Capítulo 3

Estruturas de Controle e Programação Básica

Se você escrevesse um programa Java com o que sabe até agora, provavelmente ele seria um pouco sem graça. Neste capítulo você irá aprender a fazer parte de um programa Java repetir-se, usando laços. Além disso, você irá aprender a fazer um programa decidir se vai executar determinado grupo de instruções para a realização de determinada tarefa com base em uma lógica.

3.1 – Estruturas de Controle

Um programa de computador é uma seqüência de instruções organizadas de forma tal a produzir a solução de um determinado problema. Naturalmente tais instruções são executadas em seqüência, o que se denomina fluxo seqüencial de execução. Em inúmeras circunstâncias é necessário executar as instruções de um programa em uma ordem diferente da estritamente seqüencial. Tais situações são caracterizadas pela necessidade da repetição de instruções individuais ou de grupos de instruções e também pelo desvio do fluxo de execução.

As linguagens de programação tipicamente possuem diversas estruturas de programação destinadas ao controle do fluxo de execução, isto é, estruturas que permitem a repetição e o desvio do fluxo de execução. Geralmente as estruturas de controle de execução são divididas em:

- **Estruturas de repetição simples**
  Destinadas a repetição de um ou mais comandos, criando o que se denomina laços. Geralmente o número de repetições é pré-definido ou pode ser determinado pelo programa durante a execução. No Java dispõe-se da diretiva *fo*r.
- **Estruturas de desvio de fluxo**
  Destinadas a desviar a execução do programa para uma outra parte, quebrando o fluxo seqüencial de execução. O desvio do fluxo pode ocorrer condicionalmente, quando associado a avaliação de uma expressão, ou incondicionalmente. No Java dispõe-se das diretivas *if* e *switc*h.
- **Estruturas de repetição condicionais**
  Semelhantes as estruturas de repetição simples mas cuja repetição está associada a avaliação de uma condição sendo geralmente utilizadas quando não se conhece de antemão o número necessário de repetições. No Java dispõe-se das diretivas *while* e *do while*.

Além destas estruturas existem ainda:

- Mecanismos de modularização
- Estruturas de tratamento de exceções

Os mecanismos de modularização são aqueles que permitem a construção de funções e procedimentos (dentro do paradigma imperativo) ou métodos (dentro do paradigma da orientação a objetos) e serão discutidos no próximo capítulo. Já as estruturas de

tratamento de exceções constituem uma importante contribuição a programação pois simplificam bastante a inclusão e construção de rotinas de tratamento de erros dentro do código. Elas serão apresentadas neste capítulo, mas discutidas em maior profundidade em outro capítulo. Antes de tratarmos especificamente das estruturas de controle da linguagem é necessário colocarmos algumas definições. Formalmente as instruções de um programa são chamadas diretivas *(statement*s). Tradicionalmente as diretivas são escritas uma após a outra em um programa e são separadas de alguma forma, por exemplo com uma quebra de linha ou um caractere de pontuação.

Em Java, tal como na linguagem C/C++, as diretivas são separadas uma das outras através do símbolo de pontuação ';' (ponto e vírgula), sendo possível existir várias diretivas numa mesma linha desde que separadas por um ponto e vírgula. Abaixo temos um exemplo hipotético de várias diretivas colocadas seqüencialmente:

```
diretiva1;
diretiva2;
diretiva3;
...
diretivaN;
```

Como nas outras linguagens de programação, as estruturas de controle podem operar sobre diretivas isoladas (individuais) ou sobre várias diretivas tratadas como um conjunto, o qual é denominado bloco. Um bloco em Java é um grupo de diretivas delimitadas por chaves ({K}). Por sua vez um bloco de diretivas recebe um tratamento equivalente ao de uma única diretiva individual.

```
{
 diretiva1;
 diretiva2;
 diretiva3;              diretivaUnica;
 ...
  diretivaN;
}
```

3.2 – Estruturas de Repetição Simples

Como repetição simples consideramos um trecho de código, isto é, um conjunto de diretivas que deve ser repetido um número conhecido e fixo de vezes. A repetição é uma das tarefas mais comuns da programação utilizada para efetuarmos contagens, para obtenção de dados, para impressão etc. Em Java dispomos da diretiva *for* cuja sintaxe é dada a seguir:

```
for (inicialização; condição de execução;
incremento/decremento)
diretiva;
```

O *for* possui três campos ou seções, todas opcionais, delimitados por um par de parêntesis que efetuam o controle de repetição de uma diretiva individual ou de um bloco de diretivas. Cada campo é separado do outro por um ponto e vírgula. O primeiro campo é usado para dar valor inicial a uma variável de controle (um contador). O

segundo campo é uma expressão lógica que determina a execução da diretiva associada ao *fo*r, geralmente utilizando a variável de controle e outros valores.

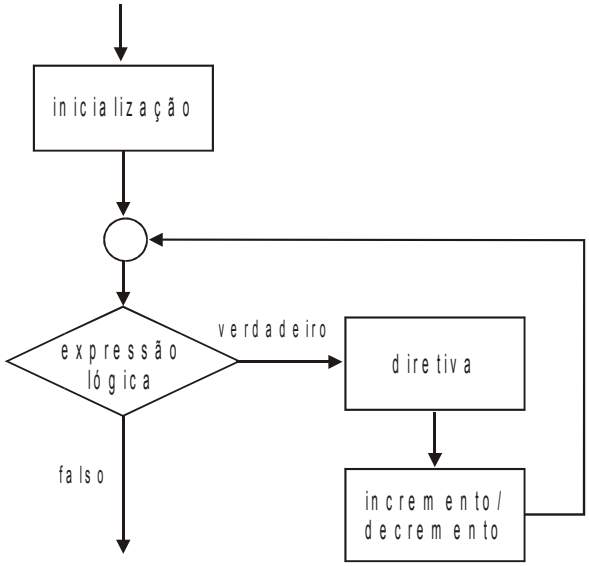

Figura 3. 1 – Comportamento da diretiva for

Após a execução da seção de inicialização ocorre a avaliação da expressão lógica. Portanto, a diretiva for também pode ser usada como uma estrutura de repetição condicional. Se a expressão é avaliada como verdadeira, a diretiva associada é executada, caso contrário o comando for é encerrado e a execução do programa prossegue com o próximo comando após o *fo*r. O terceiro campo determina como a variável de controle será modificada a cada iteração do *fo*r. Considera-se como iteração a execução completa da diretiva associada, fazendo que ocorra o incremento ou decremento da variável de controle. A seguir, mostra-se um exemplo de utilização da diretiva *fo*r:

```
public class exemploFor {
public static void main (String args[]) {
 int j;
 for (j=0; j<10; j++)
      {
       System.out.println("" + j);
      }
 }
}
```

Exemplo 3. 1 - Utilização da diretiva for

Se executado, o exemplo acima deverá exibir uma contagem de 0 até 9 onde cada valor é exibido numa linha do console. Na diretiva for, comandos podem ser agrupados. Além disso, também é possível a definição de múltiplas variáveis de controle. O exemplo abaixo mostra o uso do for nestas situações:

```
public class usoDoFor {
 public static void main(String[] args) {
  for(int i = 1, j = i + 10; i < 5; i++, j = i * 2)
    {
     System.out.println("i= " + i + " j= " + j);
    }
```

```
  }
}
```

Exemplo 3. 2 - Uso de Múltiplas Variáveis de Controle no for

O programa acima possui a seguinte saída:

```
i= 1 j= 11
i= 2 j= 4
i= 3 j= 6
i= 4 j= 8
```

3.3 – Estruturas de Desvio de Fluxo

Existem várias estruturas de desvio de fluxo que podem provocar a modificação da maneira com que as diretivas de um programa são executadas conforme a avaliação de uma condição. O Java dispõe de duas destas estruturas: *if* e *switc*h.

O *if* é uma estrutura simples de desvio de fluxo de execução, isto é, ela é uma diretiva que permite a seleção entre dois caminhos distintos para execução dependendo do resultado falso ou verdadeiro resultante de uma expressão lógica.

```
if (expressão_lógica)
  diretiva1;
else
  diretiva2;
```

A diretiva *if* permite duas construções possíveis: a primeira, utilizando a parte obrigatória, condiciona a execução da diretiva1 a um resultado verdadeiro oriundo da avaliação da expressão lógica associada; a segunda, usando opcionalmente o *else*, permite que seja executada a diretiva1 caso o resultado da expressão seja verdadeiro ou que seja executada a diretiva2 caso tal resultado seja falso. Isto caracteriza o comportamento ilustrado pela Figura 3.2.

A seguir um exemplo de uso da diretiva *if*.

```
public class exemploIf {
public static void main (String args[]) {
   if (args.length > 0)
   {
    for (int j=0; j<Integer.parseInt(args[0]); j++)
       {
        System.out.print("" + j + " ");
       }
    System.out.println("\nFim da Contagem");
   }
   System.out.println("Fim do Programa");
 }
}
```

Exemplo 3. 3 - Utilização da diretiva if

Ao executar-se este programa podem ocorrer duas situações distintas como resultado: se foi fornecido algum argumento na linha de comando (args.length > 0), o programa o

converte para um número inteiro (Integer.parseInt(args[0])) e exibe uma contagem de 0 até o número fornecido e depois é exibida uma mensagem final. Senão, apenas a mensagem final é exibida. Caso se forneça um argumento que não seja um valor inteiro válido ocorre um erro que é sinalizado como uma exceção *java.lang.NumberFormatException* que interrompe a execução do programa.

Já o *switch* é uma diretiva de desvio múltiplo de fluxo, isto é, baseado na avaliação de uma expressão ordinal é escolhido um caminho de execução dentre vários possíveis. Um resultado ordinal é aquele pertencente a um conjunto onde se conhece precisamente o elemento anterior e o posterior, por exemplo o conjunto dos números inteiros ou dos caracteres, ou seja, um valor dentro de um conjunto cujos valores podem ser claramente ordenados (0, 1, 2 .... no caso dos inteiros e 'A', 'B', 'C'... no caso dos caracteres).

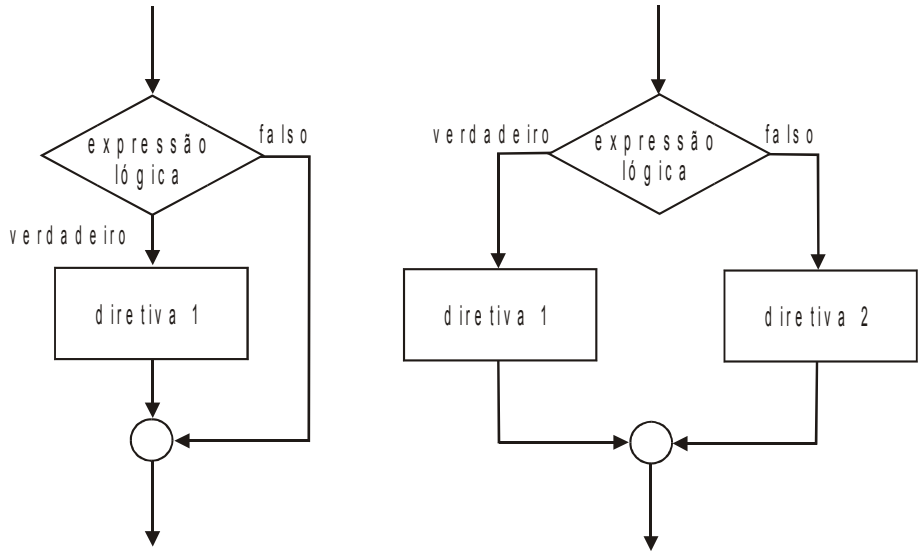

Figura 3. 2 – Comportamento da diretiva if

O *switch* equivale logicamente a um conjunto de diretivas *if* encadeadas, embora seja usualmente mais eficiente durante a execução. Na Figura 8 temos ilustrado o comportamento da diretiva *switc*h.

A sintaxe desta diretiva é a seguinte:

```
switch (expressão_ordinal)
{
 case ordinal1: diretiva3;
  break;
 case ordinal2: diretiva2;
  break;
 default: diretiva_default;
}
```

A expressão utilizada pelo *switch* deve necessariamente retornar um resultado ordinal. Conforme o resultado, seleciona-se um dos casos indicados pela construção case ordinal. As diretivas encontradas a partir do caso escolhido são executadas até o final da diretiva *switch* ou até que uma diretiva *break* encerre o *switc*h. Se o valor resultante não possuir um caso específico são executadas as diretivas *default* colocadas, opcionalmente, ao final da diretiva *switc*h.

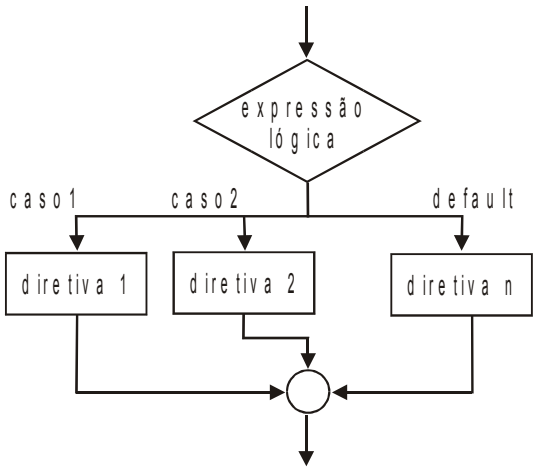

Figura 3. 3 – Comportamento da diretiva switch

Note que o ponto de início de execução é um caso *(cas*e) cujo valor ordinal é aquele resultante da expressão avaliada. Após iniciada a execução do conjunto de diretivas identificadas por um certo caso, tais ações só são interrompidas com a execução de uma diretiva *break* ou com o final da diretiva *switc*h. A seguir, temos uma aplicação simples que exemplifica a utilização das diretivas *switch* e *brea*k.

```
// exemploSwitch.java
public class exemploSwitch {
public static void main (String args[]) {
 if (args.length > 0){
    switch(args[0].charAt(0)) {
      case 'a':
      case 'A': System.out.println("Vogal A");
     break;
     case 'e':
      case 'E': System.out.println("Vogal E");
      break;
      case 'i':
      case 'I': System.out.println("Vogal I");
      break;
      case 'o':
      case 'O': System.out.println("Vogal O");
      break;
     case 'u':
     case 'U': System.out.println("Vogal U");
     break;
     default: System.out.println("Não é uma vogal");
     }
   } else {
    System.out.println("Não foi fornecido
argumento");
   }
 }
}
```

Exemplo 3. 4 - Utilização da Diretiva switch

No exemplo, quando é fornecido algum argumento, a aplicação identifica se o seu primeiro caractere é uma vogal. Se o argumento não for iniciado por vogal ocorre a execução da seção *default* da diretiva *switc*h. Caso não existam argumentos fornecidos, o programa imprime uma mensagem correspondente.

Imagine que a primeira letra seja a vogal 'i'. Após passar no teste condicional (args.length > 0), serão realizados as seguintes comparações no comando switch: 'A'=='i', 'a'=='i', 'E'=='i', 'e'=='i' e, por fim, 'i'=='i'. Como esta última comparação será verdadeira, serão executados os comandos que seguem esta linha, no caso o comando que irá exibir a mensagem "Vogal I". A execução destes comandos é, então interrompida no primeiro comando break que aparecer depois desta linha.

3.4 – Estruturas de Repetição Condicionais

As estruturas de repetição condicionais são estruturas de repetição cujo controle de execução é feito pela avaliação de expressões condicionais. Estas estruturas são adequadas para permitir a execução repetida de um conjunto de diretivas por um número indeterminado de vezes, isto é, um número que não é conhecido durante a fase de programação mas que pode ser determinado durante a execução do programa, tal como, um valor a ser fornecido pelo usuário, obtido de um arquivo ou ainda de cálculos realizados com dados alimentados pelo usuário ou lido de arquivos. Existem duas estruturas de repetição condicionais: *while* e *do while*.

O *while* é o que chamamos de laço condicional, isto é, um conjunto de instruções que é repetido enquanto o resultado de uma expressão lógica (uma condição) é avaliado como verdadeiro. Abaixo segue a sintaxe desta diretiva enquanto seu o comportamento é ilustrado a seguir.

```
while (expressão_lógica)
  diretiva;
```

Note que a diretiva *while* avalia o resultado da expressão antes de executar a diretiva associada, assim é possível que esta diretiva nunca seja executada caso a condição seja inicialmente falsa. Um problema típico relacionado a avaliação da condição da diretiva *while* é o seguinte: se a condição nunca se tornar falsa, o laço será repetido indefinidamente.

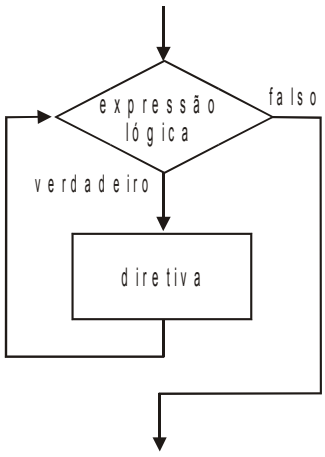

Figura 3. 4 – Comportamento da Diretiva while

O *do while* também é um laço condicional, isto é, tal como o *while* é um conjunto de instruções repetido enquanto o resultado da condição é avaliada como verdadeira mas, diferentemente do *while*, a diretiva associada é executada antes da avaliação da expressão lógica. e assim temos que esta diretiva é executada pelo menos uma vez. Segue a sintaxe da diretiva *do while* e uma ilustração do seu comportamento.

```
do
  diretiva
while (expressão_lógica);
```

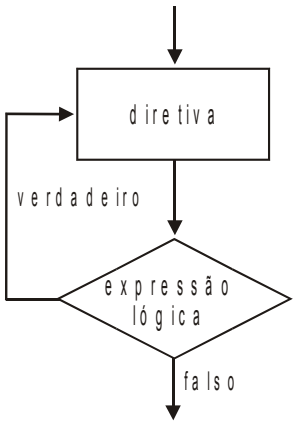

Figura 3. 5 – Comportamento da Diretiva do while

A seguir temos exemplos da aplicação destas duas diretivas de repetição.

```
public class exemploWhile {
public static void main (String args[]) {
 int j = 10;
 while (j > Integer.parseInt(args[0]))
     {
      System.out.println(""+j);
      j--;
     }
 }
}
```

Exemplo 3. 5 – Utilização da Diretiva while

O exemplo acima ilustra o uso da diretiva *while* tal como um laço de repetição simples equivalente a uma diretiva *for* como abaixo:

```
for (int j=0; j>Integer.parseInt(args[0]); j--)
   { System.out.println(""+j); }
```

A seguir um exemplo da diretiva *do while*.

```
public class exemploDoWhile {
public static void main (String args[]) {
 int min = Integer.parseInt(args[0]);
 int max = Integer.parseInt(args[1]);
 do {
     System.out.println("" + min + " < " + max);
     min++; max--;
    } while (min < max);
 System.out.println(""+ min +"<"+ max +"condicao
invalida");
 }
}
```

Exemplo 3. 6 – Utilização da Diretiva do while

3.5 – Desvios Incondicionais

Apesar dos projetistas da linguagem Java terem mantido a palavra `goto` como reservada, os mesmos decidiram não incluí-la na linguagem, já que, em geral, expressões contendo o comando `goto` não são consideradas boa prática de programação, pois comprometem consideravelmente a legibilidade de um programa.

Embora o uso irrestrito do `goto` venha sendo abolido nas linguagens de programação atuais, existem situações em que o uso destes comandos pode ser bastante benéfico. Em seu famoso artigo "Structured Programming with goto's", Donald Knuth defende o uso de goto´s para a realização de desvios ocasionais durante a execução de laços. Os projetistas da linguagem Java parecem concordar com Knuth e até incluíram, além dos comandos break e continue de C e C++, novas diretivas na linguagem para suportar este estilo de programação, o break e o continue rotulados.

O mesmo comando break que você usou para interromper a execução de um switch também pode ser usado para interromper um laço. Neste exemplo, trataremos primeiramente o break não rotulado:

```
while (ano <= 2001)
     {
      saldo = (saldo + salario) * (1 + juros);
      if (saldo >= saldoLimite) break;
      ano++;
     }
```

No exemplo acima, o laço while é interrompido caso ocorra a condição ano > 2001 no início do laço ou caso ocorra a condição saldo >= saldoLimite no meio do laço.

Você pode controlar o fluxo de execução de qualquer laço através do uso dos comandos *break* e *continue*. Como foi dito anteriormente, o comando *break* abandona um laço sem que o restante dos comandos da iteração corrente sejam executados. Já o comando *continue* interrompe a execução da iteração corrente e retorna ao início do laço para iniciar a próxima iteração. O programa abaixo mostra exemplos do uso do *break* e *continue* nos laços *for* e *while*:

```
public class BreakContinue {
  public static void main(String[] args) {
    for(int i = 0; i < 100; i++) {
      if(i == 74) break; // saÌda do loop
      if(i % 9 != 0) continue; // prÛxima iteração
      System.out.println(i);
    }
    int i = 0;
    // Um Loop infinito
    for(;;) {
      i++;
      if(j == 9621) break; // saida 1 do loop
      int j = i * 27;
      if(j == 1269) break; // saida 2 do loop
      if(i % 10 != 0) continue; // Topo do loop
      System.out.println(i);
    }
  }
}
```

Exemplo 3. 7 – Uso do break e do continue em laços

No laço *for* do exemplo acima, o valor de i nunca chegará a 100 já que o comando *break* abandona o laço assim que o valor de i for igual a 74. Normalmente você usaria um *break* como esse em situações em que a expressão condicional devesse ser avaliada no meio da interação. O comando *continue* abandona a iteração corrente e retorna para o início da próxima iteração do laço (incrementando o valor de i) sempre que o valor de i não for divisível por 9. Quando esta condição for satisfeita, então o valor de i será impresso.

A outra porção do exemplo mostra um "laço infinito", que em, em teoria, poderia continuar sua execução indefinidamente. Entretanto, dentro do laço existe um comando break que interromperá a execução do laço. Além disso você verá o comando continue que retornará a iteração do laço ao início do mesmo sem que sejam executadas as instruções restantes.

Uma segunda maneira de realizar um loop infinito é com *while(true)*. O compilador trata os comandos *while(true)* e *for(;;)* da mesma forma, sendo que o uso de cada um é definido apenas pelo gosto do programador.

Diferentemente de C/C++, Java também oferece o *break* rotulado, que lhe permite interromper a execução de laços aninhados em vários níveis. A razão para isso é simples: ocasionalmente alguma condição "estranha" pode ocorrer dentro de um laço aninhado a outros, e neste caso você poderia querer suspender a execução de todos estes laços. Uma outra forma, mais tortuosa, de se resolver este problema seria a inclusão de testes condicionais extras em todos os laços que estão aninhados (lembre que estes laços podem ser aninhados em vários níveis).

O exemplo seguinte mostra esta funcionalidade. Note que o identificador deve preceder o laço mais externo ao qual você deseja interromper.

```
public class breakRotulado {
  public static void main(String[] args) {
    int i = 0;
    externo:
    for(; true ;) { // laço infinito
      interno:
      for(; i < 10; i++) {
        System.out.println("i = " + i);
        if(i == 2) {
          prt("continue");
          continue;
        }
        if(i == 3) {
          System.out.println("break");
          i++;
          break;
        }
        if(i == 7) {
          System.out.println ("continue externo");
          i++;
          continue externo;
        }
        if(i == 8) {
          System.out.println ("break externo");
          break externo;
        }
        for(int k = 0; k < 5; k++) {
          if(k == 3) {
            System.out.println ("continue interno");
            continue interno;
          }
        }
      }
    }
    // não é possÌvel usar break ou continue
    // para os rÛtulos neste ponto do programa
  }
  static void prt(String s) {
    System.out.println(s);
  }
}
```

Exemplo 3. 8 - Uso do break e continue rotulados

É importante lembrar que a única situação em que se indica o uso de desvios rotulados em Java, é quando se tem laços aninhados e se deseja interromper ou continuar a execução desses laços em mais de um nível de aninhamento.

No paper “goto considered harmful”, Dijkstra opõe-se ao uso de rótulos e não ao uso de goto’s. Foi observado que o número de bugs em um programa tende a aumentar com o aumento do número de rótulos. Rótulos e desvios com goto comprometem a legibilidade dos programas, já que introduzem desvios não estruturados em seus

códigos (programação macarrônica). Note que os desvios rotulados de Java não apresentam este tipo de problema, posto que os mesmos não podem ser usados de modo indiscriminado por parte do programador para desviar o fluxo de um programa, o que mantém a estruturação dos mesmos. Note queeste é um caso onde uma característica da linguagem torna-se mais útil pela restrição do uso de uma diretiva.

3.6 – Tratamento de Exceções

Exceções são situações especiais que podem ocorrer durante a execução dos programas. A forma mais comum de exceções são erros de execução, tal como, tentativa de divisão por zero. Java oferece diretivas para tratamento de exceções muito semelhantes às estruturas existentes na linguagem C++. Elas tem o propósito de evitar que o programador tenha de realizar testes de verificação e avaliação antes da realização de certas operações, desviando automaticamente o fluxo de execução para as rotinas de tratamento. Através destas diretivas, delimita-se um trecho de código que será monitorado automaticamente pelo sistema. A ocorrência de exceções no Java é sinalizada através de objetos especiais que carregam informação sobre o tipo de exceção detectada.

Existem várias classes de exceções adequadas para o tratamento das situações especiais mais comuns em Java. Além disso, exceções podem ser criadas em adição às existentes ampliando assim as possibilidades de tratamento de situações especiais. Com o uso das diretivas de tratamento de exceções, a ocorrência de um ou mais tipos de exceções dentro do trecho de código delimitado desvia a execução automaticamente para uma rotina designada para o tratamento específico destas exceções.

Você terá a oportunidade de se aprofundar no tema de tratamento de exceções em um capítulo subsequente deste texto. Por enquanto, apresentaremos apenas rapidamente as diretivas *try catch* e *throws* porque elas serão necessárias em alguns exemplos posteriores a esta seção e anteriores ao capítulo de tratamento de exceções.

A sintaxe do comando *try catch* é mostrada a seguir:

```
try {
 diretiva_normal;
} catch (exception1) {
 diretiva_de_tratamento_de erro1;
} catch (exception2) {
 diretiva_de_tratamento_de erro2;
}
```

O bloco que se inicia após a palavra *try* delimita o trecho de código onde pode ocorrer uma exceção. Caso ocorra uma exceção neste trecho, a execução deste bloco será interrompida e o fluxo de controle será passado para um dos blocos que se iniciam após a palavra *catch*. O bloco que será ativado dependerá do tipo de exceção que ocorrer. No caso de não haver tratador para a exceção ocorrida, ela é propagada automaticamente.

No caso da ocorrência de uma exceção que não é devidamente tratada dentro de um método, este deverá propagá-la obrigatoriamente. Para tanto é necessário utilizar a diretiva *throws* no cabeçalho do método. Na linha de código abaixo, a exceção padrão java.io.IOException não é tratada em `main`.

```
public static void main(String args[])
   throws Java.io.IOException  {  …  }
```

3.7 – Entrada e Saída pelo Console

Através da linguagem Java é possível construirmos aplicações de console, isto é, aplicações que utilizam os consoles de operação dos sistemas operacionais para a interação com o usuário, ou seja, para a realização das operações de entrada de dados fornecidos pelo usuário e a exibição de mensagens do programa. Como console de operação ou console entendemos as seções ou janelas em modo texto (que são capazes de exibir apenas texto) de sistemas *Uni*x, as janelas *"Prompt* do MS-DOS" existentes no *Windows* e outras semelhantes existentes nos demais sistemas operacionais.

As aplicações de console, embora em desuso para a construção de muitos, podem ser utilizadas para a criação de programas utilitários do sistema, tal como certos comandos do sistema operacional que são, na verdade, programas específicos disponíveis apenas para utilização nos consoles.

3.7.1– Saída de Dados no Console

As aplicações de console utilizam como padrão a *stream* de dados **out**, disponível através da classe **java.lang.System**. Uma *stream* pode ser entendida como um duto capaz de transportar dados de um lugar (um arquivo ou dispositivo) para um outro lugar diferente. O conceito de *stream* é extremamente importante pois é utilizado tanto para manipulação de dados existentes em arquivos como também para comunicação em rede e outros dispositivos.

A *stream* de saída padrão é aberta automaticamente pela máquina virtual Java ao iniciarmos uma aplicação e permanece pronta para enviar dados. A saída padrão está tipicamente associada ao dispositivo de saída *(displa*y) ou seja, a janela de console utilizada pela aplicação conforme designado pelo sistema operacional. Enquanto a *stream* de saída **out** é um objeto da classe **java.io.PrintStream** a *stream* de entrada **in** é um objeto da classe **java.io.InputStream**.

A *stream* de saída **out** pode ser utilizada através dos seguintes métodos da classe **java.io.PrintStream**:

Método	Descrição
`print(char)`	Imprime um caractere.
`print(boolean)`	Imprime um valor lógico.
`print(int)`	Imprime um inteiro.
`print(long)`	Imprime um inteiro longo.
`print(float)`	Imprime um valor em ponto flutuante simples.
`print(double)`	Imprime um valor *doubl*e.
`print(String)`	Imprime uma *strin*g.
`println()`	Imprime o finalizador de linha.
`println(char)`	Imprime um caractere e o finalizador de linha.
`println(boolean)`	Imprime um valor lógico e o finalizador de linha.
`println(int)`	Imprime um inteiro e o finalizador de linha.
`println(long)`	Imprime um inteiro longo e o finalizador de linha.
`println(float)`	Imprime um valor em ponto flutuante simples e o finalizador de linha.
`println(double)`	Imprime um valor *double* e o finalizador de linha.
`println(String)`	Imprime uma *string* e o finalizador de linha.

Tabela 3. 1 - Métodos do Objeto java.io.PrintStream

Como foi feito em vários dos exemplos anteriores para enviarmos dados para a saída padrão devemos utilizar a construção:

```
System.out.print(valor);
System.out.println(valor);
```

A saída padrão pode ser redirecionada para outro dispositivo, ou seja, podemos especificar um arquivo ou um canal de comunicação em rede para receber os dados que seriam enviados ao console. Para isto pode-se utilizar o método estático **setOut** da classe **java.lang.System**:

```
System.setOut(novaStreamDeSaida);
```

O redirecionamento da saída padrão pode ser útil quando queremos que um programa produza um arquivo de saída ou envie sua saída para um outro dispositivo.

3.7.2– Entrada de Dados pelo Console

Da mesma forma que toda aplicação de console possui uma *stream* associada para ser utilizada como saída padrão, existe uma outra *stream* denominada de entrada padrão, usualmente associada ao teclado do sistema. Esta *strea*m, de nome **in**, disponível através da classe **java.lang.System** e pertencente a classe **java.io.InputStream** é também aberta automaticamente quando a aplicação é iniciada pela máquina virtual Java permanecendo pronta para fornecer os dados digitados. Devemos observar que, embora pronta para receber dados dos usuários, os métodos disponíveis para a entrada destes dados são bastante precários. Veja na Tabela 3.2 os principais métodos disponíveis na classe **java.io.InputStream**.

Método	Descrição
`available()`	Retorna a quantidade de bytes que podem ser lidos sem bloqueio.
`read()`	Lê um byte.
`read(byte[])`	Preenche o vetor de bytes fornecido.
`skip(long)`	Descarta a quantidade de bytes especificada da entrada.

Tabela 3. 2 - Métodos da Classe java.io.InputStream

Como se pode observar, a entrada de dados através de objetos **InputStream** é feita através da leitura de bytes. Contudo, o que digitamos no teclado são caracteres. Cada caracter digitado corresponde a um código na tabela UNICODE (coincidente com o código da tabela ASCII). Cada caracter é, portanto, convertido para byte. Isto é pouco confortável quando se pretende ler valores numéricos (inteiros ou em ponto flutuante), *strings* ou qualquer outra informação diferente de caracteres, pois exige que os vários caracteres fornecidos pelo usuário sejam concatenados e testados para a formação de um valor ou *strin*g.

A seguir, mostra-se um exemplo de aplicação capaz de efetuar a leitura de caracteres isolados entendidos como conceitos de A até E, equivalentes a notas de 5 a 1, respectivamente:

```
// Media.java
public class Media {
  public static void main(String args[]) throws
   java.io.IOException {
   float notaMedia = 0;
   int contagem = 0;
   int valor;
    System.out.println("Forneca como notas conceitos"
+
                " A - E.");
   System.out.println("Um traco '-' encerra a " +
                "entrada de notas.");
   System.out.println("Digite um valor: ");
   while ('-' != (valor = System.in.read())) {
     notaMedia += 'F' - valor;
     contagem++;
     System.in.skip(2);
     System.out.println("Digite um valor: ");
   }
   notaMedia /= contagem;
    System.out.println("Foram fornecidos " + contagem
+
               " valores.");
   System.out.println("Nota media = " + notaMedia);
   System.out.println("Conceito medio = " +
    (char) (70 - Math.round(notaMedia)));
  }
}
```

Exemplo 3. 9 - Entrada de Dados pelo Console

Nesta aplicação existem vários detalhes que devem ser comentados:

• O método **read**, da *stream* de entrada padrão **in**, quando não consegue tratar a entrada fornecida (a digitação de dois caracteres por exemplo), lança a exceção **java.io.Exception** mas como optamos por não tratá-la via *try catc*h, o método **main** propaga esta exceção.

• Cada caractere lido é na verdade um byte, que pode ser tratado numericamente. Os Caracteres maiúsculos A, B, C ... são representados pelos códigos 65, 66, 67, etc. A manipulação algébrica transforma tais códigos em valores de 5 a 1:

```
notaMedia += 'F' - valor;
```

O inverso também é possível:

```
(char) (70 - Math.round(notaMedia))
```

• O "enter" pressionado após a digitação de cada caractere deve ser extraído da *stream* de entrada. Como equivale a dois caracteres (uma sequência de "\r\n") usamos o método **skip**.

A aplicação não trata quaisquer tipos de erros, nem sequer o uso de caracteres minúsculos ao invés de maiúsculos (uma boa sugestão para um exercício adicional). A dificuldade de tratar a entrada de dados em aplicações de console é uma clara indicação

de que os projetistas do Java concentraram seus esforços para tornar esta plataforma mais adaptada para o desenvolvimento de aplicações gráficas destinadas as GUIs *(Graphical User Interface*s) dos diversos sistemas operacionais existentes.

Para contornar os problemas de leitura de valores dos tipos primitivos e também de *string*s, apresentamos a classe **Console** que contém nove métodos para leitura de valores destes tipos.

Método	**Descrição**
`readBoolean()`	Lê um valor *boolean* da entrada padrão.
`readByte()`	Lê um valor *byte* da entrada padrão.
`readShort()`	Lê um valor *short* da entrada padrão.
`readInteger()`	Lê um valor *int* da entrada padrão.
`readLong()`	Lê um valor *long* da entrada padrão.
`readFloat()`	Lê um valor *float* da entrada padrão.
`readDouble()`	Lê um valor *double* da entrada padrão.
`readChar()`	Lê um valor *char* da entrada padrão.
`readString()`	Lê uma *string* da entrada padrão.

Tabela 3. 3 - Métodos da Classe Console

Abaixo o código da classe Console onde foi utilizada a classe **java.io.BufferedReader** para prover a facilidade da leitura de uma linha da entrada de cada vez. Todos os métodos oferecidos fazem um tratamento mínimo de erros de forma que valores inválidos são retornados como valores nulos.

```
// Console.java

import java.io.*;

public final class Console {

  public static boolean readBoolean () {
      try {
         BufferedReader br = new BufferedReader (
                new InputStreamReader (System.in) );
         String s = br.readLine ();
         Boolean b = new Boolean (s);
         return b.booleanValue ();
      } catch (IOException e) {
         return false;
      }
  }
```

```
public static byte readByte () {
      try {
         BufferedReader br = new BufferedReader (
                new InputStreamReader (System.in) );
         String s = br.readLine ();
         return Byte.parseByte (s);
      } catch (IOException e) {
         return 0;
      } catch (NumberFormatException e) {
         return 0;
      }
  }

  public static short readShort () {
      try {
         BufferedReader br = new BufferedReader (
                new InputStreamReader (System.in) );
         String s = br.readLine ();
         return Short.parseShort (s);
      } catch (IOException e) {
         return 0;
      } catch (NumberFormatException e) {
         return 0;
      }
  }

 public static int readInteger () {
      try {
         BufferedReader br = new BufferedReader (
                new InputStreamReader (System.in) );
         String s = br.readLine ();
         return Integer.parseInt (s);
      } catch (IOException e) {
         return 0;
      } catch (NumberFormatException e) {
         return 0;
      }
  }

 public static long readLong () {
      try {
         BufferedReader br = new BufferedReader (
                new InputStreamReader (System.in) );
         String s = br.readLine ();
         return Long.parseLong (s);
      } catch (IOException e) {
         return 0;
      } catch (NumberFormatException e) {
         return 0;
      }
  }
```

```
  public static float readFloat () {
      try {
         BufferedReader br = new BufferedReader (
                new InputStreamReader (System.in) );
         String s = br.readLine ();
         Float f = new Float (s);
         return f.floatValue ();
      } catch (IOException e) {
         return 0;
      } catch (NumberFormatException e) {
         return 0;
      }
  }

  public static double readDouble () {
      try {
         BufferedReader br = new BufferedReader (
                new InputStreamReader (System.in) );
         String s = br.readLine ();
         Double d = new Double (s);
         return d.doubleValue ();
      } catch (IOException e) {
         return 0;
      } catch (NumberFormatException e) {
         return 0;
      }
  }

  public static char readChar () {
      try {
         return ((char) System.in.read());
      } catch (IOException e) {
         return '\0';
      }
  }

  public static String readString () {
      try {
         BufferedReader br = new BufferedReader (
                new InputStreamReader (System.in) );
         String s = br.readLine ();
         return s;
      } catch (IOException e) {
         return "";
      }
  }

}
```

Exemplo 3. 10 - A Classe Console

Os métodos da classe Console podem ser chamados através do nome da classe Console, como mostra o exemplo seguinte, que lê valores dos tipos primitivos e *string*, mas de forma bastante robusta.

```
// EntradaPadrao.java
import java.io.*;

public class EntradaPadrao {
   public static void main(String[] args) {
        System.out.print ("Boolean: ");
        System.out.println(Console.readBoolean());
        System.out.print ("Float: ");
        System.out.println(Console.readFloat());
        System.out.print ("Double: ");
        System.out.println(Console.readDouble());
        System.out.print ("Integer: ");
        System.out.println(Console.readInteger());
        System.out.print ("String: ");
        System.out.println(Console.readString());
        System.out.print ("Short: ");
        System.out.println(Console.readShort());
        System.out.print ("Long: ");
        System.out.println(Console.readLong());
        System.out.print ("Byte: ");
        System.out.println(Console.readByte());
        System.out.print ("Char: ");
        System.out.println(Console.readChar());
  }
}
```

Exemplo 3. 11 - Aplicação da Classe Console

3.8 Exercícios

1. Faça um programa que leia um valor inteiro e diga se o número é par ou ímpar.
2. Faça um programa que leia três números inteiros, verifique se formam um triângulo.Em caso positivo, indique se o triângulo é equilátero, isósceles ou escaleno.
3. Faça um programa que imprima os n primeiros números primos, sendo n lido.
4. Faça um programa que calcule o máximo divisor comum de dois números lidos.
5. Faça um programa que leia dois números x e y, e calcule a potência x^y , onde tanto x quanto y podem ser positivos, negativos ou zero.
6. Faça um programa para ler uma sequência de números inteiros e imprimí-los. O programa deve ser interrompido quando o número lido for zero. Na redação do programa você não deve testar a condição de parada em mais do que um ponto do programa. Além disso, você também não deve escrever a operação de leitura em mais do que um único ponto do programa.
7. Faça um programa que use as características do comando switch para calcular o preço de um automóvel com diversos tipos de opcionais. Considere que o automóvel é vendido em cinco categorias distintas (numeradas de um a cinco), onde cada categoria inclui todos os opcionais da categoria imediatamente inferior (a de número menor). A entrada do programa corresponde a categoria do automóvel cujo preço

será calculado. Você deve fazer o programa de tal maneira que, ao alterar-se o preço dos opcionais de uma categoria, só seja necessário alterar no programa o trecho de código relacionado com os opcionais desta categoria, isto é, não se deve ter de alterar os trechos de código associados com o preço das categorias superiores a ela.

8. Faça um programa que use o comando switch para ler um número entre 1 e 5 e imprimir o número escrito por extenso.
9. Faça um programa para ler um número inteiro positivo e calcular a tabuada de multiplicação de 0 a 9 de todos os números positivos inferiores ao número lido. Sempre que o resultado de uma multiplicação for divisível por dez, o programa deve perguntar ao usuário se ele deseja continuar com o cálculo da tabuada. Portanto, o programa deve se encerrar ao final do cálculo da tabuada ou quando o usuário responder que não deseja continuar o cálculo.