ORIENTAÇÃO A OBJETOS

AULA 11

Generics, Exceções e Comparadores

Vandor Roberto Vilardi Rissoli

APRESENTAÇÃO

- *Generics*
- Tratamento de Exceções
- Comparadores
  - *Comparable*
  - *Comparator*
- Referências

Generics

Atente aos momentos de instanciação dos objetos que são genéricos, onde a partir de suas definições eles são usados nos testes de conversão (*typechecking*) durante todo o processo de compilação, fornecendo mais confiabilidade ao código elaborado.

Declarando uma Classe com Generics

```
public class ClasseGenerica <A> {
  private A valor;

  public ClasseGenerica(A valor) {
    this.valor = valor;
  }

  public A getValor() {
    return valor;
  }
}
```

Generics

Para testar a classe genérica anterior será usada a classe abaixo com diferentes métodos que trabalham com tipos diferentes e usam referida classe genérica.

```
public class UsaGenerica {
  // Método com parâmetro String
  public String metodo(String entrada) {
    String valor = entrada;
    ClasseGenerica<String> gene = new
                        ClasseGenerica<String>(valor);
    String outroValor = gene.getValor();
    return outroValor;
  }
  // Método com parâmetro Integer
  public Integer metodo(int entrada) {
    Integer valor = new Integer(entrada);
    ClasseGenerica<Integer> gene = new
                       ClasseGenerica<Integer>(valor);
    Integer outroValor = gene.getValor();
    return outroValor;
  }
```

Generics

```
// continuação do exemplo anterior

  // Método com parâmetro Double
  public Double metodo(double entrada) {
    Double valor = new Double(entrada);
    ClasseGenerica<Double> gene = new
                        ClasseGenerica<Double>(valor);
    Double outroValor = gene.getValor();
    return outroValor;
  }

  public static void main(String[] args) {
    UsaGenerica amostra = new UsaGenerica();
    System.out.println("Uma String para Generics =\t\t"
                           + amostra.metodo("Bom dia!"));
    System.out.println("Um Inteiro para Generics =\t\t"
                           + amostra.metodo(12345678));
    System.out.println("Um Real (double) para Generics"
                  + " =\t" + amostra.metodo(987.654321));
  }
}
```

→ Uma única classe trabalhando com diversos tipos

Generics

Os Generics só trabalham com tipos referenciáveis, sendo limitante o uso de tipos primitivos. Porém, é possível restringir os tipos possíveis para um Generics através da limitação de seu domínio pelas subclasses, por exemplo:

```
public class MeuCarro <Carro extends Veiculo>  {
    :
}
```

O trabalho do Generics com Coleções pode ser compreendido com a observação dos exemplos comparativos a seguir.

Generics

```
/** Síntese
 *    Objetivo: mostrar idade dos amigos cadastrados
 *    Entrada:  nenhuma (só atribuições)
 *    Saída:    todos as idades registrada na lista
 */
import java.util.*;
  public class GenericoIdades {
    public static void main(String[] args) {
      List lista = new ArrayList();
      lista.add(new Integer(21));
      lista.add(new Integer(18));
      lista.add(new Integer(32));
      lista.add(new String("27")); // exceção na execução
      // exceção para proteger a lista só de Integer
      System.out.println("LISTA DESORDENADA\n" + lista);
      Collections.sort(lista);  // colocando em ordem
      System.out.println("\nLISTA ORDENADA\n" + lista);
  }
}
```

Generics

```
/** Síntese
 *    Objetivo: mostrar idade dos amigos cadastrados
 *    Entrada:  nenhuma (só atribuições)
 *    Saída:    todos as idades registrada na lista
 */
import java.util.*;
public class RealmenteGenericoIdades {
  public static void main(String[] args) {
    List<Integer> lista = new ArrayList<Integer>();
    lista.add(new Integer(21));
    lista.add(new Integer(18));
    lista.add(new Integer(32));
    lista.add(new String("27")); // exceção na compilação
    System.out.println("LISTA DESORDENADA\n" + lista);
    Collections.sort(lista);
    System.out.println("\nLISTA ORDENADA\n" + lista);
  }
}
```

Exercício de Fixação

1) Elabore um programa que permita ao usuário cadastrar quantos nomes completos de pessoa ele quiser e quando terminar este cadastro apresente todos os nomes cadastrados em formato de listagens na console, respeitando as seguintes ordenações:

   a) na ordem de inserção;

   b) na ordem inversa da inserção.

Sua solução deverá usar os recursos de `List` e `Generics`, além de um método que salte 10 linhas em branco entre a apresentação desta diferentes listagens. Resolva esta solução de forma simples, só não permitindo que os nomes informados sejam nulos.

Tratamento de Exceções

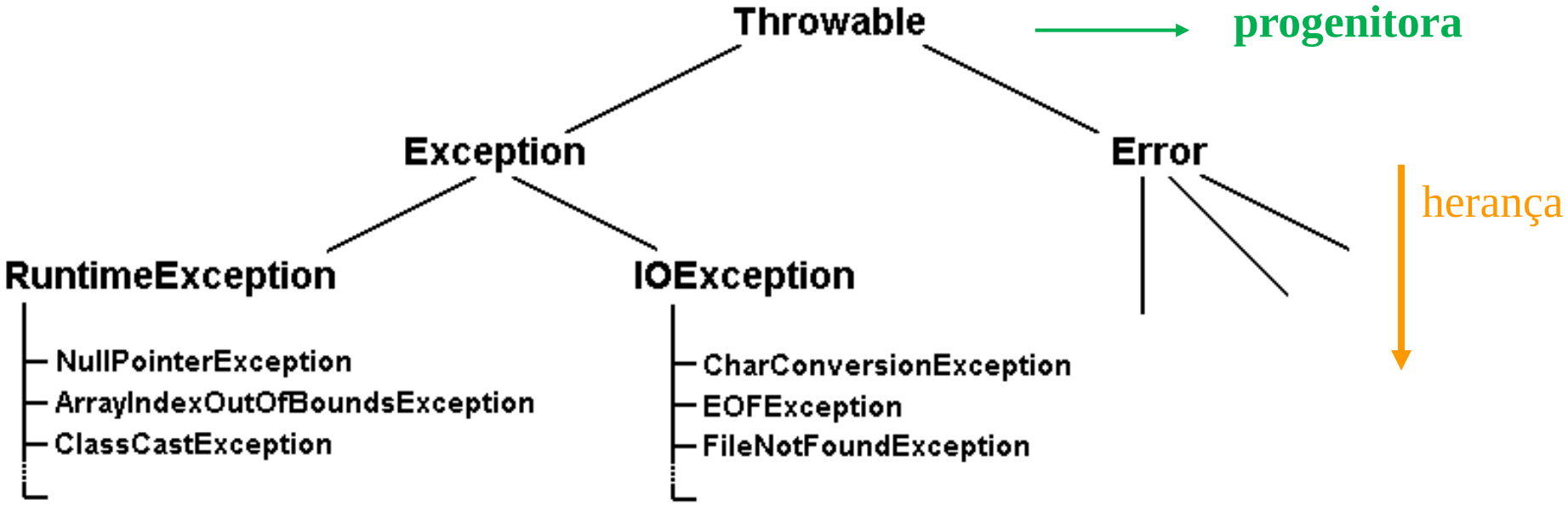
A ocorrência de uma situação inesperada no fluxo normal de processamento pode acontecer durante a execução de um programa, geralmente deixando seus usuários descontentes e até prejudicando as atividades realizadas até aquele momento inoportuno.

Estas ocorrências são tratadas de maneira especial em Java, sendo denominadas de **Tratamento de Exceção**.

As exceções se referem aos erros que podem ser gerados durante a execução de um programa, sendo aplicada a esta situação um conjunto de recursos Java que permitirão a manipulação e o tratamento da maioria destas possíveis situações.

Tratamento de Exceções

Retomando este recurso em Java, com mais conhecimento sobre a Programação Orientada a Objeto é possível compreender melhor o Tratamento de Exceções.



O tratamento das exceções se concentra, quase que totalmente, na Exception, pois praticamente a totalidade da Error esta fora do controle da aplicação, sendo interessante o encerramento de seu processo.

Tratamento de Exceções

Exceção em Métodos (*throws*)

A elaboração de métodos em Java pode ou não retornar um valor, mas deve sempre permitir a indicação de suas exceções (erros), de **tipo verificadas,** que podem acontecer durante sua execução.

Similar ao seu tipo de retorno, estas possíveis exceções devem ser inseridas no cabeçalho de seus métodos através da instrução throws. É importante destacar que se uma exceção ocorrer em um método sua execução será interrompida e um objeto desta exceção será retornado ao seu acionador (**PROPAGANDO** a exceção), sem o retorno do tipo de dado esperado deste método.

```
public String acessaDados() throws IOException {
    :
}
```

Tratamento de Exceções

Com a ocorrência desta exceção o sistema de tempo de execução irá iniciar uma busca por um manipulador de exceções que saiba tratar esta ocorrência específica.

A necessidade de tratar várias exceções em um método pode ser realizada com a especificação de cada uma delas na clausula throws separadas por vírgula, exemplo:

```
public String acessaDados() throws  EOFException,
                                    FileNotFoundException, {
   :
}
```

Apesar de ser possível, a propagação de exceções em métodos não deve envolver exceções provenientes da subclasse Error, pois elas estão fora do controle de seu programa, e da subclasse RuntimeException que são de responsabilidade do programador em resolve-las.

Tratamento de Exceções

Para evitar que o compilador Java não permita a execução de um programa, todos os métodos que tenham possibilidade de exceções do **tipo verificadas** precisam especificar em seu cabeçalho todas estas possíveis exceções para lançamento.

As subclasses, de uma superclasse que lance algumas exceções, não podem possuir mais exceções que sua superclasse, onde a inexistência de exceções na superclasse impossibilita que suas subclasses lancem qualquer exceção em seus métodos.

Aos métodos que não possuam a instrução throws não será possível lançar qualquer tipo de **exceção verificada**, apesar do compilador Java exigir seu tratamento em algumas situações, por exemplo na abertura de arquivos físicos de dados.

Tratamento de Exceções

```
// exemplo instrutivo sobre conteúdo
// a ser estudado posteriormente (arquivo)
public String lerDados(BufferedReader entra) throws
                                                EOFException
{
    :
  while (...) {
    if (arquivo == -1) // EOF chegou ao fim do arquivo
      if (aux < cont)
        throw new EOFException("Operação inválida.");
  }
    :
  return str;
}
```

- Representação de um método que prevê possível exceção de final de arquivo (EOF) em seu cabeçalho
- Instrução throw lançando uma exceção deste tipo

Tratamento de Exceções

Instrução (***throw***)

Embora as sintaxes sejam parecidas, a instrução throw é diferente de throws, sendo ambas utilizadas para funcionalidades diferentes na programação Java.

A throw dispara ou força a execução de uma exceção, por exemplo na instrução throw do exemplo anterior.

⇒ EOFException é dispara sem a existência da instrução try...catch para captura-la

```
throw new EOFException();
```

- Cria um objeto EOFException
- Dispara esta exceção, independente dela realmente ter acontecido ou não, por meio da instrução throw

Tratamento de Exceções

As classes e subclasses relacionadas as exceções (Throwable) podem ser estendidas, gerando exceções novas e particulares as necessidades de determinada aplicação, por exemplo:

```
public class ExcecaoTamanho extends IOException {
  ExcecaoTamanho(String mensagem){
    // codificação do construtor
  }
}
    :
public String lerDados(BufferedReader entra) throws
                                          ExcecaoTamanho
{   :
  while (...) {
   if (arquivo == -1) // EOF chegou ao fim do arquivo
      if (aux < cont)
         // dispara exceção particular
         throw new ExcecaoTamanho("Operação inválida.");
  }
   :
}
```

Tratamento de Exceções

Observe o exemplo a seguir que cria uma exceção particular e faz uso da throw para dispará-la em uma situação que Java não trata como exceção – divisão por zero envolvendo valores reais (resultado é infinito - erro)

```
/** Síntese
 *    Objetivo: divisão reais com exceção própria
 *    Entrada:  dois números reais
 *    Saída:    resultado da divisão
*/
import java.text.DecimalFormat;
import javax.swing.JOptionPane;
class ExcecaoParticular extends RuntimeException {
  public ExcecaoParticular(String erro) { // construtor
    super(erro);
  }
}
public class MinhasExcecoes {
  public static void main(String[] args) {
    float dividendo=0,divisor, resultado = 0;
    String valor1, valor2;
    char controle = 'o';
    DecimalFormat forma = new DecimalFormat("0.000");
```

Tratamento de Exceções

```
// continuação do exemplo anterior
    valor1 = JOptionPane.showInputDialog(null,
         "Digite o número do dividendo.","Pergunta",
                         JOptionPane.QUESTION_MESSAGE);
    valor2 = JOptionPane.showInputDialog(null,
         "Informe o valor do divisor.","Pergunta",
                         JOptionPane.QUESTION_MESSAGE);
    try {
        dividendo = Float.parseFloat(valor1);
        if (valor2.charAt(0)=='0'|| valor2.equals(".0"))
          throw new ExcecaoParticular("Erro");
        else {
          divisor = Float.parseFloat(valor2);
          resultado = dividendo / divisor;
        }
    } catch (ExcecaoParticular excecao) {
       do {
           valor2 = JOptionPane.showInputDialog(null,
              "Divisão por zero não é possível.\n" +
            "Informe o divisor novamente.","Pergunta",
                              JOptionPane.ERROR_MESSAGE);
       } while (valor2.charAt(0)=='0' ||
           valor2.equals(".0") || valor2.equals(",0"));
```

Tratamento de Exceções

```
// continuação do exemplo anterior
        divisor = Float.parseFloat(valor2);
        resultado = dividendo / divisor;
    } catch (NumberFormatException excecao) {
        JOptionPane.showMessageDialog(null,
                "Valor numérico incorreto.","Alerta",
                            JOptionPane.WARNING_MESSAGE);
        controle = 'x';
    } finally {
      if (controle != 'x')
         JOptionPane.showMessageDialog(null,
                     "O resultado da divisão é \n" +
               forma.format(resultado), "Informação",
                               JOptionPane.PLAIN_MESSAGE);
      else
         JOptionPane.showMessageDialog(null,
                  "O cálculo não pode ser efetuado.",
                "Informação",JOptionPane.PLAIN_MESSAGE);
    }
  }
}
```

Tratamento de Exceções

throws e/ou try...catch

Embora seja tentador propagar as exceções, a responsabilidade do desenvolvedor é resolver o problema quando lhe é possível. No entanto, ele pode estar em uma situação que não lhe seja possível resolver adequadamente uma ou outra exceção, sendo importante passá-la para seu acionador, conscientemente, saber desta possibilidade de ocorrência e tratá-la mais adequadamente possível.

As implementações em Java permitem que um código trabalhe com as duas situações (propagar e/ou capturar) de tratamento de exceções (throws e try...catch), conforme seja conveniente ao programa que está sendo elaborado, desde que respeitando as definições Java.

Tratamento de Exceções

Porém, as exceções do tipo **verificadas** exigem seu tratamento, podendo serem propagadas ou capturas.

Caso não seja implementado nenhum dos dois possíveis tratamentos no programa o compilador acusará erro durante a compilação. Um exemplo desta situação seria a elaboração de um método que abra um arquivo físico (arquivo.txt) para leitura de dados em Java.

```
public static void abreArquivo () {
  new java.io.FileReader("arquivo.txt"); // acessa arquivo
}
```

A codificação anterior não compila e solicita ao seu programador que trate a exceção de uma das duas formas possíveis. Este tratamento é verificado e exigido porque é possível que este arquivo não exista, entre outras situações passíveis de erro na manipulação de arquivos de dados, o que resultará em uma exceção.

Tratamento de Exceções

O tratamento somente com o throws ou com o try...catch poderia ser suficiente para resolver este problema.

No intuito de uma representação genérica para a possibilidade de tratamento usando as duas formas em um mesmo método é apresentado o esquema abaixo:

```
public void acessa(String arquivo) throws IOException {
    :
  try {
   :      // código que pode lançar IOException
  } catch (SQLException excecao) {
   :      // código para tratamento desta exceção
  }
   :
} // termina o método
```

→ É importante ressaltar que é desnecessário declarar na throws as exceções que são do **tipo não verificadas**.

Tratamento de Exceções

Acompanhamento de Erros/Exceções

A obtenção de detalhes sobre as exceções ocorridas na execução de uma aplicação pode definir o tempo de sua manutenção e correção.

Em Java é possível averiguar várias informações sobre a ocorrência de exceções, sendo, normalmente, empregado neste levantamento os métodos getMessage e printStackTrace.

- getMessage – obtém a string relacionada a exceção ocorrida

- printStackTrace – mostra o tipo de exceção gerada, qual linha de sua ocorrência, entre outros dados

Tratamento de Exceções

```
/** Síntese
 *    Objetivo: mostrar detalhes de erro/exceção
 *    Entrada:  sem entrada
 *    Saída:    mensagem de erro e dados da exceção
 */
 public class MensagemErroConsole {
   public static void main(String[] args) {
      int valor=5,divisor=0,total=0;
      try {
        total = valor / divisor; // divisão por zero
      } catch (Exception excecao) {
         // mostra mensagem de erro
         System.out.println(excecao.getMessage() +
                                                "\n\n");
        // mostra dados da exceção
        excecao.printStackTrace();
      }
   }
}
```

→ catch esta capturando uma exceção de classe mais superior (Exception), onde qualquer exceção que aconteça a executará. Porém, ser genérico é ruim para a identificação correta do que aconteceu, além de ter baixa precedência.

Tratamento de Exceções

Utilização das Exceções

- O tratamento de exceções não deve substituir um condicional simples, pois onera significativamente o desempenho do equipamento que estiver sendo usado. Este tratamento deve ser usado só em situações excepcionais
- Não individualizar o tratamento de exceções com um try para cada possível situação errônea, pois polui o código e o torna enorme e de mais difícil compreensão
- Sem esconder as exceções com recursos que possibilitam "silenciá-las", porém sua ocorrência poderá interferir em um momento nada adequado ao usuário. Por isso, trate realmente as exceções que acontecerão
- Propagar as exceções permite que os recursos com mais capacidade para tratá-las possam realizar seu tratamento adequado, não sendo problema esta propagação, muito pelo contrário, é a solução quando aplicada adequadamente

Exercícios de Fixação

2) Execute o último exemplo que realiza a divisão de dois números reais e informe um deles e pressione o botão Cancelar no outro. Analise o que acontecer e trate a exceção gerada de forma a apresentar uma orientação ao usuário que o cálculo não poderá ser realizado porque faltaram valores para sua operação.

3) A última prova de Fórmula 1 acontecerá no Brasil, e você foi contrato para elaborar uma aplicação gráfica que acompanhe as voltas de classificação de cada possível participante. Seu programa analisará o tempo de três voltas de classificação de cada carro que deseja se classificar para largada de domingo, onde até 60 carros poderão participar das classificações, mas somente os menores 26 tempos largaram neste Grande Prêmio. Armazene para cada carro, controlado somente por seu número de identificação anual, o tempo das três voltas dos participantes das provas de classificação.

Exercícios de Fixação

Continuação do exercício 3

A quantidade de participantes das provas de classificação nunca serão menores 10 nem maiores que 60 e os tempos registrados serão sempre 3 de 60 à 300 segundos em uma volta. Use os recursos computacionais que forem necessários para esta implementação de deverá possuir no mínimo três classes diferentes, podendo entre elas existir herança ou não. Depois da leitura de todos os tempos dos carros participantes, o programa deverá apresentar a lista dos 26 carros classificados para prova indicando seu número anual e o tempo em segundos, além da posição na largada. Enquanto o usuário quiser, também deverá ser permitida a consulta do número de um carro, no intervalo da quantidade que participou do treino, informando o tempo de suas três voltas registradas, seguida da média aritmética destes três tempos com formatação de 5 casas decimais, sendo este programa encerrado quando o usuário informar que não deseja mais analisar nenhum tempo de classificação.

Exercícios de Fixação

4) Elabore um programa que faça a leitura do nome e idade de uma pessoa e apresente o resultado de sua análise sobre a idade válida informada. Além de garantir a coleta da idade e primeiro nome adequado de cada pessoa registrada, verifique as possíveis exceções básicas que podem acontecer no fornecimento (digitação) desatento do usuário. Esta classificação consiste em:

   de 1 até 2 => nenê
   de 3 até 11 => criança
   de 12 até 17 => adolescente
   de 18 até 59 => adulto
   de 60 até 79 => idosos
   acima de 79 => ancião

   Realize esta análise enquanto o usuário desejar e apresente na console o nome do pessoa analisada, sua idade informada e a classificação coerente a mesma.

Comparadores

O uso das coleções possibilita rápida ordenação referente a seqüência com que os dados foram informados pelo usuário, mas normalmente existe a necessidade de se respeitar uma **ordenação adequada** aos valores que serão apresentados.

Para isso são utilizadas duas **interfaces** que permitem a ordenação coerente com o contexto desejado e apropriado ao seu usuário.

Comparable x **Comparator**

(só 1 classificação) (várias classificações)

Comparadores

Comparable

É a interface usada para definir a ordem dos elementos em uma coleção, sendo por meio dela sobrescrito o método **compareTo**.

Com isso é possível realizar a operação de comparação do próprio objeto com outro na coleção.

Estas comparações para ordenação desejada podem ser usadas caso os objetos que serão adicionados na coleção já implementem esta interface. Caso contrário elas terão que ser implementadas pelo programador, que tem a chance de embutir esta interface em seu código.

Comparadores

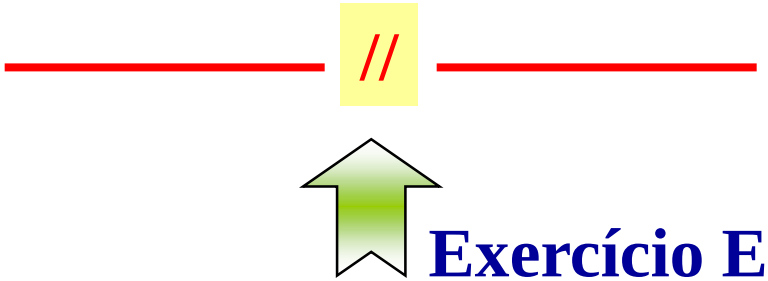
```
/** Síntese
 *    Objetivo: guardar e mostrar dados de cidades
 *    Entrada:  cidades e DDDs
 *    Saída: listagem das cidades cadastradas em ordem
*/
import java.util.ArrayList;
import javax.swing.JOptionPane;
import java.util.*;
public class ArrayLista {
  public static void main(String[] args) {
    List listaCidades = new ArrayList();
    int resposta = -1;
    do {
      do {
        resposta = JOptionPane.showConfirmDialog(null,
           "Deseja registra uma cidade?", "Pergunta",
                              JOptionPane.YES_NO_OPTION,
                              JOptionPane.PLAIN_MESSAGE);
      } while (resposta != 0 && resposta != 1);
      if (resposta == 0) {
        String ddd = "";
        Cidade cidade = new Cidade();
        cidade.setNome(JOptionPane.showInputDialog(
          null,"Digite o nome da cidade.","Registro",
                              JOptionPane.PLAIN_MESSAGE));
```

Comparadores

```
// continuação do exemplo anterior
        ddd = JOptionPane.showInputDialog(null,
             "Digite o DDD desta cidade.","Registro",
                                JOptionPane.PLAIN_MESSAGE);
        do {
          try {
             resposta = Integer.parseInt(ddd);
             if (resposta < 10)
               throw new Exception("DDD inválido.");
          } catch (Exception exc) {
             ddd = JOptionPane.showInputDialog(null,
             "DDD Inválido. Digite DDD maior que 10.",
               "Registro", JOptionPane.ERROR_MESSAGE);
             resposta = Integer.parseInt(ddd);
          }
        } while(resposta < 10);
        cidade.setDDD(resposta);
        listaCidades.add(cidade);
        resposta = 0;
      }
    } while (resposta == 0);
    System.out.println("DDD\tCidade\n===    =========");
```

Comparadores

```
// continuação do exemplo anterior
     for(int aux=0;aux < listaCidades.size(); aux++) {
       Cidade cid = (Cidade) listaCidades.get(aux);
       System.out.println("" + cid.getDDD() + "\t" +
                                              cid.getNome());
     }
   }
}
```



Exercício Extra

Observe a codificação acima e veja como a simples apresentação dos valores contidos em uma coleção resultou na programação com certa complexidade.

Por isso **altere esta apresentação** para a instrução mais adequada a tarefa de percorrer uma coleção de seus objetos armazenados.

Comparadores

```
/** Síntese
 *   Conteúdo: nome da cidade, DDD
 *     - getNome(),getDDD(),setNome(String),setDDD(int)
 *     - toString(), compareTo(Object)
 */
import java.util.Arrays;
public class Cidade implements Comparable {
  private String nome;
  private int DDD;
  public String getNome() {
    return nome;
  }
  public void setNome(String nome) {
    this.nome = nome;
  }
  public int getDDD() {
    return DDD;
  }
  public void setDDD(int DDD) {
    this.DDD = DDD;
  }
  public String toString() {
    return ("" + this.getDDD() + "\t" + this.getNome());
  }
```

Comparadores

```
// continuação do exemplo anterior
 public int compareTo(Object objeto) {
  Cidade cidadeParametro = (Cidade)objeto;
  return getNome().compareTo(cidadeParametro.getNome());
 }
}
```

- O método **compareTo** deve ser implementado para uso da interface Comparable
- O retorno deste método é inteiro e para ordenação **crescente** ele deve retornar:
  - ⇒ **1** quando elemento do objeto atual for **maior** que do parâmetro;
  - ⇒ **0** quando elemento do objeto atual for **igual** ao do parâmetro;
  - ⇒ **-1** quando o atual for **menor** que o do parâmetro.
- Esta interface é usada somente sobre objetos e já está implementada em classes como String, Integer, Float, etc.

→ Após nova execução do programa confira sua ordenação. Ela não aconteceu pelo nome da cidade porque não foi solicitada. Assim, inclua a instrução na classe ArrayLista, antes de mostrá-los: **Collections.sort(listaCidades);**

Exercício de Fixação

5) Com intuito de **ordenar o problema anterior** (cadastro de cidade e DDD), porém em ordem crescente de DDD informado pelo usuário, altere a solução e permita que o usuário do programa de cadastro de nomes de cidades e seus respectivos DDDs tenha a apresentação dos registros completos de cada cidade informada, mas em ordem do DDD usando a interface `Comparable`.

Comparadores

Comparator

É a interface usada para definir a ordem dos elementos em uma coleção, sendo por meio dela sobrescrito o método **compare**.

Por esta interface é possível realizar várias classificações diferentes sobre uma mesma coleção, sem a necessidade de alteração na classe.

Enquanto a Comparable compara os próprios objetos, a Comparator envolve um objeto externo nesta comparação e ordenação. Assim, é necessário criar e instanciar a classe responsável pela comparação na Comparator e passá-la como parâmetro para o método **`sort`**().

Comparadores

Observe o mesmo exemplo do Comparable usando o Comparator.

```
/** Síntese
 *    Objetivo: guardar e mostrar dados de cidades
 *    Entrada:  cidades e DDDs
 *    Saída: listagem das cidades cadastradas em ordem
*/
import javax.swing.JOptionPane;
import java.util.*;
public class VariasOrdens {
  public static void main(String[] args) {
    List listaCidades = new ArrayList();
    String ordenar[] = {"Nome","DDD"};
    int resposta = -1, ordem;
    do {
      do {
        resposta = JOptionPane.showConfirmDialog(null,
           "Deseja registra uma cidade?", "Pergunta",
                              JOptionPane.YES_NO_OPTION,
                              JOptionPane.PLAIN_MESSAGE);
      } while (resposta != 0 && resposta != 1);
```

Comparadores

```
// continuação do exemplo anterior
      if (resposta == 0) {
        String ddd = "";
        Cidade cidade = new Cidade();
        cidade.setNome(JOptionPane.showInputDialog(
          null,"Digite o nome da cidade.","Registro",
                              JOptionPane.PLAIN_MESSAGE));
        ddd = JOptionPane.showInputDialog(null,
             "Digite o DDD desta cidade.","Registro",
                              JOptionPane.PLAIN_MESSAGE);
        do {
             try {
                resposta = Integer.parseInt(ddd);
                if (resposta < 10)
                  throw new Exception("DDD inválido.");
             } catch (Exception exc) {
                ddd = JOptionPane.showInputDialog(null,
                "DDD Inválido. Digite DDD maior que 10.",
                  "Registro", JOptionPane.ERROR_MESSAGE);
                resposta = Integer.parseInt(ddd);
             }
         } while(resposta < 10);
```

Comparadores

```
// continuação do exemplo anterior
                cidade.setDDD(resposta);
                listaCidades.add(cidade);
                resposta = 0;
            }
        } while (resposta == 0);
        ordem = JOptionPane.showOptionDialog(null,
                "Listagem Ordenada por?","Ordenação", 0,
         JOptionPane.QUESTION_MESSAGE,null,ordenar,ordenar[0]);
        if(ordem == 0) {
          OrdenaNome ordemNome = new OrdenaNome();
          Collections.sort(listaCidades,ordemNome);
        }
        else {
          OrdenaDDD ordemDDD = new OrdenaDDD();
          Collections.sort(listaCidades,ordemDDD);
        }
        System.out.println("DDD\tCidade\n===    =========");
        for(int aux=0;aux < listaCidades.size(); aux++) {
            Cidade cid = (Cidade) listaCidades.get(aux);
            System.out.println("" + cid.getDDD() + "\t"
                                        + cid.getNome());
        }
    }
}
```

//

Comparadores

```
/** Síntese
 *   Conteúdo: nome da cidade,DDD
 *     - getNome(),getDDD(),setNome(String),setDDD(int)
 *     - toString()
 */
public class Cidade {
  private String nome;
  private Integer DDD;

  public String getNome() {
    return nome;
  }
  public void setNome(String nome) {
    this.nome = nome;
  }
  public Integer getDDD() {
    return DDD;
  }
  public void setDDD(Integer DDD) {
    this.DDD = DDD;
  }
  public String toString() {
    return ("" + this.getDDD() + "\t" + this.getNome());
  }
}
```

//

Comparadores

```
/** Síntese
 *   Conteúdo:
 *      - compare(Object, Object) por Nome da Cidade
 */
import java.util.Comparator;
public class OrdenaNome implements Comparator<Cidade> {
  public int compare(Cidade cidade1, Cidade cidade2) {
    return (cidade1.getNome().compareTo(cidade2.getNome()));
  }
}
```

//

```
/** Síntese
 *   Conteúdo:
 *     - compare(Object, Object) por DDD
 */
import java.util.Comparator;
public class OrdenaDDD implements Comparator<Cidade> {
  public int compare(Cidade cidade1, Cidade cidade2) {
    return (cidade1.getDDD().compareTo(cidade2.getDDD()));
  }
}
```

Coleções, Generics e Comparadores

Observando um **outro exemplo**, relacionado aos conteúdos deste material consiste na evolução do código inicial para este usando Comparable.

```
/** Síntese
 *    Objetivo: guardar e mostrar dados de cidades
 *    Entrada:  cidades e DDDs
 *    Saída:    lista ordenada por nome das cidades */
import java.util.*;
import javax.swing.JOptionPane;
public class GerenciaCidades {
  public static void main(String[] args) {
    List<Cidade> listaCidades = new ArrayList<Cidade>();
    int resposta = -1;
    do {
      do {
        resposta = JOptionPane.showConfirmDialog(null,
           "Deseja registra uma cidade?", "Pergunta",
                              JOptionPane.YES_NO_OPTION,
                              JOptionPane.PLAIN_MESSAGE);
      } while (resposta != 0 && resposta != 1);
```

Coleções, Generics e Comparadores

```
// continuação do exemplo anterior
      if (resposta == 0) {
        String ddd = "";
        Cidade cidade = new Cidade();
        cidade.setNome(JOptionPane.showInputDialog(
          null,"Digite o nome da cidade.","Registro",
                              JOptionPane.PLAIN_MESSAGE));
        ddd = JOptionPane.showInputDialog(null,
             "Digite o DDD desta cidade.","Registro",
                              JOptionPane.PLAIN_MESSAGE);
        do {
          try {
             resposta = Integer.parseInt(ddd);
             if (resposta < 10)
               throw new Exception("DDD inválido.");
          } catch (Exception exc) {
             ddd = JOptionPane.showInputDialog(null,
             "DDD Inválido. Digite DDD maior que 10.",
               "Registro", JOptionPane.ERROR_MESSAGE);
             resposta = Integer.parseInt(ddd);
          }
        } while(resposta < 10);
```

Coleções, Generics e Comparadores

```
// continuação do exemplo anterior
          cidade.setDDD(resposta);
          listaCidades.add(cidade);
          resposta = 0;
        }
      } while (resposta == 0);
      System.out.println("DDD\tCidade\n===    =========");
      Collections.sort(listaCidades);   // ordenando
      for(Cidade cid : listaCidades)  // mostra com for-each
         System.out.println(cid);
   }
}
```

//

```
/** Síntese
 *   Conteúdo: estrutura heterogênea de Cidade
 *     - getNome(),getDDD(),setNome(String),setDDD(int)
 */
import java.util.Arrays;
public class Cidade implements Comparable {
  private String nome;
  private int DDD;
```

Coleções, Generics e Comparadores

```
// continuação do exemplo anterior
  public String getNome() {
    return nome;
  }
  public void setNome(String nome) {
    this.nome = nome;
  }
  public int getDDD() {
    return DDD;
  }
  public void setDDD(int DDD) {
    this.DDD = DDD;
  }
  public String toString() {
    return ("" + this.getDDD() + "\t"
                                    + this.getNome());
  }
  public int compareTo(Object objeto) {
    Cidade cidadeParametro = (Cidade)objeto;
    return getNome().compareTo(cidadeParametro.getNome());
  }
}
```

Exercício de Fixação

6) Usando a JOptionPane solicite um time de futebol brasileiro e todos os anos que este foi campeão brasileiro, lembrando que este pode nunca ter sido campeão. Armazene os anos em uma lista utilizando Generics que só aceitará valores inteiros maiores que 1900 e menor que 2200. Apresente por fim o nome do time com todas as letras em maiúsculo e os anos dos títulos conquistados em ordem crescente. Caso o time não tenha ganho nem um título deverá ser mostrada a mensagem: ***Ainda não foi campeão brasileiro***.

Referência de Criação e Apoio ao Estudo

Material para Consulta e Apoio ao Conteúdo

- HORSTMANN, C. S., CORNELL, G., Core Java2, volume 1, Makron Books, 2001.
  - Capítulo 11
- FURGERI, S., Java 2: Ensino Didático: Desenvolvendo e Implementando Aplicações, São Paulo: Érica, 2002.
  - Capítulo 3
- CAMARÃO, C., FIGUEIREDO, L., Programação de Computadores em Java, Rio de Janeiro: LTC, 2003.
  - Capítulo 13
- Universidade de Brasília (UnB FGA)
  - http://cae.ucb.br/conteudo/unbfga

  (escolha a disciplina **Orientação a Objetos** no menu superior)