Módulo 5

Desenvolvimento de Aplicações Móveis

Lição 5

Sistema de Gerenciamento de Registro

Versão 1.0 - Set/2007

Autor
xxx

Equipe
Rommel Feria
John Paul Petines

Necessidades para os Exercícios

Sistemas Operacionais Suportados

NetBeans IDE 5.5 para os seguintes sistemas operacionais:

- Microsoft Windows XP Profissional SP2 ou superior
- Mac OS X 10.4.5 ou superior
- Red Hat Fedora Core 3
- Solaris™ 10 Operating System (SPARC® e x86/x64 Platform Edition)

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

- Microsoft Windows 2000 Profissional SP4
- Solaris™ 8 OS (SPARC e x86/x64 Platform Edition) e Solaris 9 OS (SPARC e x86/x64 Platform Edition)
- Várias outras distribuições Linux

Configuração Mínima de Hardware

Nota: IDE NetBeans com resolução de tela em 1024x768 pixel

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	500 MHz Intel Pentium III workstation ou equivalente	512 MB	850 MB
Linux	500 MHz Intel Pentium III workstation ou equivalente	512 MB	450 MB
Solaris OS (SPARC)	UltraSPARC II 450 MHz	512 MB	450 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Série 1.8 GHz	512 MB	450 MB
Mac OS X	PowerPC G4	512 MB	450 MB

Configuração Recomendada de Hardware

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	1 GB
Linux	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	850 MB
Solaris OS (SPARC)	UltraSPARC IIIi 1 GHz	1 GB	850 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Series 1.8 GHz	1 GB	850 MB
Mac OS X	PowerPC G5	1 GB	850 MB

Requerimentos de Software

NetBeans Enterprise Pack 5.5 executando sobre Java 2 Platform Standard Edition Development Kit 5.0 ou superior (JDK 5.0, versão 1.5.0_01 ou superior), contemplando a Java Runtime Environment, ferramentas de desenvolvimento para compilar, depurar, e executar aplicações escritas em linguagem Java. Sun Java System Application Server Platform Edition 9.

- Para **Solaris**, **Windows**, e **Linux**, os arquivos da JDK podem ser obtidos para sua plataforma em http://java.sun.com/j2se/1.5.0/download.html
- Para **Mac OS X**, Java 2 Plataform Standard Edition (J2SE) 5.0 Release 4, pode ser obtida diretamente da Apple's Developer Connection, no endereço: http://developer.apple.com/java (é necessário registrar o download da JDK).

Para mais informações: http://www.netbeans.org/community/releases/55/relnotes.html

Colaboradores que auxiliaram no processo de tradução e revisão

Aécio Júnior
Alexandre Mori
Alexis da Rocha Silva
Allan Souza Nunes
Allan Wojcik da Silva
Anderson Moreira Paiva
Andre Neves de Amorim
Angelo de Oliveira
Antonio Jose R. Alves Ramos
Aurélio Soares Neto
Bruno da Silva Bonfim
Carlos Fernando Gonçalves
Denis Mitsuo Nakasaki
Fábio Bombonato
Fabrício Ribeiro Brigagão
Francisco das Chagas
Frederico Dubiel
Herivelto Gabriel dos Santos
Jacqueline Susann Barbosa
João Vianney Barrozo Costa
Kefreen Ryenz Batista Lacerda
Kleberth Bezerra G. dos Santos
Leandro Silva de Morais
Leonardo Ribas Segala
Lucas Vinícius Bibiano Thomé
Luciana Rocha de Oliveira
Luiz Fernandes de Oliveira Junior
Marco Aurélio Martins Bessa
Maria Carolina Ferreira da Silva
Massimiliano Giroldi
Mauro Cardoso Mortoni
Paulo Afonso Corrêa
Paulo Oliveira Sampaio Reis
Pedro Henrique Pereira de Andrade
Ronie Dotzlaw
Seire Pareja
Sergio Terzella
Vanessa dos Santos Almeida
Robson Alves Macêdo

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

- **Brasil** – Tiago Flach
- **Guiné Bissau** – Alfredo Cá, Bunene Sisse e Buon Olossato Quebi – ONG Asas de Socorro

Coordenação do DFJUG

- **Daniel deOliveira** – JUGLeader responsável pelos acordos de parcerias
- **Luci Campos -** Idealizadora do DFJUG responsável pelo apoio social
- **Fernando Anselmo -** Coordenador responsável pelo processo de tradução e revisão, disponibilização dos materiais e inserção de novos módulos
- **Rodrigo Nunes -** Coordenador responsável pela parte multimídia
- **Sérgio Gomes Veloso -** Coordenador responsável pelo ambiente JEDI™ (Moodle)

Agradecimento Especial

John Paul Petines – Criador da Iniciativa JEDI™
Rommel Feria – Criador da Iniciativa JEDI™

1. Objetivos

MIDP fornece uma biblioteca de componentes onde os programas podem salvar os dados da aplicação no dispositivo de modo físico. O Sistema de Gerenciamento de Registro do *MIDP* é o meio pelo qual as *MIDlets* podem armazenar dados através de chamadas *MIDlet*. Os dados são armazenados na memória não-volátil do dispositivo, ou seja, não serão perdidos mesmo que o programa seja finalizado ou o aparelho seja desligado.

O Sistema de Gerenciamento de Registro é um sistema de banco de dados orientados a registro para dispositivos *MIDP*. Um registro neste banco de dados é simplesmente um array de bytes com um índice.

Ao final desta lição, o estudante será capaz de:

- Entender o conceito de *RecordStore*
- Criar ou abrir um *RecordStore*
- Adicionar, recuperar, atualizar e excluir registros
- Fechar um *RecordStore*
- Enumerar registros utilizando um *RecordEnumerator*
- Criar um *RecordComparator*
- Criar um *RecordFilter*

2. RecordStore

Record Store é uma coleção de registros, a classe é encontrada no pacote *javax.microedition.rms*. Identificadores de registro (*Record ID*) em um banco de dados são únicos. Esses identificadores são automaticamente alocados no momento da criação do registro e funcionam como um índice ou chave primária. São atribuídos sequencialmente, sendo que o primeiro ID de cada banco de dados possui o valor igual a 1 (um).

Quando um registro é excluído, seu identificador não será reutilizado. Se criarmos quatro registros e excluirmos o quarto, o próximo identificador que o sistema atribuirá será igual a 5 (ver Figura).

Record ID	Array de bytes
1	Dado do registro #1
2	Dado do registro #2
3	Dado do registro #3
5	Dado do registro #5

Aplicativos *MIDlets* podem criar mais de um objeto *RecordStore*. O nome do *RecordStore* deve ser único dentro de uma suite *MIDlet*. São sensíveis ao contexto (maiúsculas diferentes de minúsculas) e devem possuir um tamanho máximo de 32 caracteres.

Quando uma aplicação *MIDlet* é eliminada de um dispositivo, todos os *RecordStore* associados a esta aplicação também serão eliminados.

2.1. Criando ou Abrindo um RecordStore

Os métodos listados abaixo criam ou abrem para uso um *RecordStore*:

```
static RecordStore openRecordStore(
   String recordStoreName,
   boolean createIfNecessary)
static RecordStore openRecordStore(
   String recordStoreName,
   boolean createIfNecessary,
   int authmode,
   boolean writable)
static RecordStore openRecordStore(
   String recordStoreName,
   String vendorName,
   String suiteName)
```

Caso o parâmetro *createIfNecessary* possuir o valor *true* e o *RecordStore* não existir, este será automaticamente criado. Se o parâmetro possuir o valor *false* e o *RecordStore* não existir, uma exceção do tipo *RecordStoreNotFoundException* será lançada.

O parâmetro *authmod* poderá assumir um dos seguintes atributos constantes existentes: *RecordStore.AUTHMODE_PRIVATE* ou *RecordStore.AUTHMODE_ANY*. Utilizando o primeiro atributo o *RecordStore* será acessível apenas pela aplicação *MIDlet* que o criou. Utilizando o segundo atributo o *RecordStore* será acessível por qualquer aplicação *MIDlet*. O modo de acesso é especificado pelo parâmetro lógico *writable*. Para permitir que outros *MIDlets* (fora do aplicativo que o criou) armazenem ou leiam dados neste *Record Store*, este parâmetro deverá ser informado com o valor *true*.

Utilizando a primeira forma do método *openReacordStore()* o resultado obtido é que o *RecordStore* será acessível somente aos *MIDlets* da mesma aplicação (o valor do atributo *authmode* terá o mesmo valor da constante AUTHMODE_PRIVATE).

Para abrir um *RecordStore* existente a partir de uma aplicação *MIDlet* diferente, a terceira forma do método *openRecordStore* é utilizada. Os parâmetros *vendorName* e *suiteName* deverão ser

especificados contendo o nome do desenvolvedor e o nome do aplicativo.

Se o *RecordStore* já estiver aberto, estes métodos irão retornar sua referência deste. O sistema mantém armazenado quantas vezes foram abertos os *RecordStore*. É obrigatório que *RecordStore* seja fechado o mesmo número de vezes que tenha sido aberto.

2.2. Adicionando Registros

O seguinte método é responsável pelas adições de novos registros em um *RecordStore*:

```
int addRecord(byte[] data, int offset, int numBytes)
```

O método *addRecord* criará um novo registro no banco de dados e retornará o seu identificador (*Record ID*). As informações recebidas como parâmetro neste método são:

1. Um array de bytes contendo os dados a serem armazenados
2. Um atributo inteiro contendo a posição inicial do array enviado
3. A quantidade total de elementos do array

2.3. Recuperando Registros

Os seguintes métodos são responsáveis pelas buscas de dados em um *Record Store:*

```
byte[] getRecord(
   int recordId)
int getRecord(
   int recordId, byte[] buffer, int offset)
int getRecordSize(
   int recordId)
```

A primeira forma do método *getRecord* devolve uma cópia dos dados armazenados no registro com a *Record ID* especificada pelo parâmetro. A segunda forma copiará os dados no parâmetro (array de bytes) fornecido Ao utilizar esta forma, o array de bytes precisa estar previamente alocado. Caso o tamanho do registro seja maior que o tamanho do parâmetro uma exceção *ArrayIndexOutOfBoundsException* será lançada. O método *getRecordSize* é utilizado de modo a obter o tamanho do registro que será lido.

2.4. Modificando Registros

Não se pode modificar somente uma parte do registro. Para modificar um registro é necessário:

1. Ler o registro utilizando o método *getRecord*
2. Modificar o registro na memória
3. Chamar o método *setRecord* para modificar os dados do registro

O método *setRecord* possui a seguinte assinatura:

```
void setRecord(
   int recordId, byte[] newData, int offset, int numBytes)
```

2.5. Eliminando Registros

O seguinte método é responsável pelas eliminações dos dados em um *RecordStore*:

```
void deleteRecord(
   int recordId)
```

Quando um registro é eliminado, o ID deste registro não será reutilizado para as próximas inclusões. Isto significa que podem ocorrer buracos entre os registros. Deste modo não é aconselhável utilizar um incrementador para listar todos os registros de um *Record Store*. Um

RecordEnumerator deve ser utilizado para percorrer a lista de registros guardados.

2.6. Fechando um RecordStore

O seguinte método é responsável pelo fechamento de um *RecordStore*:

```
void closeRecordStore()
```

Ao chamar o método *closeRecordStore*(), o *RecordStore* não será fechado até que seu valor seja idêntico ao número de chamadas do método *openRecordStore*(). O sistema armazena o número de vezes que um *Record Store* foi aberto. Caso o número de chamadas ao método *closeRecordStore*() sejam superiores ao número de chamadas ao método *openRecordStore*(), será disparada uma exceção do tipo *RecordStoreNotOpenException*.

Este trecho de código mostra um exemplo de um MIDlet simples que exemplifica a criar, adição e localização de dados em um *RecordStore*:

```
// Abre ou Cria um RecordStore com o nome "RmsExample1"
recStore = RecordStore.openRecordStore("RmsExample1", true);

// Lê o conteúdo do RecordStore
for (int recId=1; recId <= recStore.getNumRecords(); recId++){
  // O método getRecord returna o tamanho do registro
  recLength = recStore.getRecord(recId, data, 0);

  // Converte para String um array de bytes
  String item = new String(data, 0, recLength);
  ...
}

// Este será o dado a ser guardado
String newItem = "Record #" + recStore.getNextRecordID();

// Converte a String para o array de bytes
byte[] bytes = newItem.getBytes();

// Grava o dado no RecordStore
recStore.addRecord(bytes, 0, bytes.length);
```

Dicas de Programação:

1. O Registro começa com o valor de ID igual a 1, e não 0. Deve-se tomar cuidado ao se utilizar laços.
2. É melhor utilizar um *RecordEnumerator* do que um índice incrementado (como neste exemplo). Registros eliminados ainda serão lidos por este exemplo e causará uma exceção do tipo *InvalidRecordIDException*.

2.7. Obtendo uma Lista de RecordStore de uma aplicação MIDlet

O seguinte método é responsável pela obtenção de uma lista de *RecordStore* existente:

```
static String [] listRecordStores()
```

O método retorna um conjunto de nomes de *RecordStore* que foram criados pela aplicação *MIDlet*. Se a aplicação não possuir nenhum *RecordStore* é retornado o valor null.

```
String[] storeNames = RecordStore.listRecordStores();
System.out.println("RecordStore for this MIDlet suite:");

for (int i=0; storeNames != null && i < storeNames.length; i++) {
  System.out.println(storeNames[i]);
}
```

Este é um exemplo de saída:

```
Record Stores for this MIDlet suite:
Prefs
RmsExample1
RmsExample2
```

A ordem dos nomes retornados não está definida e é dependente da implementação. Então, ao se desejar exibir os nomes ordenados alfabeticamente, primeiro deve-se realizar uma ordenação deste array.

2.8. Guardando Tipos Primitivos de Java

Até o momento, os dados que temos gravados e lidos no *RecordStore* são Strings. CLDC inclui as classes padrões para manipulação de tipos primitivos. Estas classes são da biblioteca padrão da Plataforma *Java Standard Edition* (Java SE).

Pode-se gravar tipos primitivos combinando as classes *ByteArrayOutputStream* e *DataOutputStream*. Para ler tipos primitivos (int, long, short, char, boolean, etc) pode-se também utilizar as classes *ByteArrayInputStream* e *DataInputStream*.

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
DataOutputStream dOut = new DataOutputStream(out);

// Armazenando um inteiro
dOut.writeInt(recStore.getNextRecordID() * recStore.getNextRecordID());

// Armazenando uma String
dOut.writeUTF("Record #" + recStore.getNextRecordID());

// Transformando para um array de bytes
byte[] bytes = out.toByteArray();

// Armazenando os dados no Record Store
recStore.addRecord(bytes, 0, bytes.length);
...

// Obtendo o próximo registro
byte[] recBytes = enumerator.nextRecord();

// Para resgatar os valores
ByteArrayInputStream in = new ByteArrayInputStream(recBytes);
DataInputStream dIn = new DataInputStream(in);
int count = dIn.readInt();
String item = dIn.readUTF();
```

2.9. Outros métodos para um RecordStore

Os seguintes métodos podem ser utilizados para obter dados de um *RecordStore*:

```
long getLastModified()
int getVersion()
```

O sistema armazena a informação de quando um *Record Store* foi modificado pela última vez. Este valor pode ser obtido através do método *getLastModified()* e retorna um valor do tipo *long*, que poderá ser utilizado pelo método *System.currentTimeMillis().*

O *RecordStore* mantém a informação da versão que pode ser obtida através do método *getVersion()*. Toda vez em que um registro é modificado, seu número de versão é atualizado. Utilizando-se qualquer um dos métodos *addRecord*, *setRecord* e *deleteRecord* o número de versão do objeto será incrementado.

Outros métodos que podem ser utilizados são:

`static void deleteRecordStore( String nomeRecordStore)`	Eliminar a *RecordStore* que contenha o nome fornecido
`String getName()`	Retornar o nome do *RecordStore*

`int getNextRecordID()`	Retornar o recordId do próximo registro a ser adicionado ao objeto *RecordStore*
`int getNumRecords()`	Retornar o número de registros atualmente no *RecordStore*
`int getSize()`	Retornar a quantidade de bytes ocupados em um *RecordStore*
`int getSizeAvailable()`	Retornar a quantidade de bytes disponíveis em um *RecordStore*

`void setMode(`
`  int authmode,`
`  boolean writable)`
	Modificar o modo de acesso de um *RecordStore*

3. Enumeração de Registro

Percorrer registros incrementando um índice é uma forma ineficiente de agir. O *RecordStore* pode ter seus registros eliminados que geram buracos no *RecordId*.

A utilização de um *RecordEnumeration* resolve este problema sem ter de tratar os registros excluídos. É possível também ordenar um objeto do tipo *RecordEnumeration* utilizando-se o método c*omparator*(). Utilizando-se o método *filter*(), pode-se saltar os registros que não são importantes para o resultado.

```
RecordEnumeration enumerateRecords(
   RecordFilter filter,
   RecordComparator comparator,
   boolean keepUpdated)
```

O método *enumerateRecords* de um *RecordStore* retornará um objeto do tipo *RecordEnumeration*, no qual é permitido obter todos os registros de um *RecordStore*. Essa é a maneira recomendada para percorrer todos os registros armazenados. Iremos discutir os objetos *filter* e *comparator* nas próximas seções.

O modo mais simples para utilizar esse método é informar o valor *null* para os parâmetros *filter* e *comparator*. Obteremos como retorno da execução deste método um objeto do tipo *RecordEnumeration* de todos os registros do *Record Store* ordenados de uma maneira indefinida.

```
// Abrir ou criar um Record Store com o nome "RmsExample2"
recStore = RecordStore.openRecordStore("RmsExample2", true);

// Carregar o conteúdo do Record Store
RecordEnumeration enumerator = recStore.enumerateRecords(null, null, false);
while (enumerator.hasNextElement()){
  // Obter o próximo registro e converter um array de byte em String
  String item = new String(enumerator.nextRecord());

  // Realiza qualquer instrução com o dado
  ...
}
```

4. Comparador de Registros

A ordenação de um objeto do tipo *RecordEnumeration* pode ser definida utilizando-se um *RecordComparator*. Um *RecordComparator* é passado para o método *enumerateRecords*. Caso seja necessário ordenar um objeto do tipo *RecordEnumeration*, isso pode ser feito definindo-se um objeto do tipo *RecordComparator* e enviando-o como o segundo parâmetro para o método *enumerateRecords*.

```
int compare(byte[] reg1, byte[] reg2)
```

Para criar um *RecordComparator*, é preciso primeiro implementar a interface *RecordComparator*. Essa interface define um único método *compare(byte[] reg1, byte[] reg2)*. Esse método precisa retornar *RecordComparator.FOLLOWS* ou *RecordComparator.PRECEDES* se *reg1* procede ou antecede *reg2* na ordem, respectivamente. *RecordComparator.EQUIVALENT* pode ser retornado se *reg1* for igual a *reg2* na ordem.

```
// Criar uma enumeration, ordenada alfabeticamente
RecordEnumeration enumerator = recStore.enumerateRecords(
  null, new AlphaOrder(), false);

// Classe que ordena alfabeticamente
class AlphaOrder implements RecordComparator {
  public int compare(byte[] reg1, byte[] reg2){
    String registro1 = new String(reg1).toUpperCase();
    String registro2 = new String(reg2).toUpperCase();
    if (registro1.compareTo(registro2) < 0){
      return(RecordComparator.PRECEDES);
    } else {
      if (registro1.compareTo(registro2) > 0){
        return(RecordComparator.FOLLOWS);
      } else {
        return(RecordComparator.EQUIVALENT);
      }
    }
  }
}
```

5. Filtro de Registros

Os exemplos que vimos até o momento somente percorrem e lêem todos registros de um *Record Store*. Entretanto, podemos utilizar um *filter* para obtermos somente os registros que são necessários.

A classe necessita implementar o método *match()* para selecionar somente os registros válidos em um determinado contexto.

```
boolean matches(byte[] candidate)
```

O retorno é um atributo do tipo lógico que deve conter o valor *true* caso o registro seja compatível com o critério definido. Caso contrário é retornado *false*.

```
public boolean matches(byte[] candidate){
  boolean resultado = false;
  try {
    ByteArrayInputStream  bin = new ByteArrayInputStream(candidate);
    DataInputStream dIn = new DataInputStream(bin);
    int count = dIn.readInt();
    String item = dIn.readUTF();

    // Retornar somente registros com o conteúdo terminado com 0
    resultado = item.endsWith("0");
  } catch (Exception e) {
  }
  return(resultado);
}
```

6. Consumo de Combustível

Iremos ajuntar algumas idéias aprendidas até o momento e montar um aplicativo prático para controlar o consumo de combustível do nosso carro.

Ao pensarmos em aplicativos para aparelhos móveis devemos ter em mente as seguintes regras:

1. Praticidade
2. Facilidade

O usuário estará no posto abastecendo o carro. Ao acessar o aplicativo precisamos obter duas informações: a quilometragem atual do veículo e a quantidade de combustível inserida. Neste primeiro registro ainda não teremos dados suficientes para o cálculo do consumo, entretanto, a partir do segundo registro já poderemos calcular o consumo, obtido pela fórmula:

```
(quilometragem atual – quilometragem anterior) / quantidade combustível
```

Uma outra idéia interessante é guardar somente as últimas 11 entradas, deste modo mostraremos as 10 últimas médias de consumo. Antes de verificar as classes descritas abaixo, tente resolver este aplicativo.

A primeira classe chamada *PrincipalMidlet*. Conforme vimos anteriormente, o método *startApp*() fará a chamada a um objeto do tipo *Canvas*.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class PrincipalMidlet extends MIDlet {
  private MenuCanvas canvas;
  public Display display;

  public void startApp() {
    if (display == null) {
      canvas = new MenuCanvas(this);
      display = Display.getDisplay(this);
    }
    display.setCurrent(canvas);
  }
  protected void pauseApp() {
  }
  protected void destroyApp(boolean arg0) throws MIDletStateChangeException {
  }
}
```

A segunda classe, chamada *MenuCanvas*, do tipo *Canvas*, conterá toda a regra de negócio.

```
import javax.microedition.lcdui.*;
import javax.microedition.midlet.MIDletStateChangeException;
import javax.microedition.rms.*;

public class MenuCanvas extends Canvas implements CommandListener {

  private PrincipalMidlet pai;
  private InsertForm form;
  private byte opc = 0;
  private Command exitCmd = new Command("Exit", Command.EXIT, 0);
  private Command insertCmd = new Command("Insert", Command.OK, 1);
  private Command reportCmd = new Command("Report", Command.OK, 1);
  // Comandos para o Form
  private Command backCmd = new Command("Back", Command.BACK, 2);
  private Command saveCmd = new Command("Save", Command.OK, 1);
  // Atributos para o Registro
  private String regGas;
  private RecordStore rsGas;

  public MenuCanvas(PrincipalMidlet pai) {
```

```
      // Cria ou abre o registro
      try {
        rsGas = RecordStore.openRecordStore("Gasolina", true);
      } catch (RecordStoreException ex) {
        ex.printStackTrace();
      }
      this.pai = pai;
      setCommandListener(this);
      addCommand(exitCmd);
      addCommand(insertCmd);
      addCommand(reportCmd);
    }
    public void paint(Graphics g) {
      g.setColor(255, 255, 255);
      g.fillRect(0, 0, getWidth(), getHeight());
      g.setColor(0, 64, 128);
      switch (opc) {
        case 0:
          g.drawString("Principal Menu", 0, 0,
          Graphics.TOP | Graphics.LEFT);
          g.setColor(0, 114, 168);
          g.drawString("Use this program for the control", 0, 24,
            Graphics.TOP | Graphics.LEFT);
          g.drawString("the consumption of gasoline in your car", 0, 36,
            Graphics.TOP | Graphics.LEFT);
          g.drawString("Options:", 0, 64, Graphics.TOP | Graphics.LEFT);
          g.drawString("INSERT - Use for add record Km/Lt", 0, 76,
            Graphics.TOP | Graphics.LEFT);
          g.drawString("REPORT - Use for list the 10 last consumer", 0, 88,
            Graphics.TOP | Graphics.LEFT);
          g.drawString("Fernando Anselmo", 0, 112, Graphics.TOP | Graphics.LEFT);
          break;
        case 1:
          int lin = 0;
          g.drawString("Report Option", 0, lin, Graphics.TOP | Graphics.LEFT);
          lin += 14;
          g.drawString("Your 10 last consumes is...", 0, lin,
            Graphics.TOP | Graphics.LEFT);
          lin += 12;
          int od1 = 0;
          int od2 = 0;
          double gas = 0.0;
          g.setColor(0, 114, 168);
          try {
            if (rsGas.getNumRecords() > 1) {
              for (int recId = 1; recId < rsGas.getNumRecords(); recId++) {
                regGas = new String(rsGas.getRecord(recId));
                od1 = Integer.parseInt(regGas.substring(0, regGas.indexOf("|")));
                gas = Double.parseDouble(regGas.substring(
                  regGas.indexOf("|") + 1));
                regGas = new String(rsGas.getRecord(recId + 1));
                od2 = Integer.parseInt(regGas.substring(0, regGas.indexOf("|")));
                lin += 15;
                g.drawString(recId + ") " + ((od2 - od1) / gas) +
                  " Km/Lt", 0, lin, Graphics.TOP | Graphics.LEFT);
              }
            }
          } catch (RecordStoreException ex) {
            ex.printStackTrace();
          }
          break;
      }
    }
    public void commandAction(Command command, Displayable displayable) {
      if (command == exitCmd) {
        try {
          rsGas.closeRecordStore();
```

```
          pai.destroyApp(true);
          pai.notifyDestroyed();
        } catch (RecordStoreException ex) {
          ex.printStackTrace();
        } catch (MIDletStateChangeException ex) {
          ex.printStackTrace();
        }
      } else if (command == insertCmd) {
        repaint();
        form = new InsertForm("Insert Option");
        pai.display.setCurrent(form);
      } else if (command == reportCmd) {
        opc = 1;
        repaint();
      } else if (command == backCmd) {
        pai.display.setCurrent(this);
        opc = 0;
        repaint();
      } else if (command == saveCmd) {
        regGas = form.getRegGas();
        byte[] data;
        try {
          if (rsGas.getNumRecords() == 11) {
            for (int recId = 1; recId < rsGas.getNumRecords(); recId++) {
              data = rsGas.getRecord(recId + 1);
              rsGas.setRecord(recId, data, 0, data.length);
            }
            data = regGas.getBytes();
            rsGas.setRecord(11, data, 0, data.length);
          } else {
            data = regGas.getBytes();
            rsGas.addRecord(data, 0, data.length);
          }
        } catch (RecordStoreException ex) {
          ex.printStackTrace();
        }
        pai.display.setCurrent(this);
        opc = 0;
        repaint();
      }
    }
    class InsertForm extends Form {
      private TextField QTDGAS;
      private TextField ODOMETER;

      public InsertForm(String title) {
        super(title);
        addCommand(backCmd);
        addCommand(saveCmd);
        setCommandListener(MenuCanvas.this);
        ODOMETER = new TextField("Odometer (Kilometer):", "", 64,
          TextField.NUMERIC);
        QTDGAS = new TextField("Gasoline (Liter):", "", 64, TextField.DECIMAL);
        this.append(new StringItem("Inform your consumption", ""));
        this.append(ODOMETER);
        this.append(QTDGAS);
      }
      public String getRegGas() {
        return ODOMETER.getString() + "|" + QTDGAS.getString();
      }
    }
  }
```

Começaremos pelo construtor que montará a janela principal. Observe que o atributo *opc* do tipo byte possuirá o controle sobre qual visão da aplicação temos. O construtor também inicializa o objeto do tipo *RecordStore* (chamado *rsGas*). Logo em seguida o método *paint* será chamado e

mostrará a seguinte janela:

Figura 1: Janela do Menu Principal

Caso o comando seja "Insert", passaremos para o método *commandAction*. Neste momento montaremos um objeto da classe *Form* e a definiremos como principal. Montaremos com este formulário a seguinte janela:

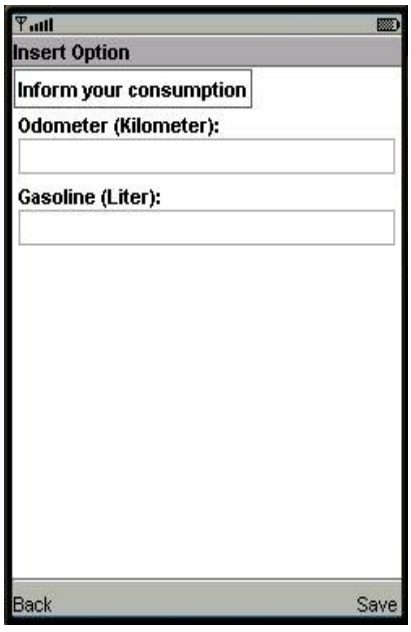

Figura 2: Janela da Opção de Inclusão

O usuário entrará com a informação da quantidade de quilômetros percorridos pelo carro até o momento e a quantidade de litros abastecidos. Pode-se escolher a opção "Save" para salvar o registro, ou "Back" para retornar ao menu principal.

Caso seja selecionada a opção "Save", realizaremos a verificação se já foram inseridos 11 registros. Em caso afirmativo, movimentaremos o registro que ocupa a segunda posição para a primeira, o que ocupa a terceira posição para a segunda e deste modo sucessivamente até que se libere a décima primeira posição, onde guardaremos este novo registro. Em caso negativo, simplesmente inserimos mais um registro ao *RecordStore*.

No menu principal, caso o comando seja "Report", passaremos para o método *commandAction*,

mudando o atributo *opc* e montamos agora a seguinte janela:

Figura 3: Janela da Opção de Relatório

Agora fica por conta do método *paint* localizar os registros informados e realizar o cálculo conforme vimos, lembrando que o método *indexOf*(String) localiza uma determinada posição dentro de um objeto *String* e o método *substring*(int1, int2) localiza uma determinada posição e retorna todos os caracteres a partir desta enquanto a posição seja menor que o segundo valor.

Note que este projeto está bem compacto e ainda permite diversas melhorias, tais como:

- Imagens
- Opções para o usuário selecionar quantos registro deseja armazenar
- Um gráfico para mostrar o consumo ao invés de informação textual

Então, utilize-o como aplicativo base para praticar.

7. Exercícios

7.1. Armazenar Preferências

Criar um aplicativo *MIDlet* que possa armazenar as preferências de um programa. Este aplicativo irá armazenar as preferências em um objeto do tipo *RecordStore*. Cada registro irá conter o nome de uma variável e seu valor. Cada conjunto variável e valor será armazenado em um único registro.

Dica: Pode-se implementar os métodos:

```
public String ler(RecordStore recStore, String nome, String valorPadrao);
public void escrever(RecordStore recStore, String nome, String valor);
```

Parceiros que tornaram JEDI™ possível

Instituto CTS
Patrocinador do DFJUG.

Sun Microsystems
Fornecimento de servidor de dados para o armazenamento dos vídeo-aulas.

Java Research and Development Center da Universidade das Filipinas
Criador da Iniciativa JEDI™.

DFJUG
Detentor dos direitos do JEDI™ nos países de língua portuguesa.

Banco do Brasil
Disponibilização de seus *telecentros* para abrigar e difundir a Iniciativa JEDI™.

Politec
Suporte e apoio financeiro e logístico a todo o processo.

Borland
Apoio internacional para que possamos alcançar os outros países de língua portuguesa.

Instituto Gaudium/CNBB
Fornecimento da sua infra-estrutura de hardware de seus servidores para que os milhares de alunos possam acessar o material do curso simultaneamente.