um serviço adicional aos seus clientes, aumentar o tráfego do site e talvez ganhar uma comissão sobre as vendas de entrada para shows. A empresa que vende entrada para shows também se beneficia da relação comercial vendendo mais bilhetes e possivelmente recebendo receitas da loja de músicas on-line pelo uso do serviço Web.

Qualquer programador Java com conhecimento de serviços Web pode escrever aplicativos que "consomem" serviços Web. Os aplicativos resultantes invocariam serviços Web em execução nos servidores que poderiam estar a milhares de quilômetros de distância.

JAVA – CAP 2022

NetBeans

O NetBeans é uma entre várias ferramentas que permitem "publicar" e/ou "consumir" serviços Web. Demonstramos como utilizar o NetBeans para implementar serviços Web utilizando as APIs do JAX-WS e JAX-RS e como invocá-los a partir de aplicativos clientes. Para cada exemplo, fornecemos o código do serviço Web e então apresentamos um aplicativo cliente que utiliza o serviço Web. Nossos primeiros exemplos constroem serviços Web e aplicativos clientes simples no NetBeans. Demonstramos então serviços Web que utilizam recursos mais sofisticados, como manipulação de bancos de dados com o JDBC e manipulação de objetos de classes. Para obter informações sobre como fazer o download e instalar o NetBeans e o servidor GlassFish v2 UR2, consulte a Seção 29.1.

31.2 Fundamentos do serviço Web

A máquina na qual um serviço Web reside é chamada **host de serviço Web**. O aplicativo cliente envia uma solicitação por uma rede ao host do serviço Web, que processa a solicitação e retorna uma resposta pela rede ao aplicativo. Esse tipo de computação distribuída beneficia os sistemas de vários modos. Por exemplo, um aplicativo sem acesso direto a dados em outro sistema poderia ser capaz de recuperar os dados via um serviço Web. De maneira semelhante, um aplicativo que não tem capacidade de processamento suficiente para realizar cálculos específicos poderia utilizar um serviço Web para tirar vantagem dos recursos superiores de outro sistema.

No Java, um serviço Web é implementado como uma classe. Nos capítulos anteriores, todas as partes de um aplicativo residiam em uma máquina. A classe que representa o serviço Web reside em um servidor — ela não é parte do aplicativo cliente. Disponibilizar um serviço Web para receber solicitações de cliente é conhecido como **publicar um serviço Web**; utilizar um serviço Web a partir de um aplicativo cliente é conhecida como **consumir um serviço Web**.

31.3 Simple Object Access Protocol (SOAP)

O Simple Object Access Protocol (SOAP) é um protocolo independente de plataforma que utiliza a XML para fazer chamadas de procedimento remoto, geralmente sobre o HTTP. Você pode visualizar a especificação SOAP em `www.w3.org/TR/soap/`. Cada solicitação e resposta são empacotadas em uma **mensagem SOAP** — marcação XML que contém as informações que um serviço Web exige para processar a mensagem. Mensagens SOAP são escritas em XML para que sejam legíveis por computadores e humanos e para que sejam independentes de plataforma. A maioria dos **firewalls** — as barreiras de segurança que restringem a comunicação entre redes — permite que o tráfego HTTP passe para que os clientes possam navegar pela Web enviando solicitações e recebendo respostas de servidores Web. Assim, serviços baseados em SOAP podem enviar e receber mensagens SOAP por meio de conexões HTTP com poucas limitações.

O SOAP suporta um extenso conjunto de tipos. O **formato wire** utilizado para transmitir solicitações e respostas deve suportar todos os tipos passados entre os aplicativos. Tipos SOAP incluem os tipos primitivos (por exemplo, `int`), bem como `DateTime`, `XmlNode`- e outros. O SOAP também pode transmitir arrays desses tipos.

Quando um programa invoca um método de um serviço Web SOAP, a solicitação e todas as informações relevantes são empacotadas em uma mensagem SOAP colocadas em um **envelope SOAP** e enviadas ao servidor no qual o serviço Web reside. Quando o serviço Web recebe essa mensagem SOAP, ele analisa a XML que representa a mensagem e então processa o conteúdo da mensagem. A mensagem especifica o método que o cliente deseja executar e os argumentos que o cliente passou para esse método. Em seguida, o serviço Web chama o método com os argumentos especificados (se houver um) e envia a resposta de volta ao cliente em outra mensagem SOAP. O cliente analisa a resposta para recuperar o resultado do método. Na Seção 31.6, você construirá e consumirá um serviço Web SOAP básico.

31.4 Representational State Transfer (REST)

O Representational State Transfer (REST) refere-se a um estilo arquitetônico de implementar serviços Web. Esses serviços Web costumam ser chamados de **serviços Web RESTful**. Embora o próprio REST não seja um padrão, serviços Web RESTful são implementados utilizando padrões Web. Cada método em um serviço Web RESTful é identificado por um URL único. Assim, quando o servidor recebe uma solicitação, ele sabe imediatamente que operação realizar. Esses serviços Web podem ser utilizados em um programa ou diretamente de um navegador Web. Os resultados de uma determinada operação podem ser armazenados em cache localmente pelo navegador quando o serviço é invocado com uma solicitação `GET`. Isso pode tornar solicitações subsequentes à mesma operação mais rápidas carregando o resultado diretamente do cache do navegador. Os serviços Web da Amazon (`aws.amazon.com`) são RESTful, assim como vários outros.

Serviços Web RESTful são alternativas àqueles implementados com o SOAP. Diferentemente dos serviços Web baseados em SOAP, a solicitação e resposta dos serviços REST não são empacotados em envelopes. O REST também não está limitado a retornar dados no formato XML. Ele pode utilizar vários formatos, como XML, JSON, HTML, texto sem formatação e arquivos de mídia. Nas seções 31.7–31.8, você construirá e consumirá serviços Web RESTful básicos.

31.5 JavaScript Object Notation (JSON)

A **JavaScript Object Notation (JSON)** é uma alternativa à XML para representar dados. JSON é um formato de troca de dados baseado em texto utilizado para representar objetos em JavaScript como coleções de pares nome/valor representados como `Strings`. Ele é comumente utilizado em aplicativos Ajax. JSON é um formato simples que facilita a leitura, criação e análise de objetos e, como é muito menos prolixo que a XML, permite que os programas transmitam dados eficientemente pela internet. Cada objeto JSON é representado como uma lista de nomes e valores de propriedade entre colchetes, no seguinte formato:

```
{ NomeDaPropriedade1 : valor1, NomeDaPropriedade2 : valor2 }
```

Arrays são representados no JSON com colchetes no seguinte formato:

```
[ valor1, valor2, valor3 ]
```

Cada valor em um array pode ser uma string, um número, um objeto JSON, `true`, `false` ou `null`. Para apreciar a simplicidade dos dados JSON, examine esta representação de um array de entradas de catálogo de endereços:

```
[ { first: 'Cheryl', last: 'Black' },
  { first: 'James', last: 'Blue' },
  { first: 'Mike', last: 'Brown' },
  { first: 'Meg', last: 'Gold' } ]
```

Muitas linguagens de programação agora suportam o formato de dados JSON. Uma extensa lista de bibliotecas JSON classificadas por linguagem pode ser localizada em `www.json.org`.