API

Agenda

- JDBC
- File I/O

JDBC

Criado em 1996 como uma forma de acessar dados em Banco de Dados relacionais a partir de programas Java.

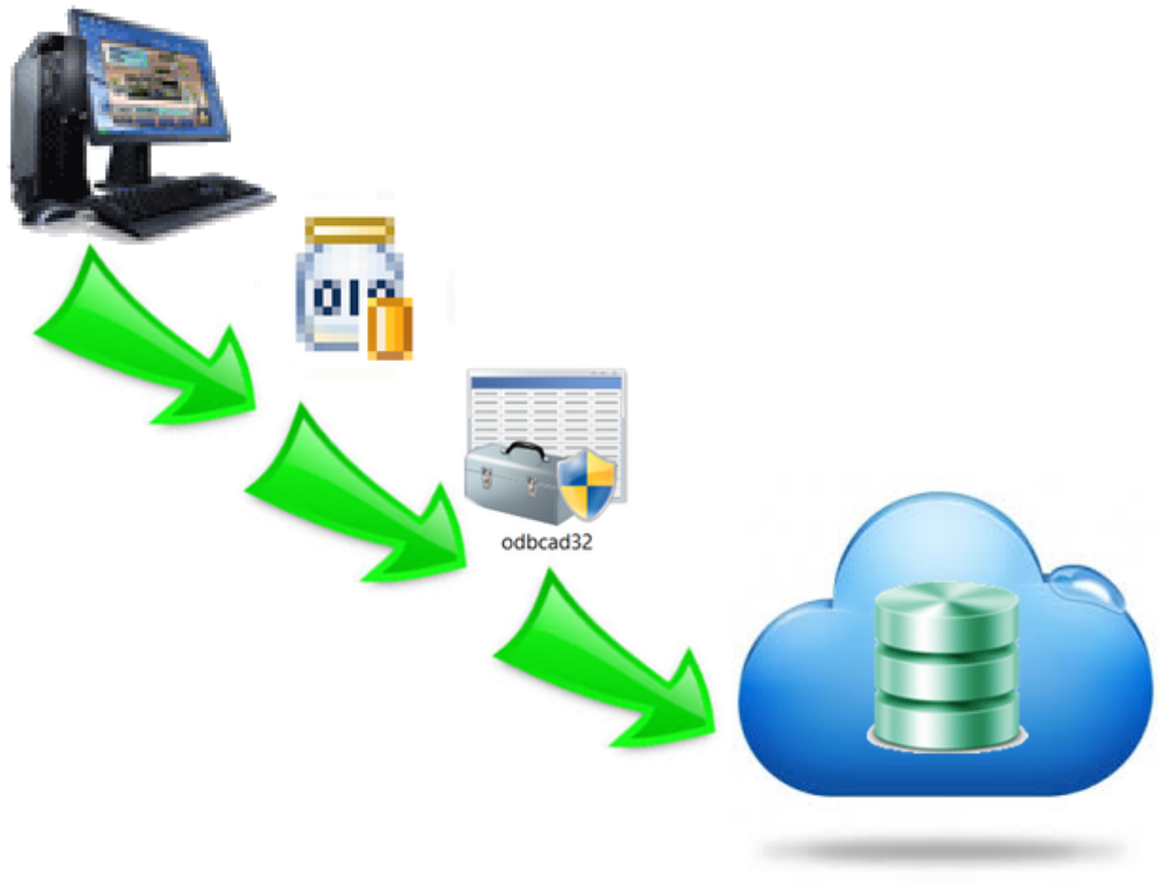
São duas API's, uma para desenvolvedores de drivers JDBC e outra para desenvolvedores de aplicações.

JDBC

O Drivers são classificados por tipos e são quatro.

O tipo 1 transforma JDBC em ODBC e necessita de um Driver ODBC (Windows) para se comunicar com o Banco de Dados.

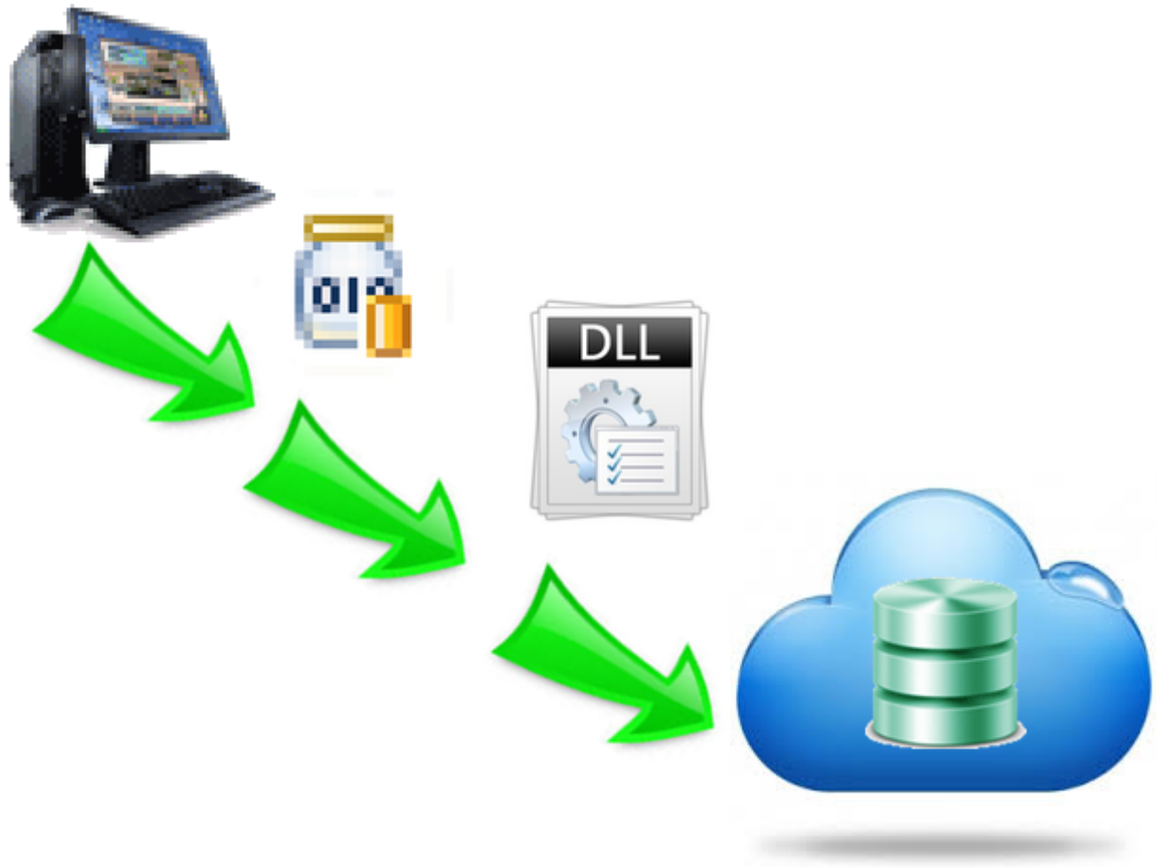
A versão que já vinha incluído no JDK foi retirada no Java 8, por ser considerado como versão de migração e não ser estável.



JDBC

O driver tipo 2 é um driver parcialmente escrito na linguagem de programação Java e parcialmente em código nativo (Ex.: Linguagem C).

Com este tipo de driver quem acessa o banco de dados é a biblioteca construída em código nativo.



JDBC

O driver tipo 3 é um driver escrito na linguagem de programação Java que conecta em um proxy de rede e este é que conecta ao banco de dados.

JDBC

O driver tipo 4 é um driver escrito na linguagem de programação Java que conecta diretamente ao banco de dados, sem a necessidade de conexões ou proxy intermediários.

JDBC

A programação com as classes JDP não é, conceitualmente, muito diferente da programação com as outras classes da plataforma Java.

Ao se conectar com um banco de dados, dever ser especificado a fonte de dados e parâmetros opcionais.

```
// Registra o Driver JDBC
Class.forName("com.mysql.jdbc.Driver");

// Estabelece a conexão ao Banco de Dados
Connection con = DriverManager.getConnection(
    "jdbc:mysql://localhost/empresa",
    "root", "senai");
```

JDBC

Para acessar informações no banco de dados, é necessário a preparação da instrução SQL através da classe **PreparedStatement**.

Assim é possível descrever qual a ação, qual tabela, quais colunas e demarcar os argumentos.

```
// Inserindo um registro numa Tabela
PreparedStatement sql = con.prepareStatement(
  "insert into contrato (numero, descricao) values (?, ?)");

sql.setInt(1, 5678);
sql.setString(2, "Dados do registro!");
sql.execute();
```

JDBC

É possível a execução de múltiplos comandos SQL em lote através do uso do método **executeBatch()** de um objeto **PreparedStatement**.

```
// Execução em Lote
sql.setInt(1, 5678);
sql.setString(2, "Dados do registro!");
sql.addBatch();

sql.setInt(1, 98324);
sql.setString(2, "Teste novo");
sql.addBatch();

sql.executeBatch();
```

JDBC

Podemos controlar a forma como são efetuados os **commits** das ações exercidas nas tabelas de banco de dados pela utilização dos métodos **setAutoCommit**, **commit** e **rollback** de um objeto **Connection**.

```
// Desativa auto gravação dos registros
con.setAutoCommit(false);

try {
  sql.setInt(1, 5678);
  sql.setString(2, "Dados do registro!");
  sql.execute();
  // efetiva a gravação
  con.commit();
} catch( SQLException ex) { // erro
  // desfaz a ação
  con.rollback();
}
```

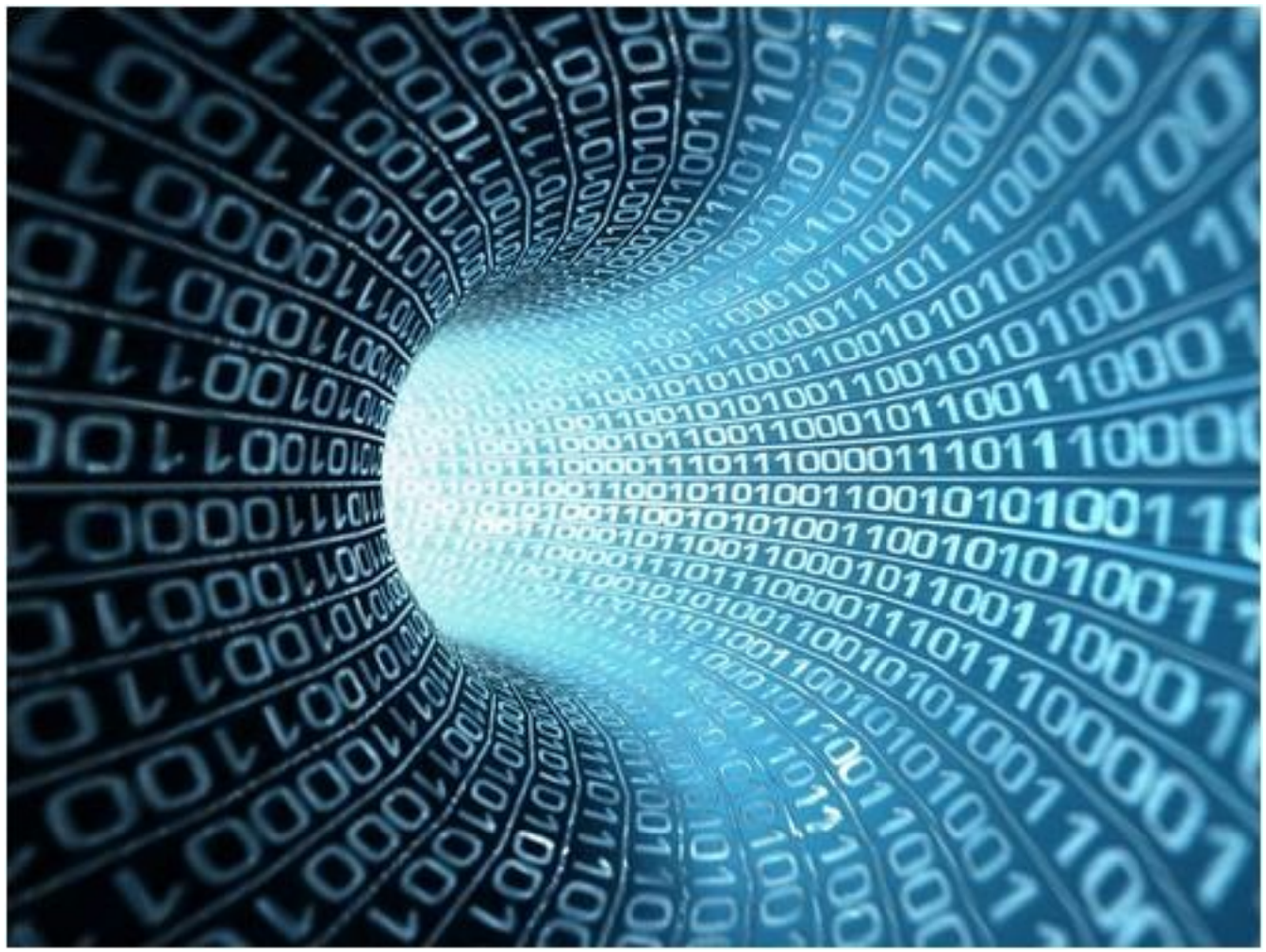
JDBC

A correspondência dos tipos Java com os tipos JDBC estão na tabela abaixo:

Tipo de dados SQL	*Tipo de dados Java*
INTEGER ou INT	int
SMALLINT	short
NUMERIC(m, n), DECIMAL(m, n) ou DEC(m, n)	java.sql.Numeric
FLOAT(n)	double
REAL	float
DOUBLE	double
CHARACTER(n) ou CHAR(n)	String
VARCHAR(n)	String
BOOLEAN	boolean
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
BLOB	java.sql.Blob
CLOB	java.sql.Clob
ARRAY	java.sql.Array

File I/O

Antes de falar a respeito de como efetuar gravações de arquivos em Java é necessário definir **Streams**.



Streams são uma abstração de baixo nível para a comunicação de dados em Java.

Um **Stream** representa um ponto num canal de comunicação.

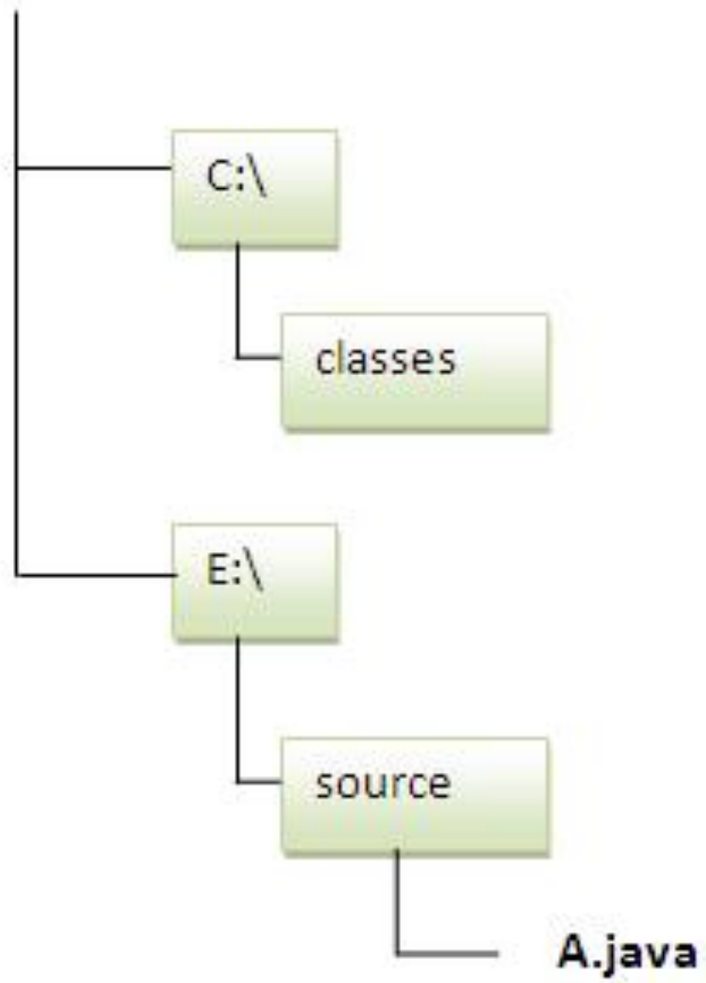
File I/O

Os **Streams** são uma fila do tipo ***FIFO*.**

Isto significa que os primeiros bytes escritos num **OutputStream** serão os primeiros a serem lidos de um **InputStream**.

File I/O

A classe **File** representa o nome de um arquivo independente da arquitetura do Sistema Operacional.



File I/O

A class **File** oferece vários métodos para determinar informações sobre arquivos e diretórios, bem como permitindo a modificação de seus atributos.

Método	Descrição
canRead()	Retorna true se o arquivo pode ser lido
canWrite()	Retorna true se o arquivo pode ser gravado
delete()	Retorna true se obtiver sucesso na deleção do arquivo
exists()	Retorna true se o arquivo existir
getName()	Retorna o nome do arquivo sem o diretório
getPath()	Retorna o nome do arquivo com a parte do diretório
getDirectory()	Retorna o nome do diretório onde o arquivo reside
isFile()	Retorna true se for um arquivo
isDirectory()	Retorna true se for um diretório
length()	Retorna o tamanho do arquivo
list()	Retorna um array de Strings contendo os nomes dos arquivos e sub-diretórios contidos num diretório
mkdir()	Retorna true se conseguir criar o diretório
renameTo()	Retorna true se conseguir renomear o arquivo

File I/O

As classes **FileOutputStream**, **FileWriter** e **PrintWriter** são utilizadas para a gravação de dados em arquivos.

```
try {
        // Declara um objeto do tipo File
        File fl = new File("c:/User/Fulano/Desktop/meuDoc.txt");

        // Abre o arquivo para gravação
        FileOutputStream fo = new FileOutputStream(fl);

        // Grava um texto no arquivo
        String txt = "Texto de exemplo 1\n";
        // Transforma a String em array de bytes
        fo.write(txt.getBytes());

        // Fecha o arquivo
        fo.close();

        // Abre o arquivo para acrecentar mais textos
        FileWriter fw = new FileWriter(fl, true);

        // Grava um texto
        fw.write("Texto de Exemplo 2\n");

        // Cria um PrintWriter a partir do FileWriter
        PrintWriter pw = new PrintWriter(fw);

        // Grava um novo texto
        pw.println("Texto de Exemplo 3");

        // Fecha o arquivo
        fw.close();

} catch(FileNotFoundException ex) {
        ex.printStackTrace();
} catch(IOException ex) {
        ex.printStackTrace();
}
```

File I/O

As classes **FileInputStream**, **FileReader** e **BufferReader** são utilizadas para a leitura de dados em arquivos.

```
try {
        // Declara um objeto do tipo File
        File fl = new File("c:/User/Fulano/Desktop/minhaImagem.dat");

        // Abre o arquivo para leitura
        FileInputStream fi = new FileInputStream(fl);

        // Reserva a área de leitura
        byte[] buffer = new byte[1024];
        // Lê 1024 bytes do arquivo
        fi.read(buffer);
        // Fecha o arquivo
        fi.close();

        // Abre o arquivo para leitura
        FileReader fr = new FileReader("C:/meusDados/lista.txt");

        // Reserva a área de leitura
        char[] texto = new char[1024];
        // Lê uma sequencia de caracteres do arquivo
        fr.read(texto);

        // Cria um PrintWriter a partir do FileWriter
        BufferedReader br = new BufferedReader(fr);

        // Lê todas as linhas restante do arquivo
        String linha = br.readLine();

        while(linha != null) {
                System.out.println(linha);
                linha = br.readLine();
        }

        // Fecha o arquivo
        fr.close();
} catch(FileNotFoundException ex) {
        ex.printStackTrace();
} catch(IOException ex) {
        ex.printStackTrace();
}
```

File I/O

As classe **ObjectOutputStream** é utilizada para a gravação de objetos em arquivos.

Um objeto para que possa ser *serializado* (gravado) deve implementar a interface **Serializable**.

```
try {
        // Cria um objeto para ser gravado
        List<String> lista = new ArrayList<String>();

        // Adiciona objetos na lista
        lista.add("Texto");
        lista.add("java");
        lista.add("Novo");

        // Cria um arquivo para gravação
        FileOutputStream fo = new FileOutputStream("meusObjetos.dat");

        // Cria um ObjectOutputStream a partir do FileOutputStream
        ObjectOutputStream objOut = new ObjectOutputStream(fo);

        // Grava o objeto
        objOut.writeObject(lista);

        // fecha o arquivo
        objOut.close();
} catch(FileNotFoundException ex) {
        ex.printStackTrace();
} catch(IOException ex) {
        ex.printStackTrace();
}
```

File I/O

As classe **ObjectInputStream** é utilizada para a leitura de objetos em arquivos.

```
try {

        // Abre um arquivo para leirura
        FileInputStream fi = new FileInputStream("meusObjetos.dat");

        // Cria um ObjectInputStream a partir do FileInputStream
        ObjectInputStream objIn = new ObjectInputStream(fi);

        // Lê o objeto
        List<String> lista = (List<String>)objIn.readObject();

        // fecha o arquivo
        objIn.close();

        // Exibe os objetos lidos
        for(String txt : lista) {
                System.out.println(txt);
        }

} catch(ClassNotFoundException ex) {
        ex.printStackTrace();
} catch(FileNotFoundException ex) {
        ex.printStackTrace();
} catch(IOException ex) {
        ex.printStackTrace();
}
```

Referências

- Programando em Java2 - Teoria & Aplicações
  Rui Rossi dos Santos - Axcel Books - 2004

- Core Java2 - Volume I - Fundamentos
  Cay S. Horstmann & Gary Cornell - The Sun Microsystems Press - Série Java - 2003

- Java Programming
  Nick Clements, Patrice Daux & Gary Williams - Oracle Corporation - 2000