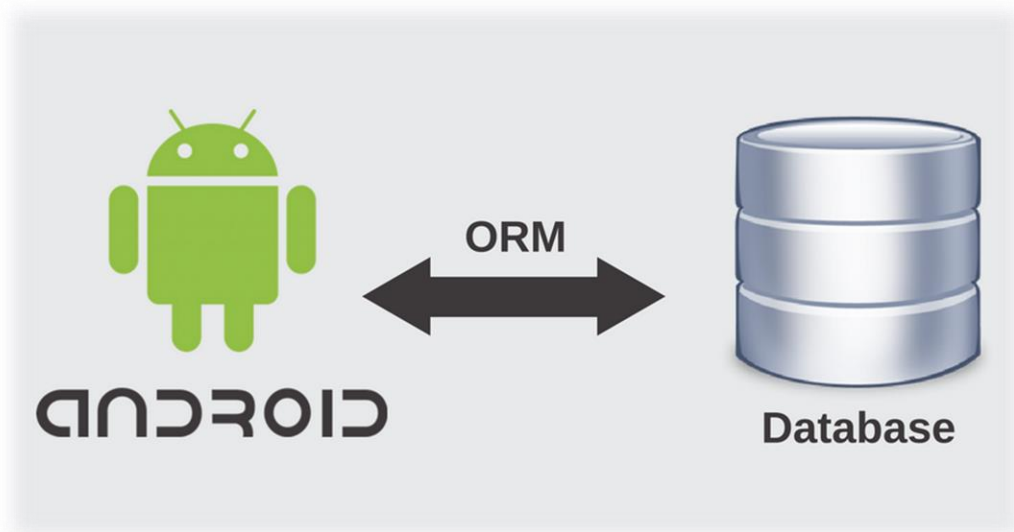


Banco de Dados Interno com **Sugar ORM**

Prof. Alexei Bueno



INTRODUÇÃO

- A ideia fundamental de banco de dados é realizarmos um processo chamado “CRUD”:
- **C** → Cadastrar, inserir alguma informação...
- **R** → Read, ler, pesquisar, mostrar, filtrar informações...
- **U** → Update, atualizar, alterar a informação...
- **D** → Delete, excluir a informação...

INTRODUÇÃO

- O sistema operacional Android tem um banco de dados nativo chamado **SQLite**



Porém iremos utilizar de
uma biblioteca chamada
Sugar ORM

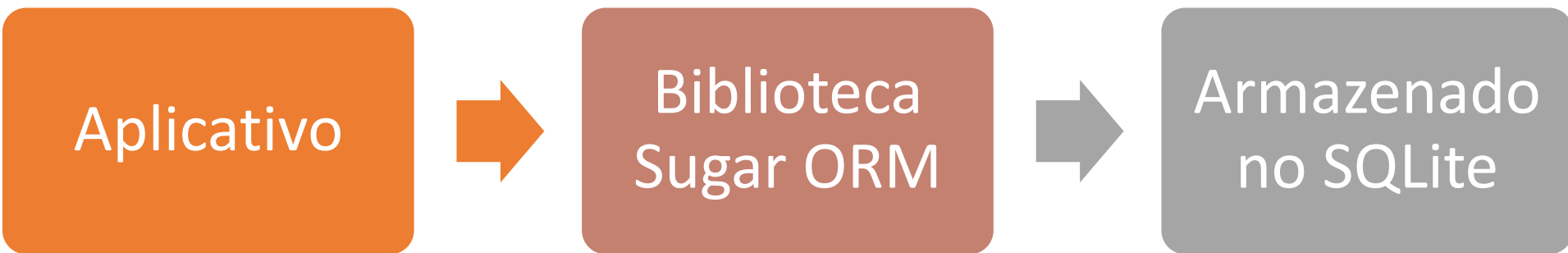


O que é **Sugar ORM**?

- **Modelagem Objeto Relacional**
- É uma biblioteca desenvolvida por programadores experientes em Java para auxiliar o trabalho com BD interno
- Lembrando que os *dados ficarão armazenados no próprio celular*, ou seja, na memória interna.
- A facilidade é que a **biblioteca** realiza a conversão de **objetos** para o modelo **relacional** dos bancos de dados (já que o **SQLite** utiliza de tabelas, mas o Java é em objetos)

VANTAGEM DE B.D. INTERNO:

- mesmo fechando o aplicativo ou mesmo desligando o celular os dados nunca poderão ser perdidos

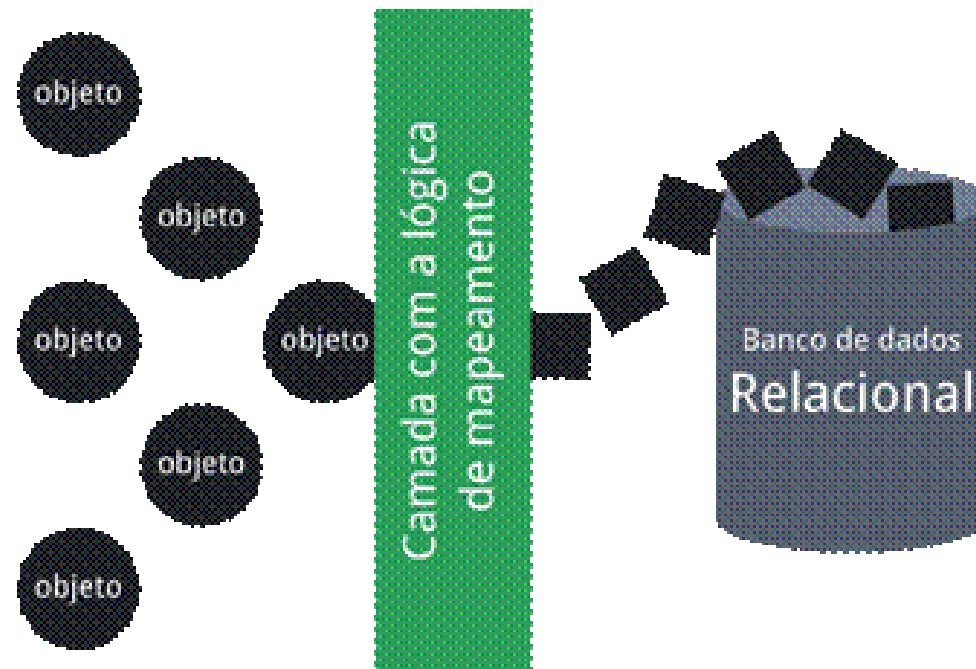


INTRODUÇÃO

- Atualmente utiliza-se de **bibliotecas** para realizar de maneira mais eficiente muitas dos processos realizado pelos sistemas informatizados...
- Iremos utilizar de uma biblioteca chamada **Sugar ORM** para trabalharmos com **banco de dados interno**
- Esta biblioteca cria um sistema de banco de dados Orientado a Objetos
- O que é muito bom já que o **Java** também é Orientado a Objetos.
- **Resumindo, nossas classes se transformarão em “tabelas”**

ORM

- Mapeamento Objeto Relacional
- Faz a comunicação dos **Objetos Java** com o Banco de Dados **Relacional**



Vantagens:

- Não precisamos escrever **comandos SQL**:
 - INSERT, UPDATE, DELETE e SELECT
- Não precisamos converter manualmente os atributos dos objetos em colunas das tabelas:
“insert into cliente (nome) values (‘ “+**objeto.getNome**+” ‘)”
- Projetos com menos linhas de códigos e conseqüentemente mais simples e rápido de se desenvolver

RESUMINDO: as classes Java são as tabela do banco de dados,
simples assim!!!

ATENÇÃO: Como é uma biblioteca que não é nativa do Android, é necessária importá-la no seu projeto...

- Incluimos no arquivo **Gradle** do Android:

```
compile 'com.github.satyan:sugar:1.4'
```

Também fazemos uma configuração no arquivo **AndroidManifest.xml**, identificando o nome do BD e registrando pelo “package name”:

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="App_Gerenciador_Livros"
    android:supportsRtl="true"
    android:theme="@style/AppTheme" >

    <meta-data android:name="DATABASE" android:value="banco.db"/>
    <meta-data android:name="DOMAIN_PACKAGE_NAME" android:value="br.com.philadelpho.app_gerenciador_livros" />

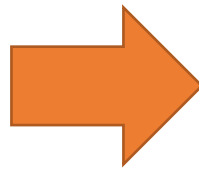
    <activity android:name=".TelaPrincipal" >
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />

            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
</application>
```

SugarORM é orientado a objetos!

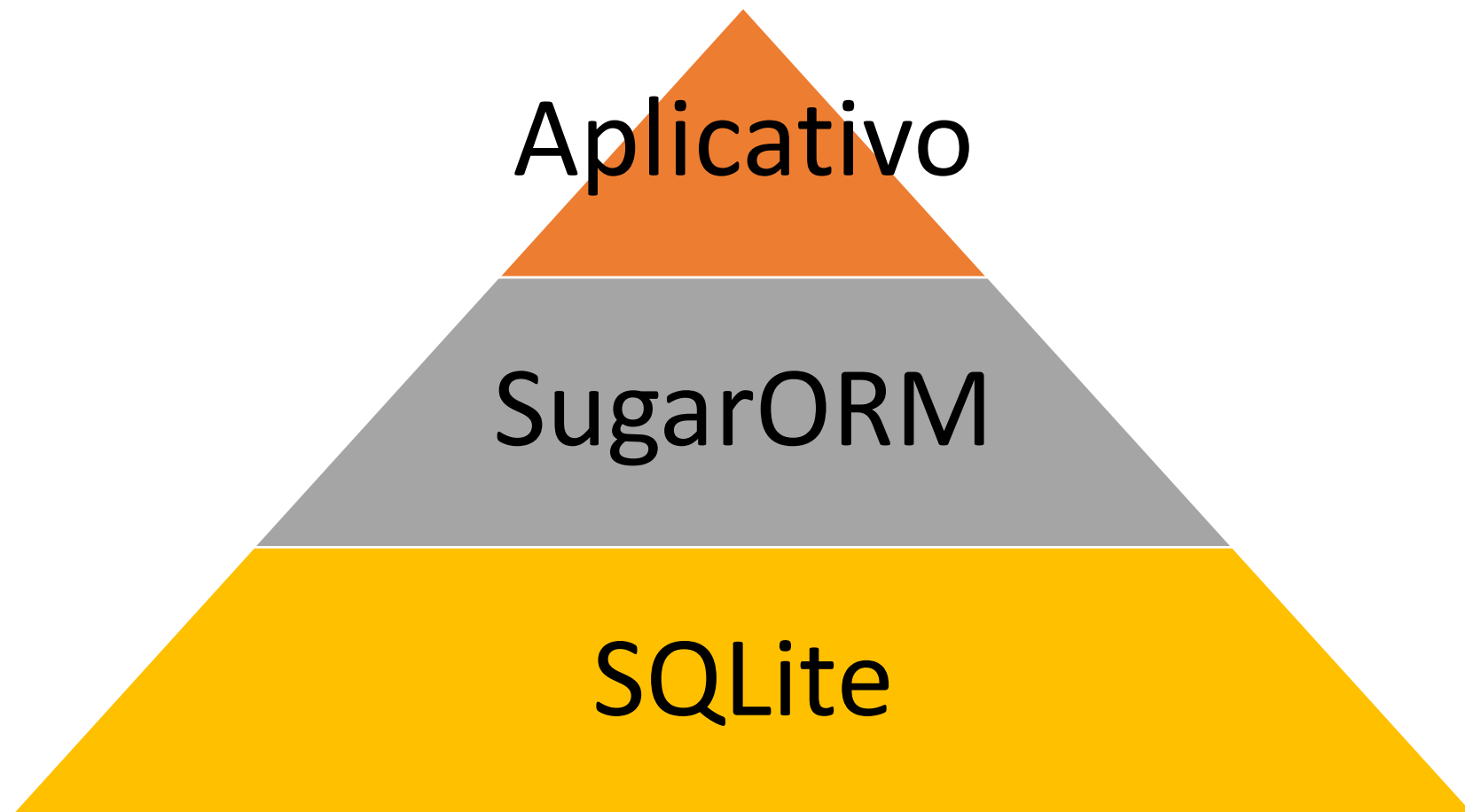
- Cada tabela do B.D. será uma classe que herdará do SugarORM:

Produto
- nome - preco - quantidade
+ incluir + atualizar + excluir



Tb_Produto			
id	nome	preco	quantidade

Por “baixo dos panos” teremos sempre o SQLite



Cada “**tabela**” do banco de dados será uma **Classe**, veja o exemplo a seguir:

```
public class Cliente extends SugarRecord {  
    //Atributo  
    Long id;  
    String nome, telefone, cidade;  
  
    //Construtor vazio  
    public Cliente(){  
    }  
  
    //Construtor com parâmetros  
    public Cliente(String nome, String telefone, String cidade)  
    {  
        this.nome = nome;  
        this.telefone = telefone;  
        this.cidade = cidade;  
    }  
  
    //Métodos get e set gerados automaticamente  
    @Override  
    public Long getId() { return id; }  
    @Override  
    public void setId(Long id) { this.id = id; }  
  
    public String getNome() { return nome; }  
    public void setNome(String nome) { this.nome = nome; }
```

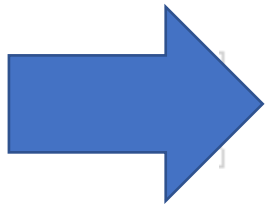
A herança **SugarRecord** diz para o Android que a classe Cliente será uma **tabela** no banco de dados:

```
public class Cliente extends SugarRecord {  
    //Atributo  
    Long id;  
    String nome, telefone, cidade;
```

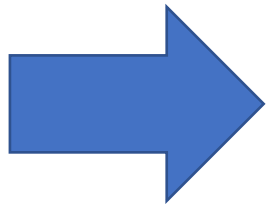
Obs.: pressionar Alt + Enter para importar as classes da biblioteca...

Toda classe terá um **construtor vazio** (que é um método com o mesmo nome da classe e mas vazio) e isto é obrigatório:

```
public class Cliente extends SugarRecord {  
    //Atributo  
    Long id;  
    String nome, telefone, cidade;  
  
    //Construtor vazio  
    public Cliente() {  
    }  
  
    //Construtor com parâmetros  
    public Cliente(String nome, String telefone, String cidade)  
    {  
        this.nome = nome;  
        this.telefone = telefone;  
        this.cidade = cidade;  
    }  
  
    //Métodos get e set gerados automaticamente  
    @Override  
    public Long getId() { return id; }
```



O **construtor com parâmetro** (não é obrigatório) mas ajuda muito na hora de criar o objeto que será armazenado:



```
//Construtor com parâmetros  
public Cliente(String nome, String telefone, String cidade)  
{  
    this.nome = nome;  
    this.telefone = telefone;  
    this.cidade = cidade;  
}
```

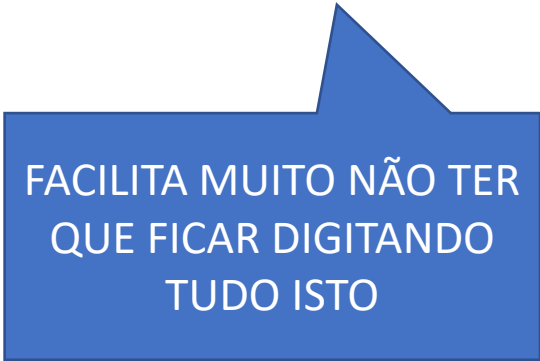
Cadastrando um cliente:

```
new Cliente("Maria dos Santos", "99454-5454", "São José do Rio preto").save();
```

Os métodos get/set o Android Studio gera automaticamente quando clicamos com o direito, generate, get and set...

```
//Métodos get e set gerados automaticamente
@Override
1 public Long getId() { return id; }
@Override
1 public void setId(Long id) { this.id = id; }

1 public String getNome() { return nome; }
1 public void setNome(String nome) { this.nome = nome; }
```



FACILITA MUITO NÃO TER
QUE FICAR DIGITANDO
TUDO ISTO

Get e Set são métodos assessores muito utilizados em java:

```
//Métodos get e set gerados automaticamente
@Override
1 public Long getId() { return id; }
@Override
1 public void setId(Long id) { this.id = id; }

1 public String getNome() { return nome; }
1 public void setNome(String nome) { this.nome = nome; }
```

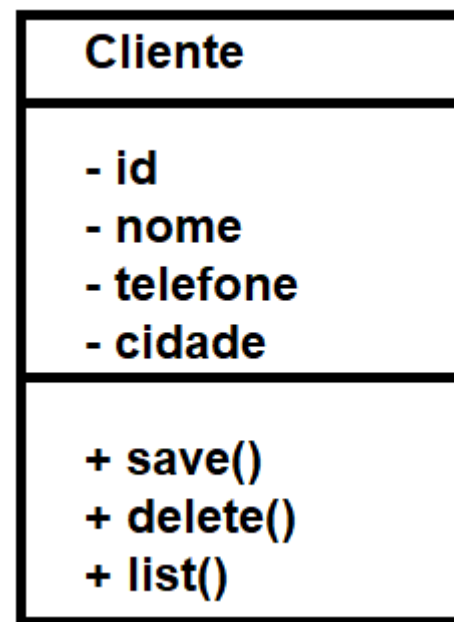
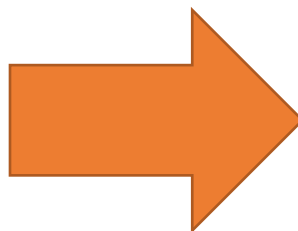
getNome() → obtenho o nome do cliente

setNome("José") → estou colocando um nome para o cliente

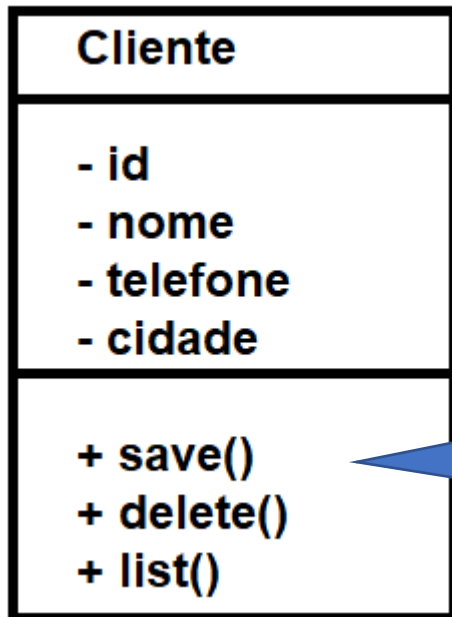
ISTO É ORIENTAÇÃO A OBJETOS

Por fim criamos uma classe que é uma tabela no banco de dados:

```
public class Cliente extends SugarRecord {  
    //Atributo  
    Long id;  
    String nome, telefone, cidade;  
  
    //Construtor vazio  
    public Cliente() {  
    }  
  
    //Construtor com parâmetros  
    public Cliente(String nome, String telefone, String cidade)  
    {  
        this.nome = nome;  
        this.telefone = telefone;  
        this.cidade = cidade;  
    }  
  
    //Métodos get e set gerados automaticamente  
    @Override  
    public Long getId() { return id; }
```



Uma facilidade do Sugar ORM é que o insert, update, delete e select é criado automaticamente:



COMO HERDAMOS DO SUGAR
ORM OS CÓDIGOS ABAIXO SÃO
GERADOS AUTOMATICAMENTE:

Save → insert e update
Delete → delete from tb...
List → select * from ...

Temos que “conectar” o aplicativo no BD interno, utilizaremos de um comando da biblioteca **Sugar ORM** e normalmente fazemos isto no método **onCreate** da activity:

```
SugarContext.init(this) ;
```

Cadastrando (gravando no BD)

“INSERT”, passo-a-passo:

```
public void cadastrar_clique(View v)
{
    //Pego os dados
    String nome = edt_nome.getText().toString();
    String telefone = edt_telefone.getText().toString();
    String cidade = edt_cidade.getText().toString();
    //Crio objeto
    Cliente cli = new Cliente(nome, telefone, cidade);
    //Gravo
    cli.save();
    //Limpo campos e mostro mensagem
    edt_nome.setText("");
    edt_telefone.setText("");
    edt_cidade.setText("");
    Toast.makeText(this, "INCLUÍDO", Toast.LENGTH_SHORT).show();
}
```

OBSERVAÇÃO

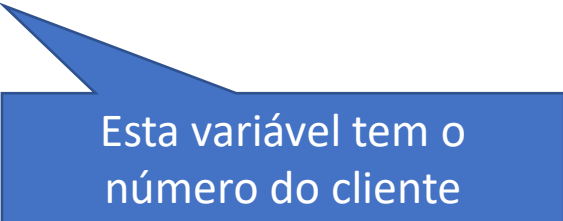
- É possível instanciar um objeto e executarmos o método para gravar em um único comando:

```
new Cliente (nome, telefone, cidade).save();
```


Alterando um cadastro “UPDATE”:

```
//Recuperando o cliente escolhido
cli = Cliente.findById(Cliente.class, id);

public void alterar_clique(View v)
{
    //Coloco os dados
    cli.setNome(edt_nome.getText().toString());
    cli.setTelefone(edt_telefone.getText().toString());
    cli.setCidade(edt_cidade.getText().toString());
    //Atualiza
    cli.save();
    //Volta para a tela de listagem
    finish();
}
```



Esta variável tem o
número do cliente

Excluindo uma cadastro “DELETE”:

```
//Recuperando o cliente escolhido  
cli = Cliente.findById(Cliente.class, id);
```

Você tem que saber qual é o cliente que vai excluir

```
public void excluir_clique(View v)  
{  
    //Exclui o cliente selecionado  
    cli.delete();  
    //Volta para a tela de listagem  
    finish();  
}
```

Como fazer “*SELECT * FROM CLIENTE*”:

```
//Lista os clientes no banco de dados
List<Cliente> clientes = Cliente.listAll(Cliente.class);

//Adapto
ArrayAdapter<Cliente> adaptador = new ArrayAdapter<>( this,
    android.R.layout.simple_list_item_1,
    clientes);

//Coloca no listview
lst_cliente.setAdapter(adaptador);
```

Agora irei mostrar alguns
comandos de banco de
dados para consulta, que
são select mais
elaborados...

Realizando o comando de banco de dados "LIKE":

```
List<Cliente> clientes =  
Cliente.find(Cliente.class,  
"nome LIKE ?", texto_pesquisa + "%");
```

ALEX encontrará:

ALEX
ALEXEI
ALEXANDRE
ALEXANDRA

Obtendo a data do pedido cujo campo
id_pedido seja 10

```
List<Pedido> pedidos = Pedido.find(Pedido.class,  
"id_pedido = ?", "10");
```

```
if(!pedidos.isEmpty()) {
```



É um WHERE

```
String data = pedidos.get(0).getData ();
```

```
}
```

Você pode localizar o pedido 10 e alterar a data:

```
List<Pedido> pedidos = Pedido.find(Pedido.class,  
"id_pedido = ?", "10");
```

```
if(!pedidos.isEmpty()) {
```

```
    pedidos.get(0).setData ("2019-11-01");
```

```
    pedidos.get(0).save();
```

```
}
```

De maneira semelhante você pode localizar o pedido 10 e EXCLUIR:

```
List<Pedido> pedidos = Pedido.find(Pedido.class,  
"id_pedido = ?", "10");
```

```
if(!pedidos.isEmpty()) {  
    pedidos.get(0).delete();  
}
```


Procedimento de “login”:

```
public void entrar_clique(View v)
{
    //Dados que o usuário digitou
    String nome = edt_usuario.getText().toString();
    String senha = edt_senha.getText().toString();

    //Consulta
    List<Usuario> usuario = Select.from(Usuario.class)
        .where(Condition.prop("nome").eq(nome),
            Condition.prop("senha").eq(senha))
        .list();

    //Se encontrou entra na tela inicial
    if (usuario.size() > 0){
        Intent tela = new Intent(this, TelaInicial.class);
        startActivity(tela);
    }else{
        Toast.makeText(this, "Usuário ou senha inválidos", Toast.LENGTH_SHORT).show();
    }
}
```

Função COUNT para contar quantos pedidos tenho cadastrados:

```
long qtd = Pedido.count(Pedido.class, null, null);
```

Para excluir todos os registros de uma tabela, a exemplo de **DELETE FROM PEDIDO**, faça:

```
Pedido.deleteAll(Pedido.class);
```


Podemos utilizar os
comandos SQL também...

No exemplo a seguir realizo um select com ORDER BY DESC utilizando do comando convencional de consultas SQL:

```
List<Tarefa> tarefas =  
    Tarefa.findWithQuery(Tarefa.class, "SELECT * FROM Tarefa ORDER BY id DESC");
```

Query Builder

- Em alguns casos precisamos criar uma query mais elaborada e concatenar vários valores pode ficar complicado, então podemos utilizar um *builder*, com os métodos *from* para selecionar a tabela, o método *where* para o filtro, *orderBy* para ordenação, *limit* para limitar o número de resultados retornados, e finalmente o método *list* para efetivamente retornar a coleção *List* com os dados retornados pela consulta.

Uma pesquisa mais elaborada:

```
List<Pessoa> pessoas = Select.from(Pessoa.class)
    .where(
        Condition.prop("nome").like("%ro%"),
        Condition.prop("altura").gt(1.50)
    )
    .orderBy("nome")
    .limit("10")
    .list();
```


Observações sobre Query Builder:

- ***like(Object)***: LIKE;
- ***gt(Object)***: Maior que;
- ***isNull(Object)***: é nulo;
- ***isNotNull(Object)***: não é nulo;
- ***notEq(Object)***: diferente
- ***eq(Object)***: Igual;
- ***lt(Object)***: Menor que;

Sem ORM:

```
string sql = "SELECT * FROM PESSOAS";
SqlCommand command = new SqlCommand(sql, connection);

DataReader reader = command.Execute();
List<Pessoas> pessoas = new List<Pessoas>();

while(row = reader.Next())
{
    pessoas.Add(new Pessoa
    {
        Id = Convert.ToInt(row["Id"]),
        Nome = row["Nome"].ToString(),
        DataNascimento = Convert.ToDateTime(row["DataNascimento"]);
    });
}
```

Com ORM:

```
List<Pessoas> pessoas = database.Pessoas.SelectAll().ToList();
```

Para realizar as funções COUNT, SUM e AVG o melhor é realizar um select convencional, tal como o exemplo de SUM abaixo:

```
SugarDb banco = new SugarDb(this);
```

```
SQLiteDatabase database = banco.getDB();
```

```
SQLiteStatement query = database.compileStatement("SELECT SUM(quantidade) FROM Pedido");
```

```
long total = query.simpleQueryForLong();
```

Minha videoaula sobre este
assunto
(assista para reforçar os
conceitos):

<https://youtu.be/Y56BDPJtXkk>

Nosso primeiro aplicativo

APP_TAREFAS



Tarefa
id : Long tarefa : String
save() delete() list()

Obs.: configuração no Android Studio 2 para funcionar Sugar ORM:

