ELABORATA

INFORMÁTICA

JAVA II

Curitiba / 2018

Srs. Alunos

A Elaborata Informática, com o objetivo de continuar prestando-lhe um excelente nível de atendimento e funcionalidade, informa a seguir algumas regras que devem ser observadas quando do uso do laboratório e seus equipamentos, visando mantê-los sempre em um perfeito estado de funcionamento para um melhor aproveitamento de suas aulas.

É proibido:

- Atender celular. Por favor, retire-se da sala, voltando assim que desligar.
- Fazer cópias ilegais de software (piratear), com quaisquer objetivos.
- Retirar da sala de treinamento quaisquer materiais, mesmo que a título de empréstimo.
- Divulgar ou informar produtos ou serviços de outras empresas sem autorização por escrito da direção Elaborata.
- Trazer para a sala de treinamento, qualquer tipo de equipamento pessoal de informática, como por exemplo:
  - Computadores de uso pessoal
  - Notebooks
  - Placas de vídeo
  - Placas de modem
  - Demais periféricos
  - Peças avulsas como memória RAM, ferramentas, etc.
- O consumo de alimentos ou bebidas
- Fumar

Atenciosamente

Elaborata Informática

Sumário

CAPÍTULO 1
PADRÕES DE PROJETO

O termo Padrão de Projeto surgiu quando Christopher Alexander catalogou cerca de 250 padrões para a área da arquitetura civil. O objetivo de Alexander era catalogar soluções para problemas recorrentes na arquitetura e construção civil.

Segundo Alexander, um padrão de projeto “descreve um problema no nosso ambiente e o núcleo da sua solução, de tal forma que você possa usar esta solução mais de um milhão de vezes, sem nunca fazê-lo da mesma forma.”

Em geral, um padrão de projeto segundo Alexander possui: um nome, uma descrição do problema, uma descrição da solução e uma relação de consequências.

Com o passar dos tempos a ideia de Alexander foi adaptada para a Engenharia de Software pelos profissinais Erich Gama, Richard Helm, Ralph Johnson, John Vlissides, também conhecidos como a gangue dos quatro (Gof – Gang of Four), através da obra Design Patterns – Elements of Reusable Object-Oriented Software, editora Addison Wesley Longman.

1.1 Padrões de Projeto GoF

Segundo o GoF (Gang of Four) os padrões de projeto para software são catalogados em três categorias e totalizam 23 padrões. As três categorias são: criação, estrutural e comportamental.

Abaixo são listados os 23 padrões de projeto para software e suas respectivas categorias segundo o GoF.

Padrões de criação

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Padrões estruturais

- Adapter
- Bridge
- Composite

- Decorator
- Facade
- Flyweight
- Proxy

Padrões comportamentais

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

1.2 Padrão de Projeto Singleton

O padrão de projeto Singleton garante que o software terá apenas uma instância do objeto declarado como singleton.

O exemplo abaixo mostra a implementação do padrão de projeto Singleton.

```
public class SingletonSample {

  private static SingletonSample singleton = new
       SingletonSample( );

  private SingletonSample() { }

  public static SingletonSample getInstance( ) {
    return singleton;
  }
```

```
  public static void method() {
    System.out.println("método do Singleton");
  }
}
```

1.3 Padrão de Projeto Factory Method

O padrão de projeto Factory Method define uma interface para criar objetos onde a decisão de qual será criado e feita por uma subclasse.

O exemplo abaixo mostra a implementação do padrão de projeto Factory Method.

```
public interface Shape {
   void draw();
}

public class Rectangle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
public class Square implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}

public class Circle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}

public class ShapeFactory {
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();
```

```
      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();

      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();

      } else
         return null;
   }
}

public class FactoryMethodDemo {
   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();

      Shape shape1 = shapeFactory.getShape("CIRCLE");
      shape1.draw();

      Shape shape2 = shapeFactory.getShape("RECTANGLE");
      shape2.draw();
   }
}
```

1.4 DAO

O DAO (Data Access Object) é um padrão para persistência de dados que define uma interface que abstrai e encapsula os mecanismos de acesso a dados. Os dados podem estar armazenados em arquivos em disco, bancos de dados ou qualquer outro meio de armazenamento de dados. Os mecanismos de acesso encapsulados pelo DAO são quatro: inserir, ler, atualizar e remover, também conhecidos como CRUD (Create Read Update Delete).

O exemplo abaixo mostra a implementação do padrão de projeto DAO.

```
import java.sql.SQLException;
import java.util.List;

public class DAO {

  public void inserir(Object entidade) throws SQLException {
    // Executar comando SQL para inserir dados
  }
```

```
  public List<Object> listarTodos() throws SQLException {
    // Executar comando SQL para recuperar dados
  }

  public Object encontrar(int codigo) throws SQLException {
    // Executar comando SQL para encontrar dados
  }

  public void excluir(int codigo) throws SQLException {
    // Executar comando SQL para excluir dados
  }

  public void alterar(Object entidade) throws SQLException {
    // Executar comando SQL para alterar dados
  }
}
```

1.5 Exercícios sobre Padrões de Projeto

Exercício 01 – Cadastro Bebidas

Implementar os padrões de projeto Singleton e DAO no software CadastroBebidas. O Singleton deve ser aplicado numa classe que forneça a conexão com o banco de dados. O DAO deve ser aplicado numa classe que disponibilize métodos para inserir, listar, procurar, excluir e alterar. Os métodos da classe DAO não serão implementados, apenas as assinaturas dos métodos serão criadas na classe DAO.

Exercício 02 – Cadastro Bebidas

Implementar o padrão de projeto Factory Method no software CadastroBebidas de tal modo que tenha um DAO para banco de dados e um outro DAO para arquivos binários. Os métodos da classe DAO não serão implementados, apenas as assinaturas dos métodos serão criadas na classe DAO.

CAPÍTULO 2

JDBC

O JDBC (Java DataBase Connectivity) é uma biblioteca que possibilita softwares escritos com a Linguagem Java acessar e manipular dados armazenados em bancos de dados relacionais.

2.1 Banco de Dados e Conexão

Para utilizar o JDBC é necessário ter um SGBD (Sistema Gerenciador de Banco de Dados) configurado e ter um banco de dados para utilização.

A instalação e configuração de um SGBD está fora do escopo deste curso. Os passos para criação do banco de dados é fornecido logo abaixo.

Passos para criar um banco de dados no Servidor MySQL.

– Criar um arquivo chamado despesaspessoais.sql num pasta de fácil acesso.

– Colocar o conteúdo da **Listagem 1** dentro do arquivo despesaspessoais.sql

– Acessar o console do Servidor MySQL

– Executar o comando abaixo para criar o banco de dados chamado DespesasPessoais

mysql > source <local-do-arquivo>/despesaspessoais.sql

Listagem 1. Script MySQL para criar banco DespesasPessoais

```
CREATE DATABASE DespesasPessoais;

USE DespesasPessoais;

CREATE TABLE despesas
(
  codigo INTEGER NOT NULL AUTO_INCREMENT,
  categoria VARCHAR(40) NOT NULL,
  descricao VARCHAR(100) NOT NULL,
  valor DECIMAL(10,2) NOT NULL,
  data_pagamento DATE NOT NULL,
  PRIMARY KEY (codigo)
);
```

2.1.1 Conexão

A conexão do software com o banco de dados depende de três coisas: driver jdbc, o nome do driver JDBC e uma string de conexão.

O driver JDBC é fornecido pelo fabricante do banco de dados. Todos os principais bancos de dados, como o MySQL, PostgreSQL, Oracle, MS SQL Server, fornecem um driver JDBC.

O nome do driver JDBC será utilizado para carregar o driver JDBC dentro do software. A **Tabela 1** mostra os nomes de drivers JDBC para os principais bancos de dados utilizados.

Banco de dados	Nome do Driver
MySQL	com.mysql.jdbc.Driver
PostgreSQL	org.postgresql.Driver
Firebird	org.firebirdsql.jdbc.FBDriver
HypersonicSQL	org.hsql.jdbcDriver
MS SQL Server	com.microsoft.sqlserver.jdbc.SQLServerDriver
Oracle	oracle.jdbc.driver.OracleDriver

Tabela 1. Nomes de drivers JDBC

A string de conexão é composta pela palavra jdbc, seguida pela URL referente ao local onde o banco de dados se encontra e o usuário e senha de acesso ao banco de dados. A Tabela 2 mostra as strings de conexão para os principais bancos de dados utilizados.

Banco de dados	Nome do Driver
MySQL	jdbc:mysql://<ip>:<porta>/<nome-banco>?user=<nome-usuario>&password=<senha>
PostgreSQL	jdbc:postgresql://<ip>:<porta>/<nome-banco>?user=<nome-usuario>&password=<senha>
Firebird	jdbc:firebirdsql://<ip>:<porta>/<caminho+nome-banco>?user=<nome-usuario>&password=<senha>
HypersonicSQL	jdbc:hsqldb:file:<caminho+nome-banco>?user=<nome-usuario>&password=<senha>
MS SQL Server	jdbc:sqlserver://<ip>:<porta>;databaseName=<nome-banco>;user=<nome-usuario>;password=<senha>
Oracle	jdbc:oracle:thin:@<ip>:<port>:<sid>?user=<nome-usuario>;password=<senha>

Tabela 2. Strings de conexão para principais bancos de dados

O exemplo abaixo mostra como estabelecer uma conexão para um banco de dados MySQL chamado *DespesasPessoais*.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class JDBCConexaoMySQL {
  public static void main(String[] args) {
    Connection conn = null;
    try {
      // Carrega o driver JDBC do banco de dados
      Class.forName("com.mysql.jdbc.Driver");

      // Obtem a conexão com o banco de dados
      conn = DriverManager.getConnection("
          jdbc:mysql://localhost:3306/
          DespesasPessoais?user=root&password=root");

      System.out.println("Conectado ao banco de dados...");
    } catch (ClassNotFoundException e) {
      System.out.println("ERRO: Driver JDBC não encontrado!");
    } catch (SQLException e) {
      System.out.println("ERRO: Verifique se os dados de conexão
          estão corretos!");
    } finally {
      // Fecha a conexão com o banco de dados
      try {
        if (conn != null) {
          conn.close();
        }
        System.out.println("Desconectado do banco de dados...");
      } catch (SQLException e) {
        System.out.println("ERRO: " + e.getMessage());
      }
    }
  }
}
```

2.2 Exercícios sobre Banco de Dados e Conexão

Exercício 01 – Cadastro de Bebidas

Criar o banco de dados para o software CadastroBebidas e adicionar a conexão com o banco de dados usando JDBC.

2.3 Inserindo e Selecionando Dados

Para inserir dados numa tabela em um banco de dados relacional é utilizado o comando SQL chamado INSERT. Abaixo esta a sintaxe do comando SQL INSERT.

INSERT INTO table_name (column1, column2, column3, …)

VALUES (value1, value2, value3, …);

O comando INSERT pode ser executado diretamente no console do servidor do MySQL ou em qualquer outra interface como, por exemplo, o MySQL Workbench.

O exemplo abaixo mostra como inserir uma despesa na tabela *despesas* do banco de dados *DespesasPessoais*.

```
INSERT INTO despesas
  (codigo, categoria, descricao, valor, data_pagamento)
VALUES
  (1, 'Casa', 'Condomínio', 289.00, '2018/01/05');
```

Para executar o comando SQL INSERT usando o JDBC existem duas classes: Statement e a PreparedStatement. A classe Statement é utilizada para executar comandos INSERT simples e sem parâmetros. A classe PreparedStatement estende a classe Statement e permite a execução de comandos INSERT com parâmetros.

Como a maioria das instruções INSERT são complexas e em algumas vezes precisam ser montadas dinamicamente, a classe PreparedStatement acaba sendo a mais utilizada.

O exemplo abaixo mostra como utilizar a classe PreparedStatement para inserir dados na tabela *despesas* do banco de dados *DespesasPessoais*.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class JDBCInsertMySQL {

  public static void main(String[] args) {
    Connection conn = null;
    PreparedStatement insert = null;
    try {
      // Carrega o driver do banco de dados
      Class.forName("com.mysql.jdbc.Driver");
```

```
      // Obtem a conexão com o banco de dados
      conn = DriverManager.getConnection("
          jdbc:mysql://localhost:3306/
          DespesasPessoais?user=root&password=root");

      // Prepara a instrução INSERT
      insert = conn.prepareStatement("INSERT INTO despesas
          (categoria, descricao, valor, data_pagamento) " +
          "VALUES (?,?,?,?);");

      // Define os valores para a instrução INSERT
      insert.setString(1, “Casa”);
      insert.setString(2, “Luz”);
      insert.setDouble(3, 156.00);
      insert.setString(4,  "2018/01/05");

      // Executa a instrução INSERT
      insert.execute();
      System.out.println(“Dados inseridos com sucesso!”);
    } catch (ClassNotFoundException e) {
      System.out.println(“ERRO: Driver JDBC não encontrado!”);
    } catch (SQLException e) {
      System.out.println(“ERRO: ” + e.getMessage());
    } finally {
      // Fecha a conexão com o banco de dados
      try {
        if (conn != null) {
          conn.close();
        }
      } catch (SQLException e) {
        System.out.println(“ERRO: ” + e.getMessage());
      }
    }
  }
}
```

2.3.1 Selecionando Dados

Para selecionar dados de uma tabela em um banco de dados relacional é utilizado o comando SQL chamado SELECT. Abaixo esta a sintaxe do comando SQL SELECT.

SELECT column1, column2, ...

FROM table_name;

O comando SELECT pode ser executado diretamente no console do servidor do MySQL ou em qualquer outra interface como, por exemplo, o MySQL Workbench.

O exemplo abaixo mostra como selecionar as despesas existentes na tabela *despesas* do banco de dados *DespesasPessoais*.

```
SELECT
  codigo, categoria, descricao, valor, data_pagamento
FROM
  despesas;
```

Para executar o comando SQL INSERT usando o JDBC é utilizado a classe Statement e a classe ResultSet. A classe Statement é responsável por executar o comando SELECT e a classe ResultSet cria um objeto onde serão armazenados os dados recuperados da tabela.

O exemplo abaixo mostra como recuperar as depesas armazenadas na tabela *despesas* do banco de dados *DespesasPessoais*.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Date;

public class JDBCSelectMySQL {

  public static void main(String[] args) {
    Connection conn = null;
    Statement select = null;
    ResultSet resultado = null;
    try {
      // Carrega o driver do banco de dados
      Class.forName("com.mysql.jdbc.Driver");

      // Obtem a conexão com o banco de dados
      conn = DriverManager.getConnection("
          jdbc:mysql://localhost:3306/
          DespesasPessoais?user=root&password=root");

      // Prepara a instrução SELECT
      String sql = "SELECT codigo, categoria, descricao, " +
```

```
                    "valor, data_pagamento FROM despesas;";
      select = conn.createStatement();

      // Executa a instrução SELECT e armazena os dados no
      // objeto 'resultado'
      resultado = select.executeQuery(sql);

      int codigo;
      String categoria;
      String descricao;
      Double valor;
      Date dataPagamento;

      while (resultado.next()) {
        codigo = resultado.getInt("codigo");
        categoria = resultado.getString("categoria");
        descricao = resultado.getString("descricao");
        valor = resultado.getDouble("valor");
        dataPagamento = resultado.getDate("data_pagamento");
        System.out.println(codigo + ", " + categoria + ", " +
               descricao + ", " + valor + ", " + dataPagamento);
      }
    } catch (ClassNotFoundException e) {
      System.out.println("ERRO: Driver JDBC não encontrado!");
    } catch (SQLException e) {
      System.out.println("ERRO: " + e.getMessage());
    } finally {
      // Fecha a conexão com o banco de dados
      try {
        if (conn != null) {
          conn.close();
        }
      } catch (SQLException e) {
        System.out.println("ERRO: " + e.getMessage());
      }
    }
  }

}
```

2.4 Exercícios sobre Inserção e Seleção de Dados

Exercício 01 – Cadastro de Bebidas

Implementar a inserção e seleção de dados no software CadastroBebidas.

2.5 Excluindo e Alterando Dados

Para excluir dados de uma tabela em um banco de dados relacional é utilizado o comando SQL chamado DELETE. Abaixo esta a sintaxe do comando SQL DELETE.

DELETE FROM table_name

WHERE condition;

O comando DELETE pode ser executado diretamente no console do servidor do MySQL ou em qualquer outra interface como, por exemplo, o MySQL Workbench.

O exemplo abaixo mostra como excluir a despesa de código 1 da tabela *despesas* do banco de dados *DespesasPessoais*.

```
DELETE FROM despesas
WHERE codigo = 1;
```

Para executar o comando SQL DELETE usando o JDBC é utilizado a classe Statement.

O exemplo abaixo mostra como excluir a despesa de código 1 da tabela *despesas* do banco de dados *DespesasPessoais*.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class JDBCDeleteMySQL {

  public static void main(String[] args) {
    Connection conn = null;
    Statement delete = null;
    try {
      // Carrega o driver do banco de dados
      Class.forName("com.mysql.jdbc.Driver");

      // Obtem a conexão com o banco de dados
      conn = DriverManager.getConnection("
          jdbc:mysql://localhost:3306/
          DespesasPessoais?user=root&password=root");

      // Prepara a instrução DELETE
      String sql = "DELETE FROM despesas WHERE codigo=1;";
```

```
      // Executa a instrução DELETE
      delete = conn.createStatement();
      delete.execute(sql);
    } catch (ClassNotFoundException e) {
      System.out.println("ERRO: Driver JDBC não encontrado!");
    } catch (SQLException e) {
      System.out.println("ERRO: " + e.getMessage());
    } finally {
      // Fecha a conexão com o banco de dados
      try {
        if (conn != null) {
          conn.close();
        }
      } catch (SQLException e) {
        System.out.println("ERRO: " + e.getMessage());
      }
    }
  }
}
```

2.6 Alterando Dados

Para alterar dados em uma tabela em um banco de dados relacional é utilizado o comando SQL chamado UPDATE. Abaixo esta a sintaxe do comando SQL UPDATE.

UPDATE table_name

SET column1 = value1, column2 = value2, ...

WHERE condition;

O comando UPDATE pode ser executado diretamente no console do servidor do MySQL ou em qualquer outra interface como, por exemplo, o MySQL Workbench.

O exemplo abaixo mostra como alterar o valor da despesa de código 2 para 160,00 da tabela *de despesas* do banco de dados *DespesasPessoais*.

```
UPDATE despesas
SET valor=160.00
WHERE codigo = 2;
```

Para executar o comando SQL UPDATE usando o JDBC é utilizado a classe PreparedStatement.

O exemplo abaixo mostra como excluir a despesa de código 1 da tabela *despesas* do banco de dados *DespesasPessoais*.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class JDBCUpdateMySQL {
  public static void main(String[] args) {
    Connection conn = null;
    PreparedStatement update = null;
    try {
      // Carrega o driver do banco de dados
      Class.forName("com.mysql.jdbc.Driver");

      // Obtem a conexão com o banco de dados
      conn = DriverManager.getConnection("
          jdbc:mysql://localhost:3306/
          DespesasPessoais?user=root&password=root");

      // Prepara a instrução UPDATE
      update = conn.prepareStatement("UPDATE despesas " +
                    "SET valor = ? WHERE codigo = ?;");

      // Define os valores para a instrução UPDATE
      update.setDouble(1, 160.00);
      update.setInt(2, 2);

      // Executa a instrução UPDATE
      update.execute();
    } catch (ClassNotFoundException e) {
      System.out.println("ERRO: Driver JDBC não encontrado!");
    } catch (SQLException e) {
      System.out.println("ERRO: " + e.getMessage());
    } finally {
      // Fecha a conexão com o banco de dados
      try {
        if (conn != null) {
          conn.close();
        }
      } catch (SQLException e) {
        System.out.println("ERRO: " + e.getMessage());
      }
    }
  }
}
```

2.7 Exercícios sobre Exclusão e Alteração

Exercício 01 – Cadastro de Bebidas

Implementar a exclusão e alteração de dados no software CadastroBebidas.

CAPÍTULO 3

HIBERNATE

O Hibernate ORM (Object-Relational Mapping) é a ferramenta do universo Hibernate responsável por permitir o mapeamento objeto-relacional entre as tabelas de um banco de dados relacional e as classes de um programa escrito na Linguagem Java.

3.1 Instalação e Configuração

A instalação do Hibernate consiste em adicionar arquivos da biblioteca Hibernate no classpath da aplicação Java. Abaixo estão os arquivos que devem ser adicionados no classpath do software Java.

– antlr-2.7.7.jar

– classmate-1.3.0.jar

– dom4j-1.6.1.jar

– hibernate-commons-annotations-5.0.1.Final.jar

– hibernate-core-5.2.12.Final.jar

– hibernate-jpa-2.1-api-1.0.0.Final.jar

– jandex-2.0.3.Final.jar

– javassist-3.20.0-GA.jar

– jboss-logging-3.3.0.Final.jar

– jboss-transaction-api_1.2_spec-1.0.1.Final.jar

Os arquivos acima mencionados estão armazenados na pasta lib/required da biblioteca Hibernate ORM. É necessário também incluir os arquivos que estão na pasta lib/optional/c3p0, que são os seguintes:

– c3p0-0.9.5.2.jar

– hibernate-c3p0-5.2.12.Final.jar

– mchange-commons-java-0.2.11.jar

Além da biblioteca Hibernate ORM é necessário adicionar no classpath da aplicação Java o driver JDBC referente ao banco de dados utilizado. O driver JDBC é encontrado no site do fornecedor do banco de dados.

3.1.1 Configuração

A configuração do Hibernate consiste em acrescentar os dados JDBC de acesso ao banco de dados num arquivo chamado *hibernate.properties* e criar um arquivo chamado *HibernateUtils.java* para ativar o Hibernate.

O arquivo *hibernate.properties* é a configuração padrão utilizada pelo Hibernate.

O exemplo abaixo mostra a implementação da configuração para acessar um banco de dados chamado DespesasPessoais.

Arquivo Hibernate.properties

```
hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect

hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost:3306/DespesasPessoais
hibernate.connection.username=root
hibernate.connection.password=root

hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=3000
hibernate.c3p0.max_statements=30
```

O arquivo *HibernateUtils.java* contém a lógica para carregar as configurações do Hibernate e também para iniciar o Hibernate. O exemplo abaixo mostra como implementar o arquivo *HibernateUtils.java*

```
public class HibernateUtils {
  private static SessionFactory sessionFactory = null;
  private HibernateUtils() {}

  public static SessionFactory getSessionFactory() {
    if (sessionFactory == null) {
      StandardServiceRegistryBuilder registryBuilder =
           new StandardServiceRegistryBuilder();

      registryBuilder.loadProperties("hibernate.properties");

      StandardServiceRegistry registry =
           registryBuilder.build();

      try {
        MetadataSources metadataSources =
```

```
                    new MetadataSources(registry);

        sessionFactory = metadataSources.buildMetadata()
                    .buildSessionFactory();
      } catch (Exception e) {
         e.printStackTrace();
        StandardServiceRegistryBuilder.destroy(registry);
      }
    }
    return sessionFactory;
  }

  public static void shutdown() {
    sessionFactory.close();
  }
}
```

3.2 Exercícios sobre Instalação e Configuração do Hibernate

Exercício 01 – Cadastro de Bebidas

Instalar e configurar o Hibernate no projeto CadastroBebidas.

3.3 Mapeamento e Operações CRUD

Mapear uma tabela de um banco de dados para uma classe significa que a tabela será manipulada através da classe. A classe que representará a tabela no banco de dados deve ser um POJO (Plain Old Java Objects) com anotações Hibernate.

O exemplo abaixo mostra como implementar o mapeamento da tabela "despesas" do banco de dados 'DespesasPessoais'.

```
@Entity
@Table(name="despesas")
public class DespesaModel implements java.io.Serializable {

  @Id
  @GeneratedValue(strategy = INDENTITY)
  @Column(name = "codigo", unique = true, nullable = false)
  private int codigo;

  @Column(name = "categoria", nullable = false, length = 40)
  private String categoria;
```

```
  @Column(name = "descricao", nullable = false, length = 100)
  private String descricao;

  @Column(name = "valor", nullable = false)
  private String valor;

  @Column(name = "data_pagamento", nullable = false)
  @Temporal(TemporalType.DATE)
  private int dataPagamento;

  // Getters e Setters
}
```

3.4 Inserindo Dados

Para inserir dados em uma tabela usando o Hibernate é necessário criar um objeto, preencher os atributos e gravar na tabela usando o método *save* da classe Session.

O exemplo abaixo mostra como implementar uma classe DAO contendo um método para inserir dados num banco de dados.

```
public class DespesasDAO {

  public void inserir(DespesaModel entidade) throws SQLException {
    Session session =
              HibernateUtils.getSessionFactory().openSession();
    Transaction tx = null;
    try {
      tx = session.beginTransaction();
      session.save(entidade);
      tx.commit();
    } catch (Exception e) {
      if (tx != null) {
        tx.rollback();
      }
      throw new SQLException(e);
    } finally {
      session.close();
    }
  }

}
```

Toda operação de inserção de dados requer uma sessão aberta, uma transação aberta, o comando para inserir os dados, o fechamento da transação e o fechamento da sessão. A sessão representa o acesso ao banco de dados. Todo acesso precisa estar dentro de uma sessão. A sessão deve ser encerrada tão logo termine a operação de acesso ao banco. A transação garante que todo os dados serão transferidos para o banco de dados. Um tratamento de exceção garante que nada ficara aberto quando algo der de errado ao acessar o banco de dados.

3.5 Recuperando Dados

Para recuperar dados de uma tabela usando o Hibernate é necessário utilizar o método *list* da classe *Query.* O exemplo abaixo mostra como implementar a recuperação de dados usando o Hibernate.

```
public class DespesasDAO {

  public List<DespesaModel> listar() throws SQLException {
    Session session =
              HibernateUtils.getSessionFactory().openSession();
    try {
      Query query = session.
               createQuery("SELECT d FROM DespesaModel d");
      List<DespesaModel> resultado = query.list();
      return resultado;
    } catch (Exception e) {
      throw new SQLException(e);
    } finally {
      session.close();
    }
  }

}
```

3.6 Excluindo Dados

Para excluir dados de uma tabela usando o Hibernate é necessário utilizar o método *delete* da classe *Session*. O exemplo abaixo mostra como implementar a exclusão de dados usando o Hibernate.

```
public class DespesasDAO {

  public void excluir(DespesaModel entidade) throws SQLException {
```

```
    Session session =
              HibernateUtils.getSessionFactory().openSession();
    Transaction tx = null;
    try {
      tx = session.beginTransaction();
      session.delete(entidade);
      tx.commit();
    } catch (Exception e) {
      if (tx != null) {
        tx.rollback();
      }
      throw new SQLException(e);
    } finally {
      session.close();
    }
  }

}
```

3.7 Alterando Dados

Para alterar dados em uma tabela usando o Hibernate é necessário utilizar o método *update* da classe *Session*. O exemplo abaixo mostra como implementar a atualização de dados usando o Hibernate.

```
public class DespesasDAO {

  public void atualizar(DespesaModel entidade)
                                          throws SQLException {
    Session session =
              HibernateUtils.getSessionFactory().openSession();
    Transaction tx = null;
    try {
      tx = session.beginTransaction();
      session.update(entidade);
      tx.commit();
    } catch (Exception e) {
      if (tx != null) {
        tx.rollback();
      }
      throw new SQLException(e);
    } finally {
      session.close();
    }
```

```
  }

}
```

3.8 Exercícios sobre Mapeamento e Operações CRUD

Exercício 01 – Cadastro Bebidas

Aplicar o mapeamento para a classe Bebida e implementar as operações CRUD no software CadastroBebidas.

3.9 Relacionamento um-para-muitos

O relacionamento um-para-muitos acontece quando duas tabelas em um banco de dados relacional estão relacionadas e na primeira tabela existe um registro que possuem um ou mais registros relacionados a ele na segunda tabela. Exemplos de relacionamentos um-para-muitos que podem ser citados são tabelas funcionário e dependentes, nota fiscal e itens, curso e disciplinas.

O **Figura 1** mostra o relacionamento um-para-muitos das tabelas pesquisas e entrevistados para um sistema de pesquisa de preferência de futebol.

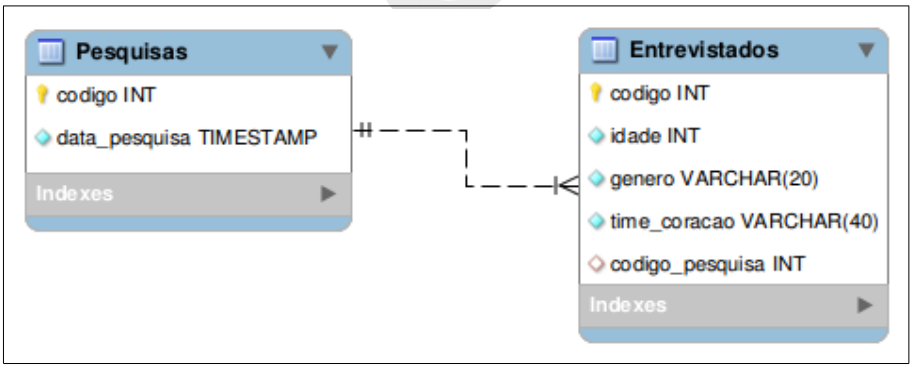

Figura 1. Relacionamento um-para-muitos

O diagrama MRN (Modelo Relacional Normalizado) representado pela **Figura 1** esta convertido para um script SQL conforme mostra a **Listagem 2**.

Listagem 2. Script MySQL relacionamento um-para-muitos

```
CREATE DATABASE PesquisaPreferenciaFutebol;
USE PesquisaPreferenciaFutebol;
CREATE TABLE Pesquisas
(
```

```
  codigo INT NOT NULL AUTO_INCREMENT,
  data_pesquisa TIMESTAMP NOT NULL,
  PRIMARY KEY (codigo)
);
CREATE TABLE Entrevistados
(
  codigo INT NOT NULL AUTO_INCREMENT,
  idade INT NOT NULL,
  genero VARCHAR(20) NOT NULL,
  time_coracao VARCHAR(40) NOT NULL,
  codigo_pesquisa INT NULL,
  PRIMARY KEY (codigo),
  FOREIGN KEY (codigo_pesquisa) REFERENCES Pesquisas (codigo)
       ON UPDATE NO ACTION
);
```

3.10 Mapeamento um-para-muitos no Hibernate

Além das annotations @Entity, @Table, @Id, @GeneratedValue e @Column utilizadas no mapeamento de uma única tabela, para mapear um relacionamento um-para-muitos no Hibernate é necessário utilizar as annotations @OneToMany, @ManyToOne e @JoinColumn. Na classe que representa a tabela principal deverá ser um arquivo que implemente a interface Set (Collections) para armazenar todos os objetos filhos. Na classe filha deverá ter um atributo que represente a classe principal.

O exemplo abaixo mostra como mapear o relacionamento um-para-muitos no Hibernate.

```
@Entity
@Table(name = "pesquisas")
public class PesquisaModel {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "codigo", unique = true)
  private Integer codigo;

  @Column(name = "data")
  @Temporal(TemporalType.TIMESTAMP)
  private Date data;

  @OneToMany(fetch = FetchType.LAZY, mappedBy = "pesquisa",
       cascade = CascadeType.ALL)
  private Set<EntrevistadoModel> entrevistados =
```

```
        new HashSet<>(0);

  // Getters e Setters
}
```

```
@Entity
@Table(name = "entrevistados")
public class EntrevistadoModel {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "codigo")
  private int codigo;

  @Column(name = "idade")
  private int idade;

  @Column(name = "genero")
  private String genero;

  @Column(name = "time")
  private String time;

  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "codigo_pesquisa", nullable = false)
  private PesquisaModel pesquisa;

  // Getters e Setters
}
```

3.11 Persistindo Relacionamento um-para-muitos

Após as classes serem devidamente anotadas para estabelecer o relacionamento um-para-muitos, é preciso preencher os objetos corretamente para que os dados sejam devidamente inseridos nas tabelas. O exemplo abaixo mostra como preencher objetos um-para-muitos.

```
PesquisaModel p = new PesquisaModel();

EntrevistadoModel e1 = new EntrevistadoModel();
e1.setIdade(18);
e1.setGenero("Masculino");
e1.setTime("Coritiba");
e1.setPesquisa(p);
```

```
EntrevistadoModel e2 = new EntrevistadoModel();
e2.setIdade(21);
e2.setGenero("Feminino");
e2.setTime("Coritiba");
e2.setPesquisa(p);

Set<EntrevistadoModel> entrevistados = new HashSet<>();
entrevistados.add(e1);
entrevistados.add(e2);

p.setData(new Date());
p.setEntrevistados(entrevistados);

Session session =
     HibernateUtils.getSessionFactory().openSession();
Transaction t = null;
try {
  t = session.beginTransaction();
  session.save(p);
  session.flush();
  t.commit();
} catch (Exception e) {
  e.printStackTrace();
  if (t != null)
  t.rollback();
} finally {
  session.close();
}
```

3.12 Exercícios sobre Relacionamento um-para-muitos

Exercício 01 – Pesquisa Preferencia Futebol

Será feito junto com os alunos durante a apresentação e explicação dos itens acima descritos.

CAPÍTULO 4

WRAPPERS E AUTOBOXING/ UNBOXING

Os Wrappers são classes Java que encapsulam tipos primitivos em objetos para que operações sobre objetos possam ser realizadas em dados do tipo primitivo.

Tipos primitivos possuem um desempenho maior em relação a objetos tanto no armazenamento como em operações de cálculo.

Apesar de o desempenho melhor, os tipos primitivos não podem ser utilizados em Collections e Generics. Para usar tipos primitivos em estruturas de dados fornecidas pelas Collections e também para criar métodos e classes genéricas, são utilizados os wrappers (encapsuladores).

A **Figura 2** mostra a hierarquia das classes wrappers da Linguagem Java.

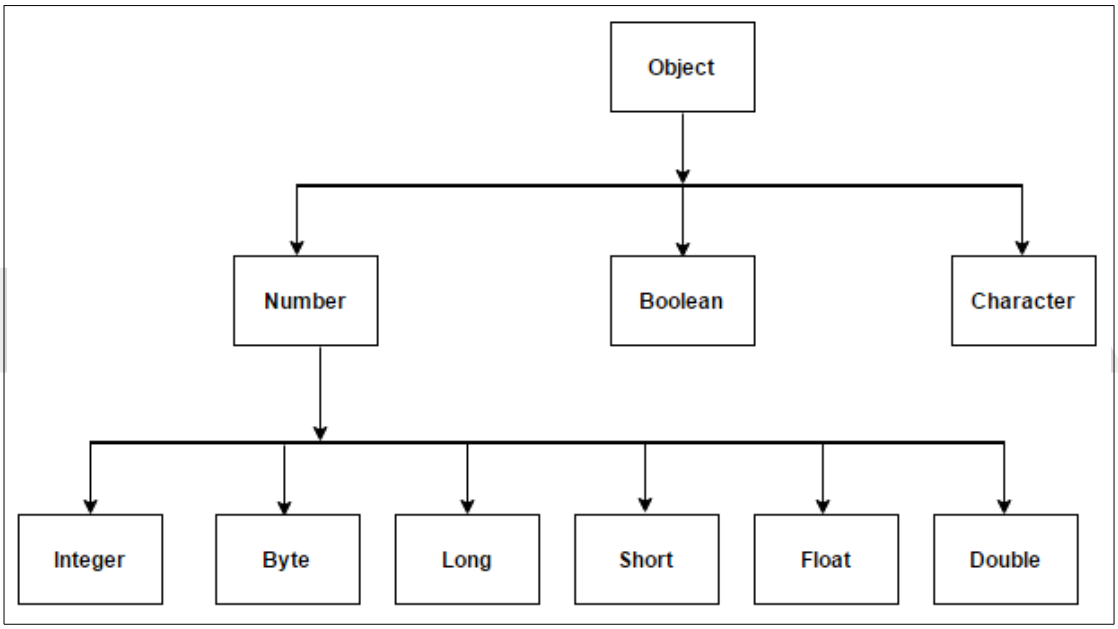

Figura 2. Hierarquia classes wrappers

O exemplo abaixo mostra como encapsular um tipo primitivo *int* num objeto *Integer*.

Integer numero = new Integer(10);

Após a declaração acima ser executada o *numero* passa a ser um objeto do tipo *Integer* contendo o valor primitivo 10.

Para comparar dois valores primitivos encapsulados em objetos deve-se utilizar o método *equals* e não o operador de *igualdade* (==). O exemplo abaixo mostra como comprar valores primitivos encapsulados.

```
public class WrappersSample {

  public static void main(String[] args) {

    Integer numero1 = new Integer(10);
    Integer numero2 = new Integer(10);

    if (numero1 == numero2)
      System.out.println("são iguais");
    else
      System.out.println("são diferentes");

    if (numero1.equals(numero2))
      System.out.println("são iguais");
    else
      System.out.println("são diferentes");
  }

}
```

O programa acima ao ser executado produzirá as seguintes saídas: são diferentes e são iguais. A primeira comparação resultará falso porque não deve-se utilizar o operador de igualdade para comparar objetos. Comparação de objetos sempre deve ser feita utilizando-se o método *equals*.

4.1 Autoboxing

A partir do Java 5 foi simplificado o encapsulamento de tipos primitivos em objetos com a inclusão do recurso Autoboxing.

O autoboxing é o processo pelo qual um tipo primitivo é encapsulado automaticamente no objeto correspondente ao tipo primitivo. Por exemplo, com o autoboxing não é mais necessário instanciar um objeto passando o tipo primitivo como parâmetro. O exemplo abaixo mostra a aplicação do autoboxing.

Integer numero = 10;

Após a declaração acima ser executada, tem-se um objeto do tipo *Integer* chamado *numero* com o valor primitivo 10.

4.2 Unboxing

O unboxing é o inverso do autoboxing. Ocorre quando um objeto é armazenado numa variável de tipo primitivo. O valor primitivo contido no objeto é automaticamente desencapsulado e armazenado na variável primitiva. O exemplo abaixo mostra a aplicação do unboxing.

```
Integer numObj = new Integer(10);

int num = numObj;
```

4.3 Exercícios sobre Wrappers e Autoboxing/Unboxing

Exercício 01 – Tabela Tipos de Dados

Criar um formulário contendo uma tabela que mostra os seguintes itens sobre os tipos de dados: Nome Tipo Dado, Bytes em Memória, Valor Mínimo e Valor Máximo.

Exercício 02 – Tabela ASCII

Criar um programa que mostre na tela a tabela ASCII. A tabela ASCII deve ter as seguintes colunas: Nome Caractere, Valor em Binario, Valor em Octal, Valor em Decimal e Valor em Hexadecimal.

CAPÍTULO 5

GENERICS

Os generics foram incluídos na versão 5 do Java como um recurso para forçar a segurança de tipos em listas de objetos e também para permitir a criação de métodos e classes genéricos.

5.1 Segurança de Tipos em Listas

Segurança de tipos em listas de objetos significa garantir que apenas objetos de um tipo de dados sejam armazenados numa lista declarada para tal tipo de dados.

Observe a lista de objetos abaixo.

```
List cidades = new ArrayList();

cidades.add("Curitiba");

cidades.add("Maringá");

cidades.add(new Integer(100));
```

Foram inseridos dois objetos do tipo String e um objeto do tipo Integer. Perfeitamente possível em relação ao armazenamento. Como não foi definido nenhum tipo específico de dados para a lista cidades, ela aceitará qualquer coisa. O problema está no momento da recuperação dos objetos armazenados na lista. A lista foi criada para armazenar objetos do tipo String, mas em algum momento foi inserido um objeto não String, isso causará um erro na recuperação dos itens.

Para garantir que uma lista de objetos armazene apenas objetos de um determinado tipo, é preciso aplicar o recurso chamado Generico a lista de objetos. Aplicando o Generico a lista ficaria assim:

```
List<String> cidades = new ArrayList<String>();

cidades.add("Curitiba");

cidades.add("Maringá");

cidades.add(new Integer(100));
```

5.2 Métodos Genéricos

Outra aplicação do recurso Generics da versão 5 do Java é na criação de métodos genéricos. Um cenário para uso de métodos genéricos seria: Criar uma classe para imprimir vetores de números inteiros e números double. Provavelmente a primeira idéia a surgir seria a de criar dois métodos, um para imprimir números inteiros e um

outro para imprimir números double. O exemplo abaixo mostra a classe com os dois métodos.

```
public class ImprimirVetores
{
  public static void main(String[] args)
  {
    int[] fibonacci = {1, 1, 2, 3, 5, 8, 13};
    double[] temperaturas = {22.5, 25.9, 24.6, 21.2, 19.0};

    System.out.println("Sequencia Fibonacci...");
    mostrarElementos(fibonacci);

    System.out.println("Temperaturas...");
    mostrarElementos(temperaturas);
  }

  private static void mostrarElementos(int[] elementos)
  {
    for (int i=0; i < elementos.length; i++)
      System.out.println(elementos[i]);
  }

  private static void mostrarElementos(double[] elementos)
  {
    for (int i=0; i < elementos.length; i++)
      System.out.println(elementos[i]);
  }
}
```

A classe foi bem desenhada, foi aplicado a sobrecarga de métodos e esta funcional. Mas agora a classe precisa imprimir vetores contendo caracteres. Qual seria a solução? Criar um terceiro método sobrecarregado para imprimir o vetor de caracteres. É uma possível solução, mas não a ideal.

A solução ideal seria criar apenas um método que utilize Generico como tipo de dado do parâmetro. O exemplo abaixo mostra como a classe ImprimirVetores ficaria com apenas um método genérico.

```
public class ImprimirVetoresGenerico
{
  public static void main(String[] args)
  {
    Integer[] fibonacci = {1, 1, 2, 3, 5, 8, 13};
```

```
    Double[] temperaturas = {22.5, 25.9, 24.6, 21.2, 19.0};

    System.out.println("Sequencia Fibonacci...");
    mostrarElementos(fibonacci);

    System.out.println("Temperaturas...");
    mostrarElementos(temperaturas);
  }

  private static <T> void mostrarElementos(T[] elementos)
  {
    for (int i=0; i < elementos.length; i++)
      System.out.println(elementos[i]);
  }
}
```

Observe que foi necessário alterar os tipos int e double para Integer e Double respectivamente e também o tipo int e double dos parâmetros dos métodos foram substituídos pela letra T. A letra T é uma convenção utilizada para indicar Type. A **Tabela 3** abaixo mostra a padronização de letras utilizadas em Generics.

Letra	Propósito
E	Simboliza um elemento
K	Simboliza uma chave
V	Simboliza um valor
T	Simboliza um tipo
N	Simboliza um número

Tabela 3. Letras utilizadas em Genéricos

5.3 Classes Genéricas

Generics podem ser utilizados na definição de classes genéricas. Uma classe genérica é a classe que terá o tipo do objeto utilizado definido como genérico.

Para aplicar o recurso Generics em classes genéricas vamos imaginar o seguinte cenário: uma classe para imprimir uma lista de objetos do tipo cliente. O exemplo abaixo mostra como implementar uma classe genérica para listagem de objetos.

```
public class Cliente
{
  private String nome;
  private String telefone;
```

```
  public Cliente() {}

  public Cliente(String nome, String telefone)
  {
    this.nome = nome;
    this.telefone = telefone;
  }

  // Getters e Setters
}
```

```
public class ListagemGenerica<E>
{
  private List<E> lista = new ArrayList<E>();

  public List<E> getLista()
  {
    return lista;
  }

  public void setLista(List<E> lista)
  {
    this.lista = lista;
  }

  public void adicionar(E item)
  {
    lista.add(item);
  }
}
```

```
public class ImprimirListagemGenerica
{
  public static void main(String[] args)
  {
    ListagemGenerica<Cliente> listagem =
         new ListagemGenerica<Cliente>();

    listagem.adicionar(new Cliente("Antonio Pedrusco",
         "99342123"));
    listagem.adicionar(new Cliente("Marieta da Penha",
         "32233344"));
```

```
    for (int i=0; i < listagem.getLista().size(); i++) {
      System.out.println(listagem.getLista().get(i).getNome());
    }
  }
}
```

5.4 Exercícios sobre Generics

Exercício 01 – Cadastro Bebidas

Aplicar Generics para eliminar todos os warnings da versão do software CadastroBebidas para Hibernate.

Exercício 02 – Cadastro Bebidas

No software CadastroBebidas, substituir a classe DAO atual por uma classe DAO genérica.

CAPÍTULO 6
COLLECTIONS

Collections é uma biblioteca Java que implementa diversas estruturas de dados, tais como pilha, fila, listas e conjuntos.

A biblioteca Collections esta dividida em três grupos: List, Set e Queue. A **Figura 4** fornece uma visão geral da API Collections.

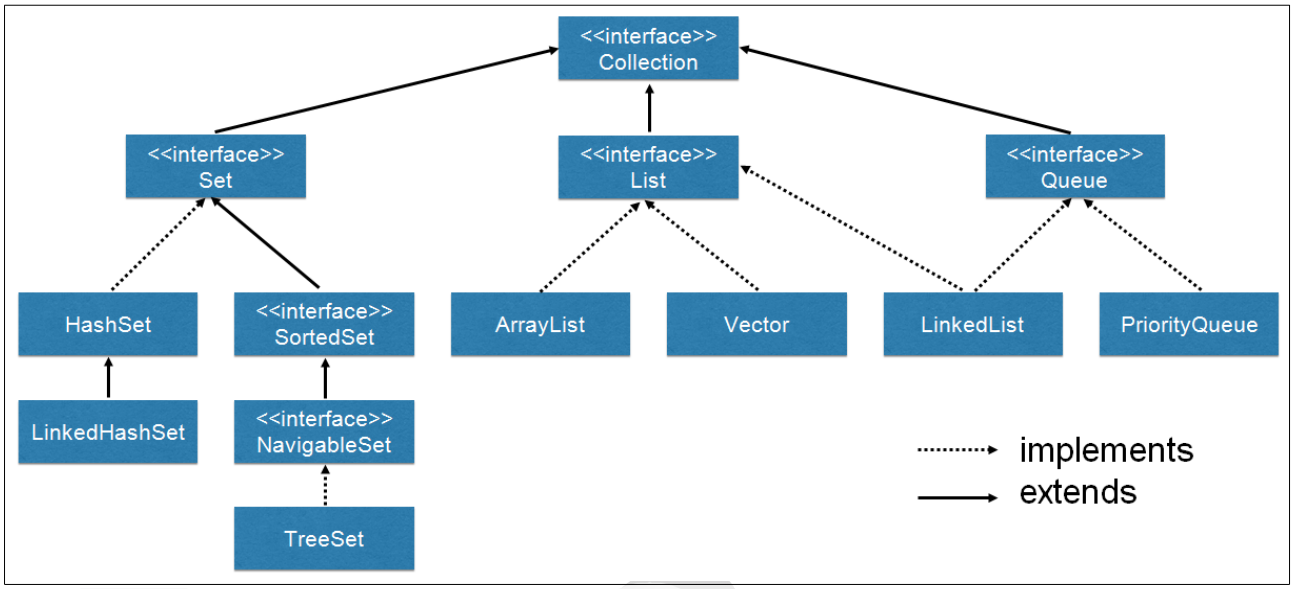

Figura 3. API Collections

6.1 List

List é uma interface que implementa a estrutura de dados chamada *listas* e deve ser utilizada para criar listas de itens ordenados com a possibilidade de incluir itens duplicados. Permite adição de itens nulos e possibilita que os itens sejam adicionados numa posição específica.

Os principais recursos fornecidos pela interface List estão relacionados abaixo.

- add(E element)

- add(int index, E element)

- get(int index)

- isEmpty()

- remove(int index)

- remove(O object)

- set(int index, E element)

- size()

A interface List é implementada pelas classes ArrayList e Vector. A diferença entre as classes ArrayList e Vector está no fato da classe Vector ser thread-safe. Isto significa que Vector deve ser utilizado apenas num contexto de programação concorrente.

O exemplo abaixo mostra como utilizar a classe ArrayList.

```
import java.util.ArrayList;
import java.util.List;

public class ArrayListSample
{
  public static void main(String[] args)
  {
    List<String> peixes = new ArrayList<String>();

    // Adicionando elementos
    peixes.add("Tilápia");
    peixes.add("Carpa");

    // Obtendo um determinado elemento da lista
    String tmp = peixes.get(0);

    // Verificando se a lista esta vazia
    if (peixes.isEmpty())
      System.out.println("A lista esta vazia");
    else
      System.out.println("A lista NÃO esta vazia");

    // Removendo o elemento 1 (Carpa)
    peixes.remove(1);

    // Alterando o elemento 0 (Tilápia agora é Lambari)
    peixes.set(0, "Lambari");

    // Pegando o total de elementos
    int totalPeixes = peixes.size();
    System.out.println("Total peixes: " + totalPeixes);

    // Imprimindo os elementos da lista
    for (String peixe : peixes)
      System.out.println(peixe);
  }
}
```

6.2 Queue

Queue é uma interface que implementa a estrutura de dados chamada *fila* e deve ser utilizada para criar filas de itens ordenados com a possibilidade de incluir itens duplicados. Permite adição de itens nulos e não possibilita que os itens sejam adicionados numa posição específica.

As operações de adição, extração e inspeção da interface Queue são fornecidas em duas formas: uma que gera exceção quando a operação falha e a outra que retorna um valor especial, falso ou nulo, quando a operação falha.

A **Tabela 4** mostra as operações de adição, extração e inspeção da interface Queue nas duas formas.

Operação	Dispara Exceção	Retorna valor especial
Inserção	add(e)	offer(e)
Extração	remove()	pool()
Inspeção	element()	peek()

Tabela 4. Operações interface Queue

O método *offer(e)* insere um elemento, se possível, caso contrário, retorna falso. Isso difere do método *add(e)*, que pode deixar de adicionar um elemento apenas lançando uma exceção não verificada. O método *offer(e)* foi projetado para uso quando a falha é uma ocorrência normal, e não excepcional, por exemplo, em filas de capacidade fixa.

Os métodos *remove()* e *poll()* removem e retornam o elemento que está no início da fila. Os métodos *remove()* e *poll()* diferem apenas em seu comportamento quando a fila está vazia: o método *remove()* lança uma exceção, enquanto o método *poll()* retorna nulo.

Os métodos *element()* e *peek()* retornam o elemento mas não o removem do ínicio da fila.

A interface Queue é implementada pelas classes LinkedList e PriorityQueue. O exemplo abaixo mostra como utilizar a classe LinkedList.

```
import java.util.LinkedList;
import java.util.Queue;

public class LinkedListSample
```

```
{
  public static void main(String[] args)
  {
    Queue<String> estados = new LinkedList<String>();

    // Adicionando elementos
    estados.add("Goias");
    estados.add("Paraná");

    // Verificando se a lista esta vazia
    if (estados.isEmpty())
      System.out.println("A lista esta vazia");
    else
      System.out.println("A lista NÃO esta vazia");

    // Pegando o total de elementos
    int totalEstados = estados.size();
    System.out.println("Total estados: " + totalEstados);

    // Imprimindo os elementos da lista
    for (String estado : estados)
      System.out.println(estado);

    while (estados.size() > 0)
      System.out.println(estados.poll());

    totalEstados = estados.size();
    System.out.println("Total estados: " + totalEstados);
  }
}
```

6.3 Set

Set é uma interface que implementa a estrutura de dados chamada *conjuntos* e deve ser utilizada para criar conjuntos que não aceitem a inclusão de itens duplicados. Permite adição de itens nulos, não mantem a ordem de inserção dos itens e não possibilita que os itens sejam adicionados numa posição específica.

Os principais métodos fornecidos pela interface Set estão relacionados abaixo.

- add(E element)

- isEmpty()

- remove(O object)

- size()

A interface Set é implementada pela classe HashSet. O exemplo abaixo mostra como utilizar a classe HashSet.

```
import java.util.HashSet;
import java.util.Set;

public class HashSetSample
{
  public static void main(String[] args)
  {
    Set<String> paises = new HashSet<String>();

    // Adicionando elementos
    paises.add("Brasil");
    paises.add("Chile");
    paises.add("Paraguai");
    paises.add("Argentina");

    // Verificando se a lista esta vazia
    if (paises.isEmpty())
      System.out.println("A lista esta vazia");
    else
      System.out.println("A lista NÃO esta vazia");

    // Removendo o elemento 1 (Chile)
    paises.remove(1);

    // Pegando o total de elementos
    int totalPaises = paises.size();
    System.out.println("Total paises: " + totalPaises);

    // Imprimindo os elementos da lista
    for (String pais : paises)
      System.out.println(pais);
  }
}
```

6.4 Exercícios sobre Collections

Exercício 01 – Caixa ATM

Implementar uma lista de objetos no software CaixaATM para que as operações de saque e depósito fiquem armazenadas em memória.

CAPÍTULO 7
ORDENAÇÃO DE ITENS

A ordenação de itens é a ação de ordenar itens segundo um critério de ordem pré-definido. Na Linguagem Java é possível ordenar números, strings e objetos utilizando recursos fornecidos pela API Collections.

7.1 Ordenação de Números

Números armazenados em vetores podem ser ordenados através do método *Array.sort()*. O exemplo abaixo mostra como ordenar em ordem crescente, números inteiros armazenados num vetor.

```
import java.util.Arrays;

public class OrdenacaoCrescenteNumeros
{
  public static void main(String[] args)
  {
    int numeros[] = {3, 1, 4, 2, 5, 0, 9, 7, 6, 8};

    Arrays.sort(numeros);

    for (int i=0; i < numeros.length; i++)
      System.out.print(numeros[i] + " ");
  }
}
```

Para ordenar números em ordem decrescente, o vetor de números precisa ser declarado como *Integer* e não com o tipo primitivo *int*. O exemplo abaixo mostra como ordenar em ordem descrecento um vetor de números inteiros.

```
import java.util.Arrays;
import java.util.Collections;

public class OrdenacaoNumerosSample
{
  public static void main(String[] args)
  {
    Integer numeros[] = {3, 1, 4, 2, 5, 0, 9, 7, 6, 8};

    Arrays.sort(numeros, Collections.reverseOrder());

    for (int i=0; i < numeros.length; i++)
      System.out.print(numeros[i] + " ");
  }
```

```
}
```

7.2 Ordenação de Strings

Strings armazenadas em listas podem ser ordenadas através do método *Collections.sort()*. O exemplo abaixo mostra como ordenar em ordem crescente strings armazenados numa lista.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class OrdenacaoCrescenteStrings
{
  public static void main(String[] args)
  {
    List<String> bairros = new ArrayList<String>();
    bairros.add("Capão Raso");
    bairros.add("Pinheirinho");
    bairros.add("Agua Verde");
    bairros.add("Batel");

    Collections.sort(bairros);

    for (String b : bairros)
      System.out.println(b);
  }
}
```

O exemplo abaixo mostra como ordenar em ordem decrescente uma lista de strings.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class OrdenacaoDecrescenteStrings
{
  public static void main(String[] args)
  {
    List<String> bairros = new ArrayList<String>();
    bairros.add("Capão Raso");
    bairros.add("Pinheirinho");
    bairros.add("Agua Verde");
```

```
    bairros.add("Batel");

    Collections.sort(bairros, Collections.reverseOrder());

    for (String b : bairros)
      System.out.println(b);
  }
}
```

7.3 Ordenação de Objetos

Objetos armazenadas em listas podem ser ordenados através da implementação da interface *Comparable*.

O objeto a ser ordenado precisa implementar a interface *Comparable* e deve sobrescrever o método *compareTo()* informando qual atributo será utilizado para ordenação.

O exemplo abaixo mostra como ordenar em ordem crescente uma lista de objetos do tipo Aluno.

Arquivo Aluno.java

```
public class Aluno implements Comparable<Aluno>
{
  private int matricula;
  private String nome;
  private char sexo;

  public Aluno() {}

  public Aluno(int matricula, String nome, char sexo)
  {
    this.matricula = matricula;
    this.nome = nome;
    this.sexo = sexo;
  }

  // Getters e Setters

  @Override
  public int compareTo(Aluno aluno)
  {
    return this.nome.compareTo(aluno.getNome());
  }
```

```
}
```

Arquivo ComparableSample.java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class ComparableSample
{
  public static void main(String[] args)
  {
    List<Aluno> alunos = new ArrayList<Aluno>();

    alunos.add(new Aluno(1002, "Francisco Alvarenga", 'M'));
    alunos.add(new Aluno(1004, "Amália Silveira Silva", 'F'));
    alunos.add(new Aluno(1001, "Danusa Pedrosa", 'F'));

    Collections.sort(alunos);

    for (Aluno a : alunos)
      System.out.println(a.getMatricula() + ", " + a.getNome());
  }
}
```

Quando a interface *Comparable* é utilizada para ordenar objetos, apenas um tipo de ordenação é possível, ou seja, se o método *compareTo()* foi sobrescrito para ordenar o objeto pelo atributo *nome* e se quisermos ordenar pelo atributo *matricula*, devemos alterar o método *compareTo()*.

Para termos mais de uma opção de ordenação sem ter que alterar o método CompareTo() devemos usar a interface *Comparator*. O exemplo abaixo mostra como utilizar a interface *Comparator* para ordenar o objeto *Aluno* de diversas maneiras.

Arquivo Aluno.java

```
public class Aluno
{
  private int matricula;
  private String nome;
  private char sexo;

  public Aluno() {}

  public Aluno(int matricula, String nome, char sexo)
  {
```

```
    this.matricula = matricula;
    this.nome = nome;
    this.sexo = sexo;
  }

  // Getters e Setters
}
```

Arquivo AlunoOrdenarNome.java

```
import java.util.Comparator;

public class AlunoOrdenarNome implements Comparator<Aluno>
{
  @Override
  public int compare(Aluno aluno1, Aluno aluno2)
  {
    return aluno1.getNome().compareTo(aluno2.getNome());
  }

}
```

Arquivo AlunoOrdenarMatricula.java

```
import java.util.Comparator;

public class AlunoOrdenarMatricula implements Comparator<Aluno>
{
  @Override
  public int compare(Aluno aluno1, Aluno aluno2)
  {
    return aluno1.getMatricula() - aluno2.getMatricula();
  }
}
```

Arquivo ComparatorSample.java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class ComparatorSample
{
  public static void main(String[] args)
  {
    List<Aluno> alunos = new ArrayList<Aluno>();

    alunos.add(new Aluno(1002, "Francisco Alvarenga", 'M'));
```

```
    alunos.add(new Aluno(1004, "Amália Silveira Silva", 'F'));
    alunos.add(new Aluno(1001, "Danusa Pedrosa", 'F'));

    Collections.sort(alunos, new AlunoOrdenarMatricula());

    for (Aluno a : alunos)
      System.out.println(a.getMatricula() + ", " + a.getNome());
  }
}
```

7.4 Exercícios sobre Ordenação

Exercício 01 – Caixa ATM

Criar dois critérios de ordenação para a tela Extrato do software CaixaATM. O primeiro critério deve ordenar as operações por data e o segundo critério deve ordenar as operações por tipo.

CAPÍTULO 8

ARQUIVOS TEXTOS

Arquivos textos são arquivos armazenados num dispositivo de armazenamento e podem ser lidos facilmente por seres humanos. Existem dois tipos de arquivos textos: arquivos contendo texto puro e arquivos contendo textos delimitados por um determinado caractere.

8.1 Criando Arquivo Texto

Para criar um arquivo texto contendo texto puro usando a Linguagem Java é necessário utilizar as classes *FileWriter* e *BufferedWriter*.

O exemplo abaixo mostra como criar um arquivo texto contendo algumas informações sobre o curso Aplicações Linguagem Java.

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class ArquivoTextoEscrita {

  public static void main(String[] args) {

    FileWriter fw = null;
    BufferedWriter out = null;
    try {
      fw = new FileWriter("dadoscurso.txt");
      out = new BufferedWriter(fw);

      out.write("Curso Aplicações Linguagem Java\n");
      out.write("60 Horas\n");
      out.write("Período noturno");

      out.flush();
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      try {
        out.close();
        fw.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
```

```
}
```

Para criar um arquivo texto contendo texto delimitado por um determinado caractere basta colocar os textos numa mesma linha, separados pelo caractere.

O exemplo abaixo mostra como criar um arquivo texto no padrão CSV aceito pelo Excel contendo as informações do curso.

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class ArquivoTextoCSVEscrita {

  public static void main(String[] args) {

    FileWriter fw = null;
    BufferedWriter out = null;
    try {
      fw = new FileWriter("dadoscurso.csv");
      out = new BufferedWriter(fw);

      StringBuilder sb = new StringBuilder();
      sb.append("Curso Aplicações Linguagem Java");
      sb.append(",");
      sb.append("60 Horas");
      sb.append(",");
      sb.append("Período noturno");

      out.write(sb.toString());

      out.flush();
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      try {
        out.close();
        fw.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
}
```

8.2 Lendo Arquivo Texto

Para ler o conteúdo de um arquivo texto contendo texto puro usando a Linguagem Java é necessário utilizar as classes *FileReader* e *BufferedReader*.

O exemplo abaixo mostra como ler um arquivo texto contendo informações sobre o curso Aplicações Linguagem Java.

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class ArquivoTextoLeitura {

  public static void main(String[] args) {
    FileReader fr = null;
    BufferedReader in = null;
    try {
      fr = new FileReader("dadoscurso.txt");
      in = new BufferedReader(fr);
      String linha;
      do {
        linha = in.readLine();
        if (linha != null) {
          System.out.println(linha);
        }
      } while (linha != null);
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      try {
        in.close();
        fr.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }

}
```

Para ler um arquivo texto contendo texto delimitado por um determinado caractere é necessário quebrar a linha após a leitura utilizando o método String.split.

O exemplo abaixo mostra como ler um arquivo texto no padrão CSV aceito pelo Excel contendo as informações do curso.

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class ArquivoTextoCSVLeitura {

  public static void main(String[] args) {
    FileReader fr = null;
    BufferedReader in = null;
    try {
      fr = new FileReader("dadoscurso.csv");
      in = new BufferedReader(fr);
      String linha;
      String dados[];
      do {
        linha = in.readLine();
        if (linha != null) {
          dados = linha.split(",");
          System.out.println(dados[0]);
          System.out.println(dados[1]);
          System.out.println(dados[2]);
        }
      } while (linha != null);
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      try {
        in.close();
        fr.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
```

```
}
```

8.3 Exercícios sobre Arquivo Texto

Exercício 01 – Caixa ATM

Na tela de extrato do software CaixaATM implementar a exportação para arquivo Excel (formato CSV) das operações contidas no objeto conta.

CAPÍTULO 9
ARQUIVOS BINÁRIOS

Arquivos binários são arquivos armazenados num dispositivo de armazenamento cujo conteúdo não pode ser lido facilmente por seres humanos. Alguns exemplos de arquivos binários são os vídeos, imagens, áudio e os programas executáveis.

Além dos exemplos acima citados de arquivos binários, é possível armazenar o conteúdo de um objeto num arquivo binário para posteriormente recuperá-lo. É isto o que muitos jogos fazem ao salvar o estado do jogo para futuramente o jogador continuar do ponto em que parou.

9.1 Armazenando Objeto em Arquivo Binário

Para armazenar um objeto num arquivo binário é necessário que o objeto seja serializável e as classes *FileOutputStream* e *ObjectOutputStream* devem ser utilizadas. Um objeto serializável é aquele que implementa a interface *Serializable*.

O exemplo abaixo mostra como armazenar um objeto num arquivo binário.

Arquivo Aluno.java

```
import java.io.Serializable;

public class Curso implements Serializable
{
  private static final long serialVersionUID =
                                -3624849046738943010L;

  private String nome;
  private int qtdeHoras;
  private String periodo;

  public Curso() {}

  public Curso(String nome, int qtdeHoras, String periodo)
  {
    this.nome = nome;
    this.qtdeHoras = qtdeHoras;
    this.periodo = periodo;
  }

  // Getters e Setters

}
```

Arquivo ArquivoBinarioObjetoEscrita.java

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class ArquivoBinarioObjetoEscrita {
  public static void main(String[] args) {
    FileOutputStream fos = null;
    ObjectOutputStream out = null;
    try {
      fos = new FileOutputStream("dadoscurso.dat");
      out = new ObjectOutputStream(fos);

      Curso curso = new Curso("Java II", 60, "Noturno");
      out.writeObject(curso);

      out.flush();
      fos.flush();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      try {
        out.close();
        fos.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
}
```

9.2 Lendo Objeto de Arquivo Binário

Para ler um objeto de um arquivo binário é necessário saber a estrutura do objeto utilizado no momento do armazenamento. As classes utilizadas para a leitura são: *FileInputStream* e *ObjectInputStream*.

O exemplo abaixo mostra como ler um objeto de um arquivo binário.

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.ObjectInputStream;

public class ArquivoBinarioObjetoLeitura {

  public static void main(String[] args) {
    FileInputStream fis = null;
    ObjectInputStream in = null;
    try {
      fis = new FileInputStream("dadoscurso.dat");
      in = new ObjectInputStream(fis);

      Curso curso = (Curso) in.readObject();

      System.out.println("Curso.....: " + curso.getNome());
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    } finally {
      try {
        in.close();
        fis.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
}
```

9.3 Armazenando uma Lista Objeto em Arquivo Binário

O processo para armazenar uma lista de objetos num arquivo binário é o mesmo utilizado para armazenar um objeto, com a diferença que a lista deve ser percorrida do início ao fim e os objetos devem ser armazenados um por um.

O exemplo abaixo mostra como armazenar uma lista de objetos num arquivo binário.

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.List;

public class ArquivoBinarioListaObjetosEscrita
{
  public static void main(String[] args)
  {
    FileOutputStream fos = null;
    ObjectOutputStream out = null;
    try {
      fos = new FileOutputStream("listaCursos.dat");
      out = new ObjectOutputStream(fos);

      List<Curso> cursos = new ArrayList<Curso>();

      cursos.add(new Curso("Java I", 60));
      cursos.add(new Curso("Java II", 60));

      for (Curso c : cursos) {
        out.writeObject(c);
      }

      out.flush();
      fos.flush();
    }
    catch (FileNotFoundException e) {
      e.printStackTrace();
    }
    catch (IOException e) {
      e.printStackTrace();
    } finally {
      try {
        out.close();
        fos.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
}
```

9.4 Lendo uma Lista de Objetos de um Arquivo Binário

O processo para ler uma lista de objetos de um arquivo binário é o mesmo utilizado para ler um objeto, com a diferença que os objetos serão lidos um por um e armazenados individualmente na lista.

O exemplo abaixo mostra como ler uma lista de objetos de um arquivo binário.

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.util.ArrayList;
import java.util.List;

public class ArquivoBinarioListaObjetosLeitura
{
  public static void main(String[] args)
  {
    FileInputStream fis = null;
    ObjectInputStream in = null;
    try {
      fis = new FileInputStream("listaCursos.dat");
      in = new ObjectInputStream(fis);

      List<Curso> cursos = new ArrayList<Curso>();
      while (true) {
        try {
          Curso curso = (Curso)in.readObject();
          if (curso != null) {
            cursos.add(curso);
          }
        }
        catch (Exception e) {
          break;
        }
      }

      for (Curso c : cursos) {
        System.out.println("Curso..: " + c.getNome() + " - " +
                                  c.getQtdeHoras());
      }
    }
    catch (FileNotFoundException e) {
```

```
      e.printStackTrace();
    }
    catch (IOException e) {
      e.printStackTrace();
    } finally {
      try {
        in.close();
        fis.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }

}
```

9.5 Exercícios sobre Arquivos Binários

Exercício 01 – Caixa ATM

Alterar o software CaixaATM para que as operações sejam armazenadas em arquivo binário.

CAPÍTULO 10

SISTEMA DE ARQUIVOS

O sistema de arquivos é o recurso utilizado pelos sistemas operacionais para organizar e gerenciar dados em um disco rígido (hard disk). Pelo sistema de arquivos é possível gravar, ler e remover dados do disco rígido.

Normalmente um sistema de arquivos estrutura o disco rígido em três partes: MBR (Master Boot Record), Diretórios e Arquivos.

O MBR esta localizado no primeiro setor do disco rígido e é nele que estão armazenadas dados e informações sobre a estrutura organizacional do disco rígido. É responsável também por inicializar o sistema operacional armazenado no disco rígido.

Os diretórios são estruturas utilizadas para centralizar e organizar os arquivos. Arquivos são estruturas onde os dados efetivamente são armazenados. Por exemplo, para guardar dados de contato de uma determinada pessoa, cria-se um arquivo contendo os dados da pessoa e este arquivo será gravado num diretório.

Cada sistema operacional implementa um tipo de sistema de arquivos. As vezes um sistema operacional pode aceitar diversos tipos de sistemas de arquivos. A **Tabela 5** relaciona os principais sistemas operacionais e seus respectivos sistemas de arquivos.

Sistema Operacional	Sistemas de arquivos suportados
Microsoft Windows	FAT 16, FAT 32 e NTFS
Linux	EXT, EXT2, EXT3, EXT4, Reiser, HPFS, JFS, XFS e ZFS
macOS	HSF Plus

Tabela 5. Sistemas operacionais e sistemas de arquivos

10.1 Extraindo Informações

A linguagem Java disponibiliza a classe *File* para manipular diretórios e arquivos. O exemplo abaixo mostra como extrair informações referentes ao espaço do disco rígido.

```
import java.io.File;

public class DirectorySample {

  public static void main(String[] args) {

    File drive = new File("c:/");
    System.out.println("Total space: " + drive.getTotalSpace());
```

```
    System.out.println("Free space: " + drive.getFreeSpace());

  }

}
```

O exemplo acima mostra o espaço total e o espaço livre de um drive padrão do sistema operacional Microsoft Windows. Para obter tais dados de um drive num sistema operacional Linux é preciso alterar o valor passado para o construtor da classe *File*.

Para tornar o código do exemplo acima portável deve-se obter o tipo do sistema operacional utilizando o método *System.getPropertie("os.name")*. O exemplo abaixo mostra como obter o nome do sistema operacional utilizado.

```
import java.io.File;

public class DirectorySample {

  public static void main(String[] args) {

    String osName = System.getProperty("os.name");

    String driveRoot = null;
    if (osName.indexOf("win") >= 0)
      driveRoot = "c:";
    else if (osName.indexOf("mac") >= 0)
      driveRoot = "/";
    else if (osName.indexOf("nix") >= 0)
      driveRoot = "/";
    else if (osName.indexOf("nux") >= 0)
      driveRoot = "/";
    else if (osName.indexOf("aix") > 0)
      driveRoot = "/";
    else if (osName.indexOf("sunos") >= 0)
      driveRoot = "/";

    File drive = new File(driveRoot);
    System.out.println("Total space: " + drive.getTotalSpace());
    System.out.println("Free space: " + drive.getFreeSpace());

  }

}
```

10.2 Selecionando Arquivos com JFileChooser

A classe *JFileChooser* possibilita de forma visual a seleção de arquivos. A caixa de diálogo exibida é a mesma exibida pelos itens de menu *File Open* e *File Save* de diversos softwares. O exemplo abaixo mostra como utilizar a classe *JFileChooser* para selecionar arquivos do tipo texto (.txt).

```
import java.io.File;

import javax.swing.JFileChooser;
import javax.swing.filechooser.FileNameExtensionFilter;

public class JFileChooserSample {

  public static void main(String[] args) {
    JFileChooser fc = new JFileChooser();

    fc.setDialogTitle("Selecione um arquivo...");
    fc.setCurrentDirectory(new
               File(System.getProperty("user.home")));

    FileNameExtensionFilter filterTXT = new
               FileNameExtensionFilter("Arquivo Texto", "txt");
    fc.setFileFilter(filterTXT);

    fc.setAcceptAllFileFilterUsed(false);

    int resposta = fc.showOpenDialog(null);
    if (resposta == JFileChooser.APPROVE_OPTION) {
      System.out.println("Arquivo selecionado..: " +
                      fc.getSelectedFile().getName());
    } else {
      System.out.println("Nenhum arquivo selecionado.");
    }
  }
}
```

10.3 Exercícios sobre Sistemas de Arquivos

Exercício 01 – Caixa ATM

Alterar o software CaixaATM para que as contas no campo combobox sejam obtidas do arquivo em disco.

CAPÍTULO 11

REDE E SOCKETS

A Linguagem Java disponibiliza diversas bibliotecas e recursos para o desenvolvimento de aplicações para ambientes onde é utilizado rede de computadores. É possível criar aplicações que enviem e-mails, aplicações que coletem dados sobre a rede e aplicações que se comuniquem e troquem dados utilizando o protocolo de rede TPC/IP.

11.1 Obtendo Dados da Rede

Para obter dados sobre o protocolo IP é necessário utilizar a classe *InetAddress*. O exemplo abaixo mostra como obter o endereço IP e o hostname do computador local.

```
import java.net.InetAddress;
import java.net.UnknownHostException;

public class InetAddressSample
{
  public static void main(String[] args)
  {
    try {
      InetAddress ia = InetAddress.getLocalHost();
      String ip = ia.getHostAddress();
      String hostname = ia.getHostName();

      System.out.println("IP Local....: " + ip);
      System.out.println("Hostname....: " + hostname);
    }
    catch (UnknownHostException e) {
      e.printStackTrace();
    }
  }
}
```

O exemplo acima apenas mostra o endereço IP da interface padrão. Atualmente os computadores possuem mais de uma interface de rede devido à presença de redes sem-fio e também de máquinas virtuais. Para obter os endereços IPs de todas as interfaces de rede instaladas é necessário utilizar a classe *NetworkInterface*.

O exemplo abaixo mostra como listar todas as interfaces de rede instaladas no computador junto aos respectivos endereços Ips.

```
import java.net.InetAddress;
import java.net.NetworkInterface;
import java.net.SocketException;
import java.util.Collections;
import java.util.Enumeration;

public class NetworkInterfaceSample
{
  public static void main(String args[]) throws SocketException
  {
    Enumeration<NetworkInterface> nets =
             NetworkInterface.getNetworkInterfaces();
    for (NetworkInterface ni : Collections.list(nets)) {
      System.out.println("Display name..: " +
             ni.getDisplayName());
      System.out.println("Name.........: " + ni.getName());

      System.out.println("InetAddress...: ");
      Enumeration<InetAddress> inetAddresses =
            ni.getInetAddresses();
      for (InetAddress ia : Collections.list(inetAddresses)) {
        System.out.println("- " + ia);
      }
      System.out.println("\n");
    }
  }
}
```

11.2 Exercícios sobre Obtendo Dados da Rede

Exercício 01 – Cliente do Servidor de Configurações

Criar um projeto chamado ServidorConfiguracao contendo uma tela para exibir os seguintes dados sobre o computador:

- Nome sistema operacional

- Arquitetura

- Versão do Java

- Fabricante do Java

- Endereço IP

- MAC Address

Além da tela, criar uma classe contendo atributos para armazenar os dados acima descritos e também uma classe que contenha métodos para fornecer os dados acima.

11.3 Enviando Dados pela Rede

O envio de dados entre aplicações usando o protocolo TCP requer a implementação da arquitetura chamada cliente/servidor. O servidor é uma aplicação que aguarda por uma conexão de um cliente e implementa a lógica para receber os dados e armazenar. A aplicação cliente conecta-se no servidor, envia os dados e encerra a conexão.

O exemplo abaixo mostra como implementar a arquitetura cliente/servidor.

```
public class Objeto implements Serializable
{
  private static final long serialVersionUID = 1L;

  private int atributo1;
  private String atributo2;

  public Objeto()
  {
  }

  public Objeto(int atributo1, String atributo2)
  {
    this.atributo1 = atributo1;
    this.atributo2 = atributo2;
  }

  // Getters e Setters
}
```

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class EnviarObjetoPorTCPServidor
{
  public static void main(String[] args)
```

```
  {
    int servidorPorta = 2233;
    try {
      ServerSocket servidor = new ServerSocket(servidorPorta);

      Socket cliente = servidor.accept();

      ObjectInputStream entrada =
              new ObjectInputStream(cliente.getInputStream());
      ObjectOutputStream saida =
              new ObjectOutputStream(cliente.getOutputStream());

      Objeto bufferRecebido = (Objeto)entrada.readObject();
      System.out.println(bufferRecebido.getAtributo1());
      System.out.println(bufferRecebido.getAtributo2());

      Objeto objCliente = new Objeto(2, "Olá cliente TCP...");
      saida.writeObject(objCliente);

      saida.close();
      entrada.close();

      servidor.close();
    }
    catch (IOException e) {
      e.printStackTrace();
    }
    catch (ClassNotFoundException e) {
      e.printStackTrace();
    }
  }
}
```

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.net.UnknownHostException;

public class EnviarObjetoPorTCPCliente
{
  public static void main(String[] args)
  {
    String servidorIP = "localhost";
```

```
    int servidorPorta = 2233;
    try {
      Socket servidor = new Socket(servidorIP, servidorPorta);

      ObjectOutputStream saida =
              new ObjectOutputStream(servidor.getOutputStream());

      Objeto objServidor = new Objeto(1, "Olá servidor TCP...");
      saida.writeObject(objServidor);

      ObjectInputStream entrada =
              new ObjectInputStream(servidor.getInputStream());

      Objeto bufferEntrada = (Objeto)entrada.readObject();
      System.out.println(bufferEntrada.getAtributo1());
      System.out.println(bufferEntrada.getAtributo2());

      entrada.close();
      saida.close();

      servidor.close();
    }
    catch (UnknownHostException e) {
      e.printStackTrace();
    }
    catch (IOException e) {
      e.printStackTrace();
    }
    catch (ClassNotFoundException e) {
      e.printStackTrace();
    }
  }
}
```

11.4 Exercícios sobre Redes e Sockets

Exercício 01 – Servidor de Configurações

Criar a tela do servidor de configurações contendo uma tabela para exibir os dados MAC Address, Sistema Operacional, Arquitetura, Versão do Java e Fabricante do Java. Implementar também a comunicação pelo protocolo TCP.

CAPÍTULO 12

THREADS

O termo *thread* significa linha ou encadeamento de execução. É um recurso utilizado para dividir um processo em várias partes de tal modo que cada parte seja executada paralelamente.

Um processo é um programa que está sendo executado pelo sistema operacional. O processo ocupa um espaço na memória do computador, possui um identificador e é controlado pelo sistema operacional.

Quando dois ou mais processos estão em excecução ao mesmo tempo, tem-se um cenário de multitarefa. Em computadores com apenas um processador, o sistema operacional utiliza uma ferramenta chamada escalonador para executar os processos. Na verdade os processos não rodam paralelamente em arquiteturas monoprocessadas. O que acontece é o escalonador do sistema operacional executar um processo e depois outro processo, interrompe o segundo, executa o primeiro, interrompe o primeiro, executa o segundo e assim por diante. Como o processador é muito rápido, tem-se a noção de que os processos estão rodando ao mesmo tempo.

Em arquiteturas multiprocessadas tem-se o cenário de multitarefa real, pois cada processo será executado em processadores diferentes.

Diferente dos processos, as threads não possuem um identificador próprio e não ocupam um espaço na memória do computador. Isto significa que o sistema operacional não pode interromper uma determinada thread de um processo. A **Figura 5** mostra um cenário contendo um processo com uma thread.

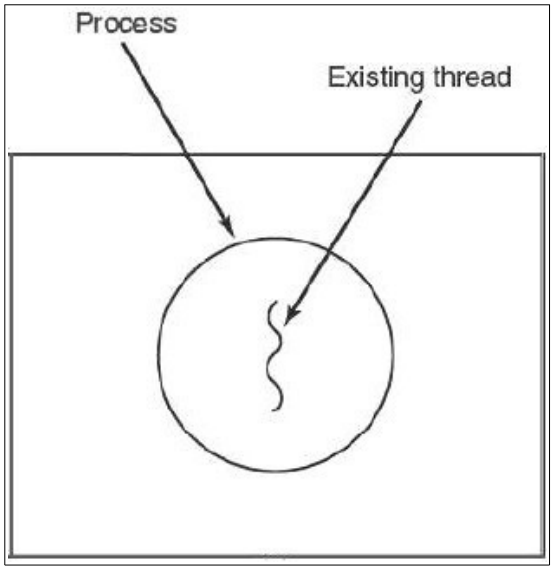

Figura 4. Processo e Thread

12.1 Ciclo de Vida das Threads

As threads possuem um ciclo de vida composto por vários estágios. Quando uma thread é criada ela entra no estado *pronta para execução*. Após ser executada a thread passa para o estado *em execução*. Estando no estado de execução a thread pode ser interrompida (estado *suspensa*) ou encerrada (estado *morta*). A **Figura 6** resume o ciclo de vida das threads.

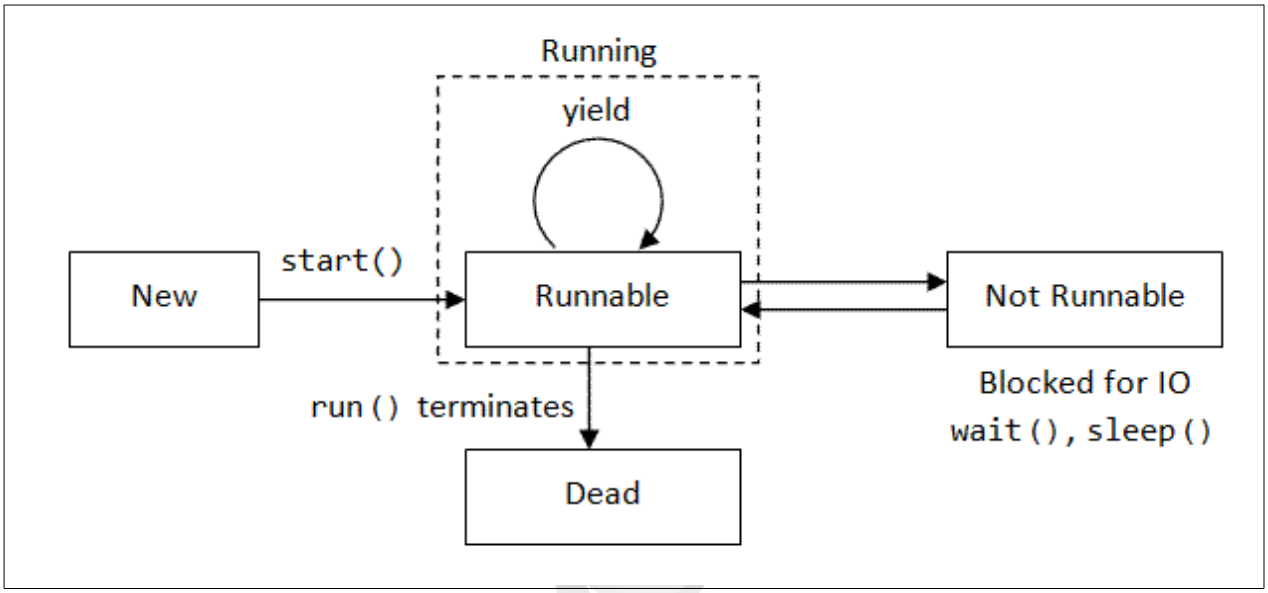

Figura 5. Ciclo de vida das threads

12.2 Criando uma Thread

A Linguagem Java oferece duas maneiras para se criar uma thread. A primeira é estender a classe Thread e a outra maneira é implementar a interface Runnable. A diferença está na possibilidade de estender a thread criada. Quando a thread é criada pela extensão da classe Thread do Java ela não poderá mais ser extendida.

O exemplo abaixo mostra como criar uma thread que estende a classe Thread.

```
public class ThreadSample extends Thread {

  public void run() {
    String nomeDaThread = this.getName();
    for (int i=0; i < 10; i++) {
      System.out.println(nomeDaThread + ": " + i);
    }
  }

}
```

```
public class DemoThreadSample {

  public static void main(String[] args) {

    ThreadSample th1 = new ThreadSample();
    th1.setName("Thread 1");

    ThreadSample th2 = new ThreadSample();
    th2.setName("Thread 2");

    th1.run();
    th2.run();

  }

}
```

O exemplo abaixo mostra como criar uma thread que implemente a interface Runnable.

```
public class RunnableSample implements Runnable {

  @Override
  public void run() {
    String nomeDaThread = Thread.currentThread().getName();
    for (int i=0; i < 10; i++) {
      System.out.println(nomeDaThread + ": " + i);
     }
  }

}
```

```
public class DemoRunnableSample {

  public static void main(String[] args) {

    RunnableSample rs = new RunnableSample();

    Thread th1 = new Thread(rs);
    th1.setName("Thread 1");

    Thread th2 = new Thread(rs);
    th2.setName("Thread 2");

    th1.start();
```

```
    th2.start();

  }

}
```

12.3 Exercícios sobre Threads

Exercício 01 – Servidor de Configuração

Implementar thread no servidor de configuração para que o servidor aceite e gerencie multíplas conexões.

12.4 Sincronização de Threads

Quando existem duas ou mais threads em execução e elas precisam compartilhar o uso de algum recurso, tal como um arquivo em disco ou uma lista de objetos na memória, o recurso deve estar sincronizado para evitar conflitos de acesso, uma vez que as threads são executadas paralelamente.

Para sincronizar um determinado recurso basta acrescentar a palavra *synchronized* no início da assinatura do método que prove acesso ao recurso. O exemplo abaixo mostra como aplicar o sincronismo entre threads.

```
public class Somar {

  private int soma;

  synchronized int somar(int numeros[]) {
    soma = 0;

    for (int i=0; i < numeros.length; i++) {
      soma += numeros[i];

      try {
        Thread.sleep(10);
      } catch (InterruptedException e) {
        System.out.println("Thread interrompida.");
      }
    }

    return soma;
  }
```

```
}
```

```
public class ThreadSoma extends Thread {

  private static Somar somar = new Somar();
  private int numeros[];
  private int resposta;

  public ThreadSoma(int numeros[]) {
    this.numeros = numeros;
  }

  public void run() {
    System.out.println("Iniciando a thread " +
            Thread.currentThread().getName());

    resposta = somar.somar(numeros);

    System.out.println("Total da soma: " + resposta);

    System.out.println("Finalizando a thread " +
            Thread.currentThread().getName());
  }
}
```



```
public class SincronismoSample {

  public static void main(String[] args) {

    int numeros1[] = {1, 2, 3, 4, 5};
    int numeros2[] = {5, 6, 7, 8, 9};

    ThreadSoma t1 = new ThreadSoma(numeros1);
    t1.setName("Thread 1");

    ThreadSoma t2 = new ThreadSoma(numeros2);
    t2.setName("Thread 2");

    t1.start();
    t2.start();

  }
}
```

12.5 Exercícios sobre Sincronização de Threads

Exercício 01 – Simulador Fila Atendimento

Implementar um simulador de fila de atendimento utilizando threads.

CAPÍTULO 13

JAVA 7

JAVA II

A versão 7 da Linguagem Java trouxe diversas novidades e melhorias na linguagem que quando aplicadas contribuem para um código mais limpo e mais fácil de ser entendido.

13.1 Separador de Literais

Para facilitar a leitura de literais numéricos longos é possível adicionar o caractere underscore (_) como separador. Dessa forma, um número como 99999999 poderia ser escrito como 99_999_999. Durante a compilação, o caractere underscore é ignorado e o número é interpretado normalmente como se o separador não existisse.

O exemplo abaixo mostra outras situações onde o caractere underscore (_) é utilizado como separador de literais numéricos.

```
public class SeparadorLiteraisNumericos {

  public static void main(String[] args) {

    byte b =  1_2_7;
    int v1 = 999_999;
    int v2 = 100_000 - 1_000;
    Integer i1 = new Integer(12_12_12_1);
    Integer i2 = new Integer(1_2_1_2_1_2_1);
    double f = 123_456.3_4_1;

    System.out.printf("O valor de b é: %d\n", b);
    System.out.printf("O valor de v1 é: %d\n", v1);
    System.out.printf("O valor de v2 é: %d\n", v2);
    System.out.printf("i1 e i2 são iguais?: %b\n",
          i1.equals(i2));
    System.out.printf("O valor de f é: %.4f\n", f);
  }

}
```

13.2 Literais Binários

Até a versão 6 da Linguagem Java era possível trabalhar apenas com literais inteiros definidos como octais, decimais e hexadecimais. A partir da versão 7 é possível definir literais binários apenas com a adição do '0b' (zero e a letra 'b') antes de uma sequência de 0s e 1s. O exemplo abaixo mostra como defir literais binários.

```
public class LiteraisBinarios {

  public static void main(String[] args) {

    int binario1 = 0b1100;  // número 12
    int binario2 = 0b01;    // número 1
    int binario3 = 0b1101;  // número 13

    System.out.println(binario1);
    System.out.println(binario2);
    System.out.println(binario3);
  }

}
```

13.3 Strings em Estruturas Switch

Até a versão 6 da Linguagem Java a estrutura de decisão switch aceitava apenas os tipos primitivos char, short e int. O Java 7 permite a utilização de strings na estrutura switch. O exemplo abaixo mostra como utilizar strings em estrutura switch.

```
public class SwitchString {

  public static void main(String[] args) {

    String param = args[0];

    switch (param) {
      case "-help":
        System.out.println("Mostrar help");
        break;
      case "-verbose":
        System.out.println("Exibir saída detalhada");
        break;
      default:
        System.err.printf("Parâmetro %s não reconhecido\n",
            param);
        System.exit(-1);
    }
  }
}
```

13.4 Inferência em Tipos Genéricos

O recurso chamado Generics introduzido pela versão 5 da Linguagem Java trouxe segurança na definição de listas de objetos mas também trouxe complexidade.

O exemplo abaixo mostra como definir uma lista de objetos do tipo HashMap até a versão 6 do Java.

```
List<HashMap<String, Object>> items =
        new ArrayList<HashMap<String, Object>>();
```

A mesma lista de objeto do tipo HashMap pode ser reescrita da seguinte forma na versão 7 da Linguagem Java.

```
List<HashMap<String, Object>> items = new ArrayList<>();
```

13.5 Comando try com multi-catch

Até a versão 6 da Linguagem Java um comando try..catch deveria capturar explicitamente cada uma das exceções disparadas pelos comandos executados dentro do try. Por exemplo, se dentro de um comando try..catch existir comandos que disparem exceções do tipo ClassNotFoundException e SQLException, o bloco try..catch deveria ter um tratamento para a exceção do tipo ClassNotFoundException e um outro catch para tratar a exceção do tipo SQLException. O exemplo abaixo mostra essa situação.

```
public class TryCatchJava6 {
  public static void main(String[] args) {
    try {
      Class.forName("...");
      DriverManager.getConnection("...");
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

A versão 7 do Java adicionou a possibilidade de definir um comando try..catch para tratar múltiplas exceções. Isso pode simplificar muito o código quando o tratamento de exceções é o mesmo para as diversas exceções tratadas pelo try..catch.

O exemplo abaixo mostra como implementar um comando try..catch para tratar múltiplas exceções.

```
public class TryCatchJava7 {
  public static void main(String[] args) {
    try {
      Class.forName("...");
      DriverManager.getConnection("...");
    } catch (ClassNotFoundException | SQLException e) {
      e.printStackTrace();
    }
  }
}
```

13.6 Comando try-with-resources

Até a versão 6 da Linguagem Java quando um comando try..catch era utilizado para tratar exceções geradas por comandos que utilizem recursos tais como arquivos em disco, banco de dados e conexões socket era necessário a adição do bloco finally ao comando try..catch para garantir que o recurso alocado dentro do comando try fosse liberado mesmo quando uma exceção fosse disparada.

A versão 7 do Java trouxe um recurso chamado try-with-resources que dispensa o uso do bloco finally. O exemplo abaixo mostra como utilizar o try-with-resources.

```
public class TryWithResources {
  public static void main(String[] args) {
    try (FileInputStream fis = new FileInputStream("..."))
    {
      // Utilizar o objeto 'fis'
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

A utilização do recurso try-with-resources é possível apenas em classes que implementam a interface Closeable que estende AutoCloseable. No exemplo acima a classe FileInputStream estende a classe InputStream que implementa a interface Closeable.

13.7 Classe Files

A classe Files é uma evolução da classe File, pertence ao pacote java.nio.file e contém diversos métodos estáticos para manipulação de diretórios e arquivos. Muitos métodos que na classe File retornavam apenas um valor booleano, agora na classe Files disparam exceções sinalizando o motivo da falha da operação.

13.8 Copiando Arquivo com a Classe Files

O exemplo abaixo mostra como copiar um arquivo de um diretório para um outro diretório utilizando o método copy da classe Files. Observação: o método copy não existia na classe File.

```
public class FilesCopySample {

  public static void main(String[] args) {

    Path file1 = Paths.get("/home/aluno/curso/dir1/texto.txt");
    Path file2 = Paths.get("/home/aluno/curso/dir2/arquivo.txt");

    try {
      Files.copy(file1, file2);
      System.out.println("Arquivo copiado com sucesso!");
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

O método Files.copy() disparará uma exceção caso o arquivo chamado arquivo.txt já existir no diretório /home/aluno/dir2. Para sobrescrever o arquivo é necessário passar o parâmetro REPLACE_EXISTING para o método Files.copy() como mostra o exemplo abaixo.

```
public class FilesCopyReplaceSample {

  public static void main(String[] args) {

    Path file1 = Paths.get("/home/aluno/curso/dir1/texto.txt");
    Path file2 = Paths.get("/home/aluno/curso/dir2/arquivo.txt");

    try {
      Files.copy(file1, file2,
```

```
                StandardCopyOption.REPLACE_EXISTING);
      System.out.println("Arquivo copiado com sucesso!");
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

13.9 Notificações de Alterações em Diretórios

Um recurso muito interessante introduzido na versão 7 da Linguagem Java é a classe WatchService para recebimento de notificações referentes a alterações em um diretório. É possível saber quando um determinado arquivo foi alterado, quando foi excluído um arquivo ou quando foi criado um nome arquivo no diretório.

O exemplo abaixo mostra como utilizar a classe WatchService para monitorar alterações em um determinado diretório.

```
import java.io.IOException;
import java.nio.file.FileSystems;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.WatchEvent;
import java.nio.file.WatchEvent.Kind;
import java.nio.file.WatchKey;
import java.nio.file.WatchService;

import static java.nio.file.StandardWatchEventKinds.*;

public class WatchServiceSample {

  private Path path = null;
  private WatchService watchService = null;

  private void init() {
    path = Paths.get("/home/eros/curso");
    try {
      watchService = FileSystems.getDefault().newWatchService();
      path.register(watchService, ENTRY_CREATE, ENTRY_DELETE,
              ENTRY_MODIFY);
    } catch (IOException e) {
      System.out.println("IOException" + e.getMessage());
    }
  }
```

```
  private void doRounds() {
    WatchKey key = null;
    while (true) {
      try {
        key = watchService.take();
        for (WatchEvent<?> event : key.pollEvents()) {
          Kind<?> kind = event.kind();
          System.out.println("Event on " +
                  event.context().toString() + " is " + kind);
        }
      } catch (InterruptedException e) {
        System.out.println("InterruptedException: " +
                e.getMessage());
      }
      boolean reset = key.reset();
      if (!reset)
        break;
    }
  }

  public static void main(String[] args) {
    WatchServiceSample ws = new WatchServiceSample();
    ws.init();
    ws.doRounds();
  }

}
```

13.10 Exercícios sobre Java 7

Exercício 01 – Caixa ATM

Aplicar os novos recursos do Java 7 no software CaixaATM de modo a tornar o código mais simples.

Exercício 02 – Comparador de Arquivos

Criar um programa para comparar dois arquivos. Os arquivos deverão ser passados por linha de comando.

CAPÍTULO 14

JAVA 8

Assim como as outras versões da Linguagem Java, o Java 8 trouxe diversos recursos e melhorias para a linguagem e para a máquina virtual. Os recursos mais esperados para a versão 8 eram, sem dúvidas, a nova API para Data e Hora e as expressões lambda.

Trabalhar com data e hora até a versão 7 do Java é muito trabalhoso e tedioso, havia duas versões da classe Date em pacotes diferentes (java.util.Date e java.sql.Date), para obter a hora atual deveria ser utilizado a classe Date (não existia uma classe específica para trabalhar com horas), qualquer operação envolvendo data e hora deveria ser feito numa segunda classe chamada Calendar.

Com a introdução da nova API para tratar datas e horas, o Java 8 trouxe consistência na definição de classes para trabalhar com data, hora e data hora. Todas as classes da nova API são imutáveis, ou seja, são thread-safe.

Outra novidade muito aguardada pela comunidade de desenvolvedores era as expressões lambda. O termo lambda vem do cálculo lambda que é uma área da matemática que estuda funções recursivas computáveis que podem ser armazenadas em variáveis, passadas como parâmetros e retornadas como valores de outras funções.

14.1 Nova API Data e Hora

Na nova API de Data e Hora do Java 8 existe uma classe específica para trabalhar com data, outra classe específica para trabalhar com hora e uma terceira classe para trabalhar com data e hora. Todas as classes disponibilizam métodos para operações tais como adicionar dias a uma data, extrair os minutos de uma hora e extrair partes de uma data.

O exemplo abaixo mostra como utilizar a nova API de data e hora do Java 8.

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.time.format.DateTimeFormatter;
import java.time.temporal.TemporalAdjusters;

public class DateSample {

  public static void main(String[] args) {
```

```
    // Pegando data atual
    LocalDate dataAtual = LocalDate.now();
    System.out.println(dataAtual);

    // Definindo uma data
    LocalDate nascimento = LocalDate.parse("2010-10-10");
    System.out.println(nascimento);

    // Pegando hora atual
    LocalTime horaAtual = LocalTime.now();
    System.out.println(horaAtual);

    // Definindo uma hora
    LocalTime horaNascimento = LocalTime.parse("15:20:30");
    System.out.println(horaNascimento);

    // Pegando data e hora atual
    LocalDateTime dataHoraAtual = LocalDateTime.now();
    System.out.println(dataHoraAtual);

    // Data e hora com fuso horario
    ZoneId fusoHorarioNovaYork = ZoneId.of("America/New_York");
    ZonedDateTime agoraEmNovaYork =
         ZonedDateTime.now(fusoHorarioNovaYork);
    System.out.println("Data/Hora em Nova York: " +
         agoraEmNovaYork);

    // Operações com data
    LocalDate data = LocalDate.parse("2018-02-01");

    int diaSemana = data.getDayOfWeek().getValue();
    int diaMes = data.getDayOfMonth();
    int diaAno = data.getDayOfYear();
    System.out.printf("Dia semana: %d, Dia mês: %d,
         Dia ano: %d\n", diaSemana, diaMes, diaAno);

    int ultimoDiaMes = data.with(TemporalAdjusters
         .lastDayOfMonth()).getDayOfMonth();
    System.out.println("Ultimo dia mês: " + ultimoDiaMes);

    // Formatando data
    LocalDate hoje = LocalDate.now();
    DateTimeFormatter formatador =
         DateTimeFormatter.ofPattern("dd/MM/yyyy");
```

```
    System.out.println("Hoje: " + hoje.format(formatador));
  }

}
```

14.2 Expressões Lambda

As expressões lambda podem ser aplicadas em diversas situações e quando aplicadas muitas vezes tornam o código mais simples e fácil de entender. É certo que algumas construções usando expressões lambda podem ser tão complexas e difícil de entender que a utilização pode ficar comprometida.

14.2.1 Threads e Expressões Lambda

Threads podem ser definidas através de expressões lambda. O exemplo abaixo mostra uma definição padrão de uma thread que mostra uma mensagem simples na tela do computador.

```
Runnable r = new Runnable() {
  public void run() {
    System.out.println("Uma thread simples.");
  }
};

new Thread(r).start();
```

O código acima pode ser reescrito da seguinte forma utilizando expressão lambda.

```
Runnable r = () -> System.out.println("Uma thread simples!");
new Thread(r).start();
```

14.2.2 Collections e Expressões Lambda

A manipulação de coleções de objetos pode ser simplificada com a utilização de expressões lambda. O exemplo abaixo mostra uma lista de objetos do tipo inteiro de forma tradicional.

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6);
for(Integer n: numeros) {
  System.out.println(n);
}
```

O código acima pode ser reescrito da seguinte forma utilizando expressão lambda.

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6);
numeros.forEach(n -> System.out.println(n));
```

Dentro do corpo da expressão lambda é possível executar mais de um comando. O exemplo abaixo mostra como imprimir apenas os números pares da lista.

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6);
numeros.forEach(n -> {
if (n % 2 == 0)
  System.out.println(n);
});
```

14.2.3Ordenação de Coleções de Objetos e Expressões Lambda

A definição de critérios para ordenação de coleções de objetos normalmente é complexa e trabalhosa. O exemplo abaixo mostra como ordenar uma lista de objetos do tipo Pessoa usando a classe Comparator de forma anônima.

```
List<Pessoa> pessoas = Arrays.asList(
      new Pessoa("Caio", 25),
      new Pessoa("Pedro", 38),
      new Pessoa("Lucas", 23));

System.out.println("Ordenando pelo nome...");
Collections.sort(pessoas, new Comparator<Pessoa>() {
  @Override
  public int compare(Pessoa pessoa1, Pessoa pessoa2){
    return pessoa1.getNome().compareTo(pessoa2.getNome());
  }
});
pessoas.forEach(p -> System.out.println(p.getNome()));

System.out.println("Ordenando pela idade...");
Collections.sort(pessoas, new Comparator<Pessoa>() {
  @Override
  public int compare(Pessoa pessoa1, Pessoa pessoa2){
    return pessoa1.getIdade().compareTo(pessoa2.getIdade());
  }
});
pessoas.forEach(p -> System.out.println(p.getNome()));
```

O código acima pode ser reescrito de uma forma mais simples usando expressão lambda da seguinte forma:

```
List<Pessoa> pessoas = Arrays.asList(
      new Pessoa("Caio", 25),
      new Pessoa("Pedro", 38),
```

```
            new Pessoa("Lucas", 23));

      System.out.println("Ordenando pelo nome:");
      Collections.sort(pessoas, (Pessoa pessoa1, Pessoa pessoa2) ->
            pessoa1.getNome().compareTo(pessoa2.getNome()));
      pessoas.forEach(p -> System.out.println(p.getNome()));

      System.out.println("Ordenando pela idade...");
      Collections.sort(pessoas, (Pessoa pessoa1, Pessoa pessoa2) ->
            pessoa1.getIdade().compareTo(pessoa2.getIdade()));
      pessoas.forEach(p -> System.out.println(p.getNome()));
```

14.2.4Filtrando Objetos em Coleções com Expressões Lambda

Coleções de objetos podem ser facilmente filtradas com a utilização de expressões lambda. O exemplo abaixo mostra como utilizar uma expressão lambda para filtar uma lista de objetos.

```
      List<Pessoa> pessoas = Arrays.asList(
            new Pessoa("Caio", 25),
            new Pessoa("Pedro", 38),
            new Pessoa("Lucas", 23));

      System.out.println("Pessoas com mais de 30 anos...");
      List<Pessoa> maioresTrinta = pessoas.stream().filter(p ->
           p.getIdade() > 30).collect(Collectors.toList());
      maioresTrinta.forEach(p -> System.out.println(p.getNome()));

      System.out.println("Pessoas cujo nome inicia com L:");
      List<Pessoa> nomesIniciadosL = pessoas.stream().filter(p ->
            p.getNome().startsWith("L"))
            .collect(Collectors.toList());
      nomesIniciadosL.forEach(p ->
            System.out.println(p.getNome()));
```

14.2.5Listeners e Expressões Lambda

Listeners são classes que implementam o padrão de projeto chamado Observer e são normalmente implementados como classes anônimas, tornando o código complexo muitas vezes.

O exemplo abaixo mostra a implementação de um listener para um objeto do tipo JButton da API Swing.

```
button.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent e) {
        System.out.println("O botão foi pressionado!");
        //TODO Fazer mais alguma coisa
      }
});
```

O código acima pode ser reescrito de uma forma mais simples usando expressão lambda da seguinte forma:

```
button.addActionListener( (e) -> {
      System.out.println("O botão foi pressionado!");
      //TODO Fazer mais alguma coisa
});
```

14.2.6 Expressões Lambda como Parâmetros de Métodos

Em Java 8 é possível criar métodos que recebem expressões lambda como parâmetros. Isto permite a criação de métodos génericos. O exemplo abaixo mostra como passar uma expressão lambda como parâmetro.

```
public static void imprimirNumeros(
      List<Integer> list, Predicate<Integer> predicate) {
  list.forEach(n -> {
    if (predicate.test(n)) {
      System.out.println(n + " ");
    }
  });
}

List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6);

System.out.println("Imprime todos os números:");
imprimirNumeros(numeros, (n)->true);

System.out.println("Imprime apenas número pares:");
imprimirNumeros(numeros, (n)-> n%2 == 0);

System.out.println("Imprime apenas números maiores que 5:");
imprimirNumeros(numeros, (n)-> n > 5);
```

14.3 Exercícios sobre novidades do Java 8

Exercício 01 – Caixa ATM

JAVA II

Aplicar os novos recursos do Java 8 no software CaixaATM de modo a tornar o código mais simples.

CAPÍTULO 15

REFERÊNCIAS

SCHILDT, Herbet. Java para iniciantes. 6ª ed. Porto Alegre: Bookman, 2015.

Rua Monsenhor Celso, 256 - 1º andar - Curitiba – Paraná

41.3324.0015

41.99828.2468

Agradecimentos:

Equipe Elaborata Informática