Technologie de l'information

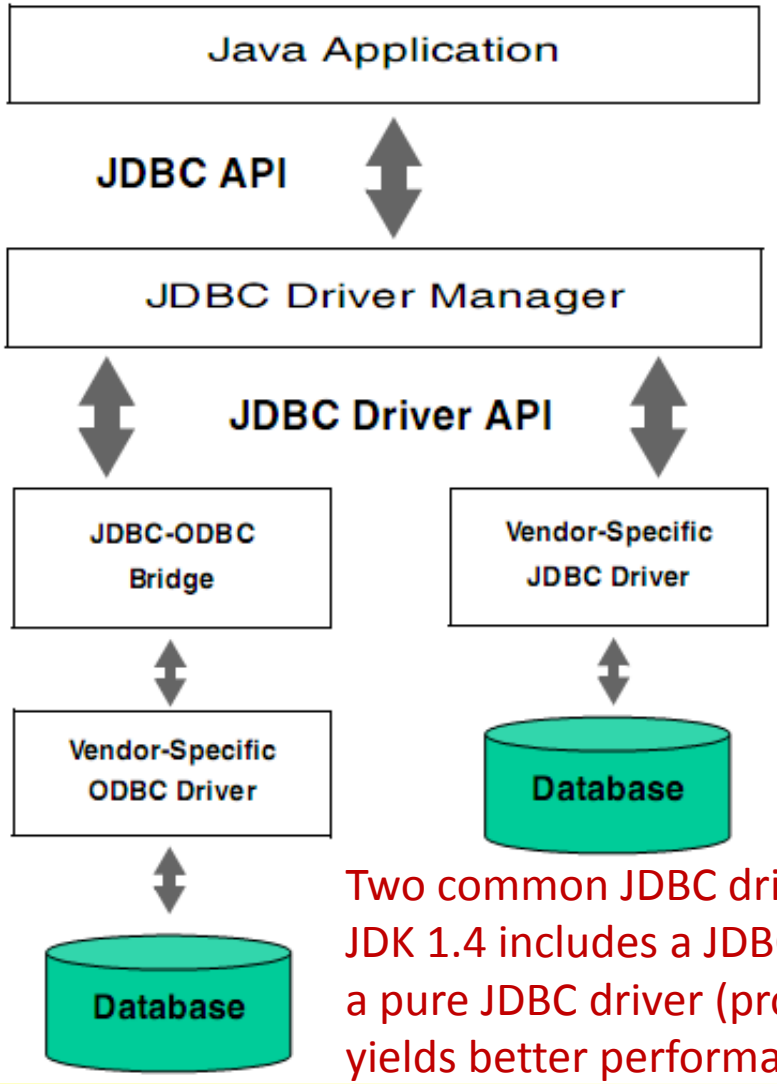
Data Base and JDBC (Java DataBase Connectivity)

PhD.Ing. Rodrigue Carlos Nana Mbinkeu

Member of DBgroup, University of Modena, Italy
www.dbgroup.unimo.it

Personnal's page: http://www.dbgroup.unimo.it/~nana/

JDBC provides a standard library for accessing relational databases.



Two common JDBC driver implementations. JDK 1.4 includes a JDBC-ODBC bridge; however, a pure JDBC driver (provided by the vendor) yields better performance.

Using JDBC in General

By using the JDBC API, you can access a wide variety of SQL databases with exactly the same Java syntax. It is important to note that although the JDBC API standardizes the approach for connecting to databases, the syntax for sending queries and committing transactions, and the data structure representing the result, JDBC does not attempt to standardize the SQL syntax. So, you can use any SQL extensions your database vendor supports. In this section we present **the seven standard steps for querying databases**.

1. **Load the JDBC driver.** To load a driver, you specify the classname of the database driver in the Class.forName method. By doing so, you automatically create a driver instance and register it with the JDBC driver manager.

2. **Define the connection URL.** In JDBC, a connection URL specifies the server host, port, and database name with which to establish a connection**.**

3. **Establish the connection.** With the connection URL, username, and password, a network connection to the database can be established. Once the connection is established, database queries can be performed until the connection is closed.

4. **Create a Statement object.** Creating a Statement object enables you to send queries and commands to the database.

5. **Execute a query or update.** Given a Statement object, you can send SQL statements to the database by using the execute, executeQuery, executeUpdate, or executeBatch methods.

Using JDBC in General(2)

6. **Process the results.** When a database query is executed, a ResultSet is returned. The ResultSet represents a set of rows and columns that you can process by calls to next and various getXxx methods.

7. **Close the connection.** When you are finished performing queries and processing results, you should close the connection, releasing resources to the database.

Load the JDBC Driver

The driver is the piece of software that knows how to talk to the actual database server. To load the driver, you just load the appropriate class; a static block in the driver class itself automatically makes a driver instance and registers it with the JDBC driver manager.

```
try {

Class.forName("connect.microsoft.MicrosoftDriver");
Class.forName("oracle.jdbc.driver.OracleDriver");
Class.forName("com.sybase.jdbc.SybDriver");

} catch(ClassNotFoundException cnfe) {

System.err.println("Error loading driver: " + cnfe);

}
```

Using JDBC in General(3)

In principle, you can use **Class.forName** for any class in your **CLASSPATH.** In practice, however, most JDBC driver vendors distribute their drivers inside JAR files. So, during development be sure to include the path to the driver JAR file in your CLASSPATH setting. For deployment on a Web server, put the JAR file in the **WEB-INF/lib** directory of your Web application. Check with your Web server administrator, though. Often, if multiple Web applications are using the same database drivers, the administrator will place the JAR file in a common directory used by the server. For example, in Apache Tomcat, JAR files common to multiple applications can be placed in **install_dir/common/lib**.

Define the Connection URL

Once you have loaded the JDBC driver, you must specify the location of the database server. URLs referring to databases use the jdbc: protocol and embed the server host, port, and database name (or reference) within the URL. **The exact format is defined in the documentation that comes with the particular driver,** but here are a few representative examples.

```
String host = "dbhost.yourcompany.com";
String dbName = "someName";
int port = 1234;

String oracleURL = "jdbc:oracle:thin:@" + host + ":" + port + ":" + dbName;
String sybaseURL = "jdbc:sybase:Tds:" + host + ":" + port + ":" + "?SERVICENAME=" +
dbName;

String msAccessURL = "jdbc:odbc:" + dbName;
```

Establish the Connection

To make the actual network connection, pass the URL, database username, and database password to the **getConnection method of the DriverManager class**, as illustrated in the following example. Note that getConnection throws an **SQLException**, so you need to use a try/catch block. We're omitting this block from the following example since the methods in the following steps throw the same exception, and thus you typically use a single try/catch block for all of them.

```
String username = "jay_debesee";
String password = "secret";
Connection connection = DriverManager.getConnection(oracleURL,username, password);
```

Using JDBC in General(5)

The **Connection** class includes other useful methods, which we briefly describe below.

- **prepareStatement**. Creates precompiled queries for submission to the database.
- **prepareCall**. Accesses stored procedures in the database.
- **rollback/commit.** Controls transaction management.
- **close**. Terminates the open connection.
- **isClosed**. Determines whether the connection timed out or was explicitly closed.

An optional part of establishing the connection is to look up information about the database with the getMetaData method. This method returns a DatabaseMetaData object that has methods with which you can discover the name and version of the database itself (getDatabaseProductName, getDatabaseProductVersion) or of the JDBC driver (getDriverName, getDriverVersion). Here is an example.

```
DatabaseMetaData dbMetaData = connection.getMetaData();
String productName = dbMetaData.getDatabaseProductName();
System.out.println("Database: " + productName);
String productVersion = dbMetaData.getDatabaseProductVersion();
System.out.println("Version: " + productVersion);
```

Create a Statement Object

A Statement object is used to send queries and commands to the database. It is created from the Connection using createStatement as follows.

```
Statement statement = connection.createStatement();
```

Most, but not all, database drivers permit multiple concurrent Statement objects to be open on the same connection.

Execute a Query or Update

Once you have a Statement object, you can use it to send SQL queries by using the executeQuery method, which returns an object of type ResultSet. Here is an example.

```
String query = "SELECT col1, col2, col3 FROM sometable";
ResultSet resultSet = statement.executeQuery(query);
```

Using JDBC in General(7)

The following list summarizes commonly used methods in the **Statement** class.

• **executeQuery**. Executes an SQL query and returns the data in a **ResultSet**. The ResultSet may be empty, but never null.

• **executeUpdate**. Used for UPDATE, INSERT, or DELETE commands. Returns the number of rows affected, which could be zero. Also provides support for Data Definition Language (DDL) commands, for example, CREATE TABLE, DROP TABLE, and ALTER TABLE.

• **executeBatch**. Executes a group of commands as a unit, returning an array with the update counts for each command. Use addBatch to add a command to the batch group. Note that vendors are not required to implement this method in their driver to be JDBC compliant.

• **setQueryTimeout**. Specifies the amount of time a driver waits for the result before throwing an SQLException.

• **getMaxRows/setMaxRows**. Determines the number of rows a ResultSet may contain. Excess rows are silently dropped. The default is zero for no limit.

Process the Results

The simplest way to handle the results is to use the next method of ResultSet to move through the table a row at a time. Within a row, ResultSet provides various **getXxx methods** that take a column name or column index as an argument and return the result in a variety of different Java types. For instance, use getInt if the value should be an integer, getString for a String, and so on for most other data types. If you just want to display the results, you can use getString for most of the column types. However, if you use the version of getXxx that takes a column index (rather than a column name), **note that columns are indexed starting at 1** (following the SQL convention), not at 0 as with arrays, vectors, and most other data structures in the Java programming language.

Here is an example that prints the values of the first two columns and the first name and last name, for all rows of a ResultSet.

```
while(resultSet.next()) {

  System.out.println(resultSet.getString(1) + " " +
                     resultSet.getString(2) + " " +
                     resultSet.getString("firstname") + "  "
                     resultSet.getString("lastname"));

}
```

NB: We suggest that when you access the columns of a ResultSet, you use the column name instead of the column index. That way, if the column structure of the table changes, the code interacting with the ResultSet will be less likely to fail.

Using JDBC in General(9)

The following list summarizes useful ResultSet methods.

• **next/previous.** Moves the cursor to the next (any JDBC version) or previous (JDBC version 2.0 or later) row in the ResultSet, respectively.

• **relative/absolute.** The relative method moves the cursor a relative number of rows, either positive (up) or negative (down). The absolute method moves the cursor to the given row number. If the absolute value is negative, the cursor is positioned relative to the end of the ResultSet (JDBC 2.0).

• **getXxx.** Returns the value from the column specified by the column name or column index as an Xxx Java type (see java.sql.Types). Can return 0 or null if the value is an SQL NULL.

• **wasNull.** Checks whether the last getXxx read was an SQL NULL. This check is important if the column type is a primitive (int, float, etc.) and the value in the database is 0. A zero value would be indistinguishable from a database value of NULL, which is also returned as a 0. If the column type is an object (String, Date, etc.), you can simply compare the return value to null.

• **findColumn.** Returns the index in the ResultSet corresponding to the specified column name.

- **getRow.** Returns the current row number, with the first row starting at 1 (JDBC 2.0).
- **getMetaData.** Returns a ResultSetMetaData object describing the ResultSet.

ResultSetMetaData gives the number of columns and the column names. The getMetaData method is particularly useful. Given only a ResultSet, you have to know the name, number, and type of the columns to be able to process the table properly. For most fixed-format queries, this is a reasonable expectation. For ad hoc queries, however, it is useful to be able to dynamically discover high-level information about the result. That is the role of the ResultSetMetaData class: it lets you determine the number, names, and types of the columns in the ResultSet. Useful ResultSetMetaData methods are described below.

- **getColumnCount.** Returns the number of columns in the ResultSet.
- **getColumnName**. Returns the database name of a column (indexed starting at 1).
- **getColumnType.** Returns the SQL type, to compare with entries in java.sql.Types.
- **isReadOnly.** Indicates whether the entry is a read-only value.
- **isSearchable.** Indicates whether the column can be used in a WHERE clause.
- **isNullable.** Indicates whether storing NULL is legal for the column.

ResultSetMetaData does not include information about the number of rows; however, if your driver complies with JDBC 2.0, you can call last on the ResultSet

Close the Connection

To close the connection explicitly, you would do:

```
connection.close();
```

Closing the connection also closes the corresponding Statement and ResultSet objects. You should postpone closing the connection if you expect to perform additional database operations, since the overhead of opening a connection is usually large. In fact, reusing existing connections is such an important optimization that the JDBC 2.0 API defines a ConnectionPoolDataSource interface for obtaining pooled connections.

Opérateur d'égalité

- **les valeurs numériques ne peuvent pas être comparées à des valeurs booléennes. Une telle comparaison entraine une erreur de compilation;**

- **La promotion numérique se produit avant comparaison des valeurs numériques. Ce qui permet la comparaison entre tous les types numériques.**

NB: Quand les références à des objets sont comparées les operateurs == et != vérifient si les objets A et B sont une même instance et pas s'ils ont la même valeur.

Par exemple, supposons que les objets A et B soient de la même classe et qu'ils aient exactement les mêmes valeurs de champs. Si ces objets sont distincts(situés à des emplacements de mémoire différents), alors **A == B renverra false**, et **A!=B, la valeur true.**

La comparaison de deux objets se fait grâce à la méthode 'equals()'. Cette méthode est définie dans la classe Object. Elle permet de dire si deux objets ont la même valeur.

Exemple 1:

```
class EqualString11
{
 public static void main(String args()){
  String s  = ``ab``;
  String s1 = s + ``cd``;
  String s2 = ``abcd``;
  System.out.println(s1);
  System.out.println(s2);
  System.out.println(s1==s2);
  }
}

class EqualString22
{
 public static void main(String args()){
  String s  = ``ab``;
  String s1 = s + ``cd``;
  String s2 = ``abcd``;
  System.out.println(s1);
  System.out.println(s2);
  System.out.println(s1.equals(s2));
  } }
```

Opérateur instanceof

L'operateur **instanceof** est un operateur binaire qui détermine si une référence à un objet(l'opérande de gauche) est une instance de la classe, de l'interface ou du tableau du type spécifié dans l'opérande de droite.

L'operateur instanceof renvoie la valeur booléenne **true** si l'opérande de gauche fait référence à un objet de classe C(ou du tableau de type T), de valeur non **null**, et que l'une des conditions suivantes est vérifiée(condition non exhaustive):

- **l'opérande de droite est une classe C', et C est une sous classe de C';**
- **l'opérande de droite est un interface I et C implémente I;**
- **l'opérande de droite est un tableau de type T', l'opérande est un tableau de type T, et T est une sous classe ou une sous-interface de T' ou gal à T'.**

instanceof renvoie **false** si aucune des conditions susmentionnées n'est vérifiées ou si l'opérande de gauche est null.

Opérateur instanceof

```
Import java.util.*;

class Instance
{
Public static void main(String args())
{
String s = ``abcd``;
Vector v = new Vector();
v.add(s);
Object  o =  v.elementAt(0);
System.out.println(s   instanceof  String);
System.out.println(s  instanceof   Object);
System.out.println(o  instanceof   String);
System.out.println(o  instanceof   Object);
System.out.println(o  instanceof   Vector);

}
}
```

Opérateur de projection de type

L'operateur de projection (type) est utilisé pour convertir des valeurs numériques d'un type vers un autre type ou pour modifier la référence à un objet en un type compatible.

Conversion élargie et réduites

Une conversion élargie est une conversion vers un type numérique plus important, par exemple lorsqu'une valeur byte est promue en type int. Une conversion réduite est une conversion vers un type numérique inferieur, par exemple de long en short. Dans ce cas la conversion doit se faire de façon explicite :

```
double d = 123.456
short t  = (short) d;  // c'est une conversion réduite
```

Conversion élargie et réduites

Quand la projection de type est utilisée sur les références aux objets, les règles suivantes s'appliquent:

1. Toute référence à un objet peut être projetée dans une référence à un objet de classe Object.
2. Une référence à un objet peut être projetée dans une référence à un objet de classe C' si la classe de l'objet est une sous classe de C'.

Exemple:

```
String s = ``abcd``;
String obj =  s;
String str = (String) obj;
```

Operateur ternaire

operande1 ? Operande2 : Operande3

Si la condition est vraie alors l'operande2 est évaluée au cas contraire c'est l'operande3 qui est évaluée.

Les déclarations sont essentielles à l'écriture d'un programme Java. Vous devez savoir comment déclarer les classes, les interfaces, les variables, les méthodes et les constructeurs pour pouvoir briquer le titre de programmeur Java. Si les déclarations de base vous sont sans doute déjà familières, vous tomberez malgré tout sur des points inconnus ou que vous avez déjà oubliés.

Déclaration et utilisation des variables

Une déclaration de variable sert a identifier le type associé à la variable. Elle peut aussi servir à initialiser une variable à une valeur particulière de ce type. Les variables peuvent être déclarées d'un type primitif ou du type d'un objet.

Exemple:

```
Personne p; //déclaration avec type d'un objet
int i; // déclaration avec type primitif
```

Les tableaux

Les tableaux sont des objets Java. D'un point de vue technique, les tableaux de java sont toujours à une dimension. Les tableaux a 2 dimensions sont des tableaux de tableaux.

Les dimensions d'un tableau peuvent être spécifiées à droite de son type ou à droite de son identificateur.

POO: Déclaration et Controle d'accés(2)

Par exemple, toutes les déclarations de tableau suivantes sont équivalentes:

String[] s;
String []s;
String s[];
String s [];

Pour les tableaux multidimensionnel, on a :

String []s[];
String [][]s;
String s[][];

Remarquez que les dimensions du tableau ne sont pas spécifiées. Un tableau est crée de deux manières. La plus simple consiste à utiliser un initialiseur de tableau pour le créer et initialiser ses éléments:

String[] s = { 'abc', 'def', 'ghi'}

Les tableaux multidimensionnels peuvent être initialisés en imbriquant les initialiseurs de tableaux:

int[][] i = {{1, 2, 3}, {4, 5, 6} }

La technique illustrée ci-dessus est pratique pour les tableaux de petites tailles. Pour en créer de plus grand, il faut utiliser new, suivi du type du tableau et d'un ou de plusieurs groupe de crochets. La longueur d'une dimension du tableau est indiquée par un entier entre les crochet:

String s = new String[100];
String s = new String[20][30];

POO: Déclaration et Controle d'accés(3)

Lres objets sont créés à l'aide de l'operateur **new** et du constructeur de la classe de l'objet. Par exemple, si un objet est de classe **C**, on utilise un constructeur de la forme suivante:

new C(listeArguments)

La liste des arguments doit correspondre aux paramètres effectivement déclarés pour le constructeur utilisé. Quand un objet est créé, il est habituellement (pas toujours) affecté à une variable:

C myC = new C (listeArguments);

Les champs et les méthodes de l'objet peuvent ensuite être référencés en utilisant la variable. Par exemple si C comporte un champ f1 et une méthode m1, on peut accéder à f1 par **myC.f1** et appeler m1 par **myC.m1(listeArguments)**

Dans certains cas, il n'est pas nécessaire d'enregistrer l'objet nouvellement créé avant d'y accéder. Par exemple, vous pouvez utiliser new **C().m1()** pour appeler la méthode **m1** d'un nouvel objet **C**.

Les mots clés **this** et super permettent de faire référence à l'instance de l'objet en cours. Si **this** est utilisé dans un constructeur, il fait référence à l'objet créé. Quand il est utilisé dans une méthode non statique, il fait référence à l'objet dont la méthode est utilisée. Il ne faut pas utiliser this dans une méthode statique, car ces dernières sont associées à la classe elle-même et pas à une instance de la classe. Le mot clé super indique la superclasse de l'instance de l'objet en cours.

Variables locales

Les variables locales sont déclarées et initialisées de la même manière que les variables de champ, mais elles ne peuvent utiliser que le modificateur final, qui identifie les variables accessibles à partir de classes locales intérieure.

NB: Le modificateur final indique une valeur de variable définitive, qui ne peut plus être modifiée une fois qu'elle a été attribuée.

Variable de champ

Les variables de champ déclarés avec la syntaxe suivante:

modificateur type déclarateur;

Les modificateurs valides sont des **modificateurs d'accés**: **public, protected et private**; et les **modificateurs spéciaux**: **final, static, transient et volatile**.

Déclaration et utilisation des méthodes

Les méthodes sont déclarées selon la syntaxe suivante.

```
modificateur valeurRetour nomMéthode(listeParametre) throws Clause
{
// corps de la méthode
    }
```

Une méthode peut posséder un modificateur d'accès quelconque (public, protected, private ou accès de paquetage) ou des modificateurs spéciaux (abstract, final, native, static ou synchronized).

Le type de valeur de retour d'une méthode peut être:

void: la méthode ne renvoie pas de valeur.

Type primitif: les types primitifs peuvent être utilisés comme type de retour de la méthode.

Type d'objet: une méthode peut renvoyer une référence à un objet d'une classe ou interface quelconque.

Type de tableau: Une méthode peut renvoyer une référence à un tableau Java. Le type renvoyé est spécifié comme le type du tableau.

Déclaration et utilisation des méthodes

Une liste de paramètre de méthode est une liste de déclaration de paramètres séparés par des virgules, chaque déclaration identifiant le type du paramètre et un nom de référence. Un paramètre peut être référencé n'importe ou dans la méthode.

La clause throws d'une méthode identifie tous les types d'exception vérifiées qui peuvent être lancés durant l'exécution d'une méthode. Cela comprend des exceptions lancées par d'autres méthodes appelées par la méthode. Exemple d'appel de méthode:

variable.nomMethode(listeArguments)

Les mots clés **this** et **super** peuvent être utilisés à la place d'un nom de variable.

La signature d'une méthode se compose du nom de la méthode et de l'ensemble des types de déclarations de ses paramètres. Il est illégal dans une classe de déclarer deux méthodes ayant la même signature.

Les méthodes statiques sont des méthodes qui s'appliquent à la classe dans son ensemble et pas simplement à une de ses instances. Les méthodes statiques sont appelées de la même manière que les autres. Mais comme elles ne sont pas associées à une instance de la classe, il est plus logique d'y faire référence en utilisant le nom de la classe au lieu d'un nom de variable.

Transmission des arguments

Un argument transmis à une méthode peut être une valeur primitive ou une référence à un objet.

dans le premier cas, une copie de la valeur primitive est effectuée et mise à la disposition de la méthode. La valeur d'origine n'est pas modifiée par les opérations qui peuvent affecter l'argument dans la méthode.

Exemple:

```
class PassedValue{

 public static void main(String args[]){
StaticClass.display('Exemple de résultat');
   }
}

class StaticClass{

Static void display(String s){
System.out.println('StaticClass:  ' + s);
}
}
```

Transmission des arguments(suite)

Un argument transmis à une méthode peut être une valeur primitive ou une référence à un objet.

dans le deuxième cas, une copie de la référence à l'objet est effectuée. Les modifications à la référence de l'objet se produisant durant l'exécution de la méthode n'affectent pas la référence de l'objet d'origine. Mais les modifications effectuées sur l'objet lui-même dans la méthode affectent l'objet d'origine.

Exemple:

```
class PassedReference{
public static void main(String args[]){
Vector v = new Vector();
v.add(new String('a'));
v.add(new String('b'));
System.out.println(v);
modifyReference(v);
System.out.println(v);
modifyReferenceObject(v);
System.out.println(v);
   }
// suite de la definition de la classe à la page sivante
```

Transmission des arguments(suite)

```
static void modifyReference(Vector v) {

v = new Vector();
v.add(new String('1'));
v.add(new String('2'));


}

static void modifyReferenceObject(Vector v) {

v.removeAllElements();
v = new Vector();
v.add(new String('n'));
v.add(new String('m'));

}
} // fin de la définition de la classe
```

Modificateurs d'accès

Java propose trois modificateurs d'accès(public , protected et private) et un accès par défaut (accès de paquetage) .

public permet l'accès à une classe ou à une interface en dehors de son paquetage. Il permet aussi l'accès à une variable, une méthode ou un constructeur à partir de tout emplacement d'où l'on peut accéder à sa classe.

protected permet l'accès à une variable, une méthode ou un constructeur à partir de classes ou d'interface du même paquetage ou de sous-classes de la classe dans laquelle il est déclaré.

private interdit l'accès à une variable, une méthode ou un constructeur à partir d'autres classes que celle ou il est déclaré.

Accès de paquetage il se produit lorsque public , protected ou private ne sont pas spécifiés. Il s'applique aux classes, interfaces, variables, méthodes et constructeurs. Il permet l'accès à l'élément déclaré à partir de toute classe ou interface du même paquetage.

Modificateurs d'accès

Abstract le modificateur abstract est utilisé pour identifier des classes et des méthodes abstraites. Une classe abstraite reporte son implémentation dans ses sous-classes et ne peut pas avoir d'instances. Les classes abstraites peuvent définir des méthodes abstraites. Une méthode abstraite est une méthode dont l'implémentation est différée

```
Abstract class Meeting{

abstract void formatDate();

}
```

final, il est utilisé pour indiquer qu'un élément déclaré ne peut pas être changé. Quand il est utilisé sur une classe, il empêche toute extension de la classe. Utilisé sur une variable de champ , il indique qu'elle ne peut pas être modifiée une fois qu'une valeur lui a été attribuée. **Les variables de type final servent à créer des constantes.**

static, il est utilisé pour indiquer qu'une variable, une méthode s'applique à une classe dans son ensemble.

POO: Déclaration et Controle d'accés(12)

Exemple:

```
class StaticApp{

static String s = 'ce code est partout';
String t= 'ce code est  limité';
}

public static void main(String[] args){

display(s);
StaticApp app = new staticApp();
App.display(app.t)
}

static void display(String s){
System.out.println(s);
}

}
```