

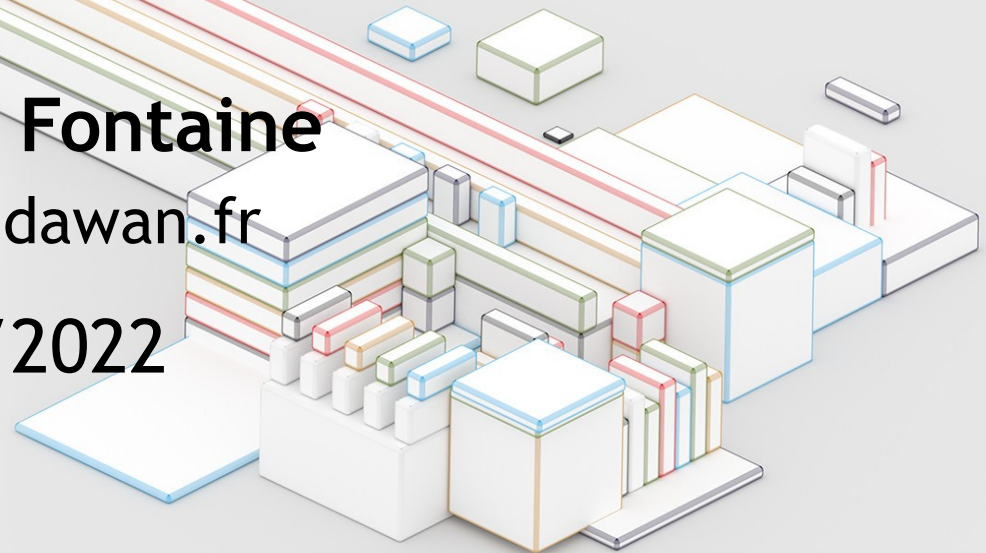
Java SE

Approfondissement

Christophe Fontaine

`cfontaine@dawan.fr`

21/11/2022



Objectifs



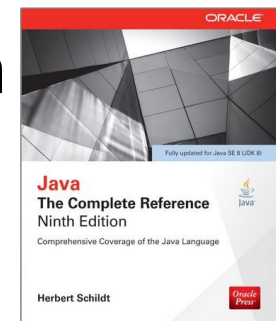
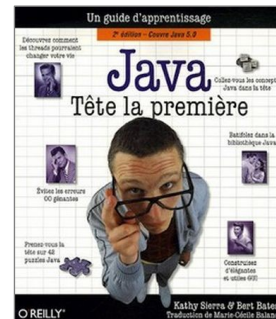
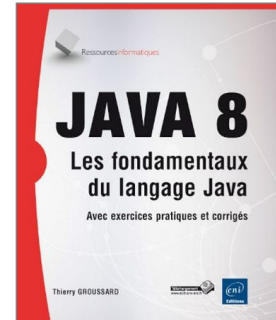
- Connaître et maîtriser les concepts avancés de la programmation Java
- Réaliser et déployer des applications complètes, performantes et maintenables

Durée : 2 jours

Pré-requis : avoir suivi la formation Java initiation
ou posséder les connaissances équivalentes

Bibliographie

- **Java 8 Les fondamentaux du langage Java**
Thierry Groussard
Éditions ENI - Juillet 2014
- **Java Tête la première** (couvre Java 5.0)
Kathy Sierra, Bert Bates
Editions O'REILLY - Novembre 2006
- **Java The Complete Reference**
Herbert Schildt
Edition Oracle Press - 9th edition - Juin 2015
- **Java Platform Standard Edition 8 Documentation**
<https://docs.oracle.com/javase/8/docs/>
- **Développons en Java**
<https://www.jmdoudoux.fr/java/dej/index.htm>



Plan



- Accéder à des bases de données
- Généricité
- Patrons de conception (Design patterns)
- Interfaces graphiques Swing
- Threads
- Introspection (Reflection API)
- Communications réseau (Socket)
- Spécificités de la plate-forme Java

Installation du JDK



- **Téléchargement**

- Oracle JDK : <https://www.oracle.com/java/technologies/downloads/>
- OpenJDK : <https://adoptium.net/>

- **Paramétrages des variables d'environnement**

Dans le menu Windows, taper : **env**

Choisir → Modifier les variables d'environnement du système

↳ Variable d'environnement ...

Dans **variables système**

- Créer une variable **JAVA_HOME** qui contient le chemin vers le dossier contenant le JDK
- Modifier la variable **PATH** en ajoutant **%JAVA_HOME%\bin**

- **Vérification**

> javac -version → doit afficher la version de java

Configuration d'Eclipse



- À partir de eclipse **4.18**, le JRE qui va exécuter eclipse est intégrée sous forme de plugin (openjdk 17)

Il n'est plus nécessaire de le configurer dans le fichier **eclipse.ini** avec l'option **-vm**

-vm

C:\Program Files\Java\jdk1.8.0_351

- Dans windows → préférence

- filtre sur **jre**

Installed JREs → Add → choisir :

- Standard VM
 - JRE home : C:\Program Files\Java\jdk1.8.0_351\jre
 - JRE Name : jdk1.8.0_351

Configuration d'Eclipse



- filtre sur **compiler**

JDK Compliance → Compiler compliance level → 1.8

- filtre sur **text editors**

cocher : Insert spaces for tabs

cocher : remove multiple spaces and backspace/delete

- filtre sur **spelling**

décocher : enable spelling

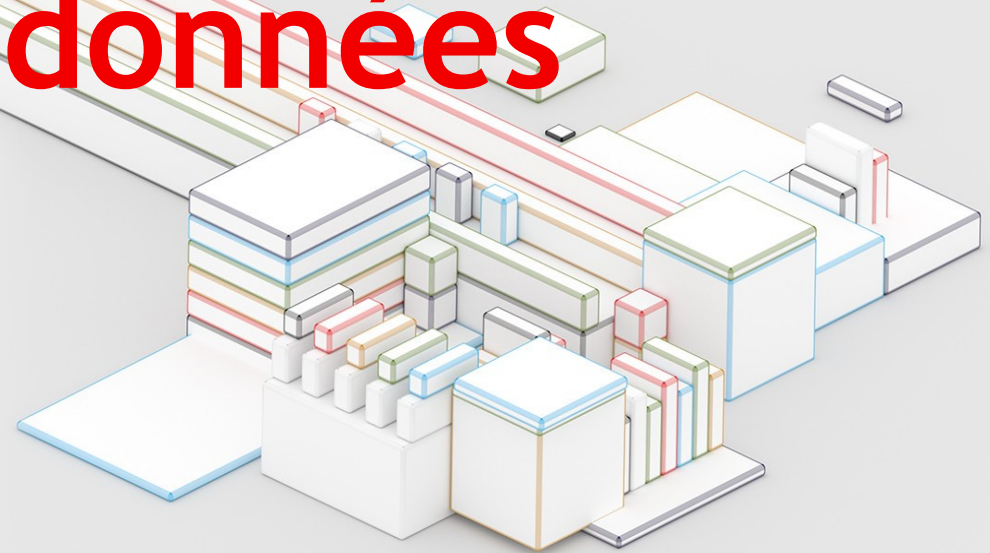
- filtre sur **formatter**

java → code style → Formatter

new → profil name : Eclipse

indentation : tab policy choisir **space only**

Accéder à des bases de données



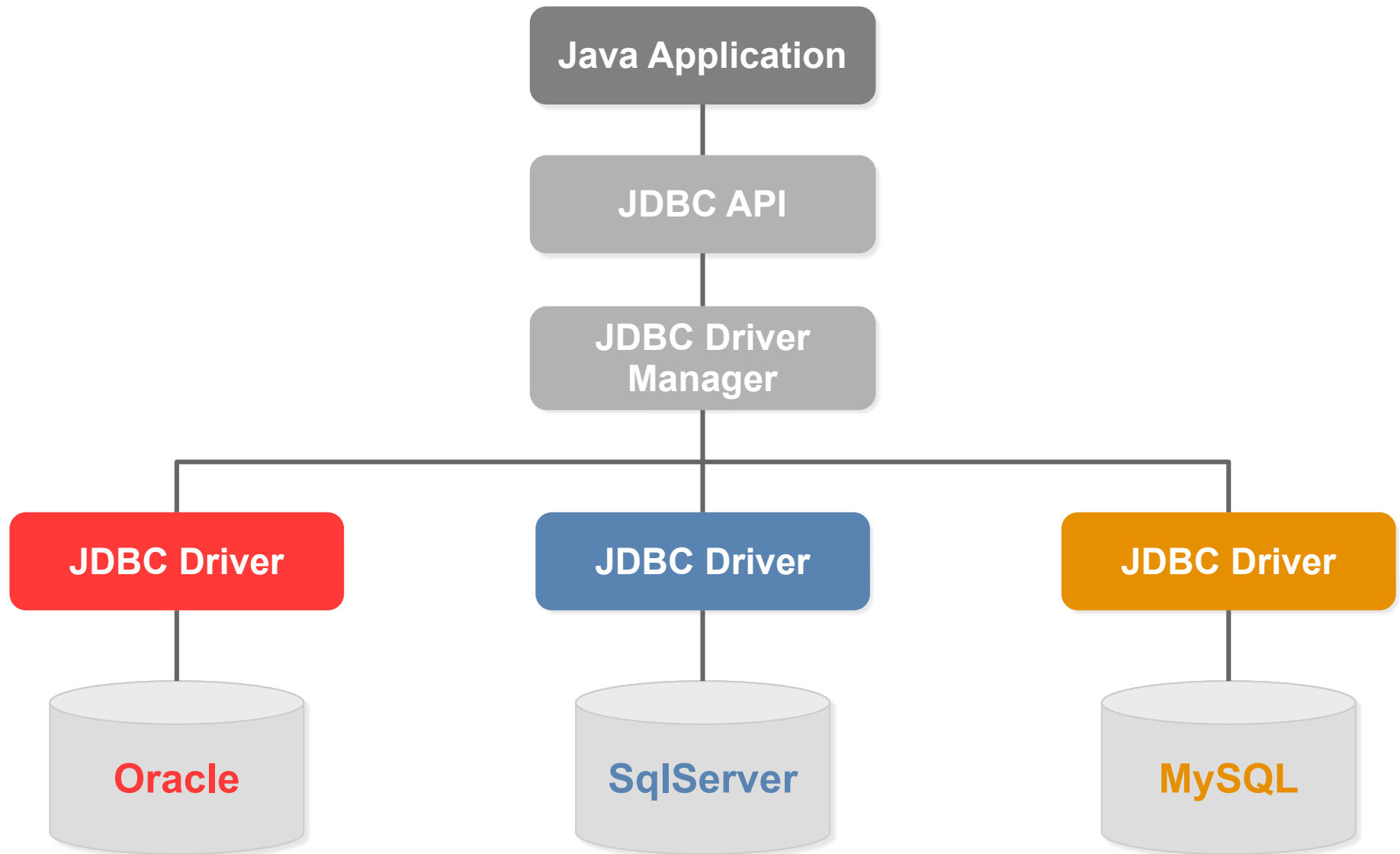
JDBC



- JDBC (**J**ava **D**ata **B**ase **C**onnectivity) est l'API Java pour accéder à des bases de données relationnelles avec le langage SQL
- Les interfaces et les classes se trouvent dans le package **java.sql**
- JDBC ne fournit pas les classes qui implémentent les interfaces
- C'est le driver JDBC qui implémente ces interfaces
- Les drivers dépendent du Système de gestion de base de donnée auquel ils permettent d'accéder

Ils sont fournis par les éditeurs des SGDB ([mysql](#), [mariadb](#), [postgresql](#), [oracle database](#), [SQL server](#) ...)

JDBC



Principe d'accès à une base de données avec JDBC



- Chargement du pilote de la base de données
- Ouverture d'une connexion (**Connection**) à la base de données
- Création d'un objet (**Statement** ou **PreparedStatement**) qui va permettre de d'exécuter la requête SQL sur la connexion
- S'il y en a, récupération du résultat de la requête dans un objet (**ResultSet**), et exploitation de ce résultat
- Fermeture de la connexion

Chargement du pilote de la base de donnée



- Le driver d'une base de données est une classe Java qui implémente de l'interface **java.sql.Driver**
- Il faut ajouter le chemin de la classe du driver dans le classpath (fichier.jar)

- **en ligne de commande**

avec l'option **-classpath** de la commande java

```
> java -classpath CheminDuDriverJDBC.jar
```

- **avec eclipse**

projet → properties

filtre sur **java build path:**

- onglet : libraries
- Add External JARs... → CheminDuDriverJDBC.jar

Chargement du pilote de la base de donnée



- **Avant JDBC 4.0**

Il faut charger dynamiquement la classe du driver par appel de la méthode `Class.forName("nom classe");`

Le nom de la classe varie suivant la base de données

MySQL	<code>com.mysql.cj.jdbc.Driver</code>
MariaDb	<code>org.mariadb.jdbc.Driver</code>
H2	<code>org.h2.Driver</code>
PostgreSQL	<code>org.postgresql.Driver</code>
Oracle	<code>oracle.jdbc.driver.OracleDriver</code>

Chargement du pilote de la base de donnée



- **À partir de JDBC 4.0 (Java 6)**

Lors du chargement d'un JAR, Java examine le contenu du répertoire **META-INF** du fichier

S'il y trouve un fichier **services/java.sql.Driver**, alors il charge les classes définies dans ce fichier et les enregistre en tant que pilotes JDBC

Il n'est plus nécessaire de charger le driver avec `Class.forName()`

Connection



- La méthode **getConnection** de **DriverManager** permet de créer la Connexion
 - `getConnection(String url)`
 - `getConnection(String url, Properties prop)`
 - `getConnection(String url, String user, String password)`
- L'url est composée de 3 parties séparées par :
 - protocole
 - nom du SGDB
 - détails de connexion (spécifiques à la base de données)

pour MySql → `//url du serveur:port/nom de la base`

```
jdbc:mysql://localhost:3306/zoo
```

Statement



- **Création**

Un objet de type **Statement** s'obtient en appelant la méthode **createStatement()** de l'interface **Connection**

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

- **Exécution**

- **executeQuery(String req)**
retourne un objet de type **ResultSet** (SELECT)

```
ResultSet rs = smt.executeQuery("SELECT nom, prix FROM articles");
```

- **executeUpdate(String req)**
retourne un nombre d'objets modifiés
(INSERT, DELETE, UPDATE)

```
int count = smt.executeUpdate("DELETE FROM articles");
```


Prepared Statement

- L'interface **PreparedStatement** étend **Statement**
- Il ajoute la possibilité de paramétrer des requêtes SQL
- Les instances de **PreparedStatement** s'utilisent quand une même requête doit être exécutée plusieurs fois, avec des paramètres différents

```
PreparedStatement ps = connection.prepareStatement(  
    "INSERT INTO Articles (nom, prix) values (?, ?)");
```

- ? → paramètres, dont la valeur est donnée avec les méthodes **setType(int numParam, Type valeur)**
- L'indexation des paramètres commence à **1**

```
ps.setString(1, "batterie");  
ps.setDouble(2, 76.0);
```

Clés générées

- Pour récupérer la valeur de la clé primaire générée par la base de donnée:
 - On ajoute en paramètre à `executeUpdate` après la requête : **`Statement.RETURN_GENERATED_KEYS`**
 - Après l'exécution de la requête, on récupère la clé primaire générée avec la méthode **`getGeneratedKeys()`**

```
Statement stmt = connection.createStatement();
int rowCount = stmt.executeUpdate("INSERT INTO articles (nom,
prix) VALUES ('Stylo', 1.7)", Statement.RETURN_GENERATED_KEYS);
ResultSet rs = stmt.getGeneratedKeys();
if (rs.next()) {
    // récupération de la clé primaire
    int idStylo = rs.getInt(1);
}
```

ResultSet



- Le résultat d'une requête est stocké dans un objet de type **ResultSet**
- Un **ResultSet** peut être vu comme un tableau de résultats, dont chaque colonne est un champ, et chaque ligne un enregistrement
- La méthode **next()** : déplace le curseur sur la ligne suivante. Elle retourne **false** si le curseur a dépassé la fin du tableau, sinon elle retourne **true**
- Les méthodes
Type **getType(String nomColonne)**
Type **getType(int numColonne)**
permettent de lire la valeur des colonnes
- L'indexation des colonnes commence à 1

JDBC Example

```
try {  
    Class.forName( "com.mysql.jdbc.Driver" );  
    cnx = DriverManager.getConnection(  
        "jdbc:mysql://localhost/test","root", "");  
    Statement st = cnx.createStatement();  
    ResultSet rs = st.executeQuery("SELECT * FROM articles");  
    while(rs.next()) {  
        // ...  
        rs.getString(1)  
        // ...  
    }  
    cnx.close();  
}catch( ) {  
    // ...  
}
```

Transaction



- Une transaction est un ensemble d'une ou plusieurs requêtes exécutées en tant qu'unité, de sorte que soit toutes les requêtes sont exécutées, soit aucune des requêtes n'est exécutée
- **Désactivation du mode auto-commit**
Par défaut un connexion est en mode auto-commit, Chaque instruction SQL est automatiquement validée juste après son exécution

Pour grouper plusieurs requêtes dans une transaction il faut désactiver le mode auto-commit

```
cnx.setAutoCommit (false);
```

Transaction (commit et rollback)



- La méthode **commit()** de Connexion permet de valider les requêtes
- Sinon la méthode **rollback()** va permettre d'annuler les requêtes

```
try{
    cnx.setAutoCommit(false);
    Statement stmt = cnx.createStatement();
    String SQL = "INSERT INTO Employees
                  VALUES (2000, 'John', 'Doe')";
    stmt.executeUpdate(SQL);
    // ... autre requête
    cnx.commit();
} catch(SQLException se){
    cnx.rollback();
}
```

DAO



- Le pattern **DAO** (Data Access Object) permet d'isoler la couche métier de la couche de persistance
 - Permet de centraliser les requête SQL dans un seul objet
 - Permet de changer facilement de système de stockage de données (Bdd, XML ...)
- Avec l'objet DAO, on va réaliser les opérations CRUD
 - Créer l'objet en base (INSERT)
 - Rechercher l'objet en base pour le recréer (find, RETRIEVE)
 - Mettre à jour l'objet en base (UPDATE)
 - Supprimer l'objet en base (DELETE)
- On aura **un DAO par objet Métier**

Object Relational Mapping



- Concept permettant de connecter un modèle objet à un modèle relationnel
- Couche qui va interagir entre l'application et la base de données
- **Avantages**
 - Gain de temps au niveau du développement d'une application
 - Abstraction de toute la partie SQL
 - La portabilité de l'application d'un point de vue SGBD
- **Inconvénients**
 - L'optimisation des frameworks/outils proposés
 - La difficulté à maîtriser les frameworks/outils

ORM: JPA

- Une API (Java Persistence API)
- Des implémentations



- Permet de définir le mapping entre des objets Java et des tables en base de données
- Remplace les appels à la base de données via JDBC

ORM: Exemple Entité

```
@Entity
@Table(name="products")
public class Product {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;

    private String name;
    private double price;

    @ManyToMany(cascade=CascadeType.ALL)
    @JoinTable(name="products_suppliers")
    private Set<Supplier> suppliers;

    ...
}

-----

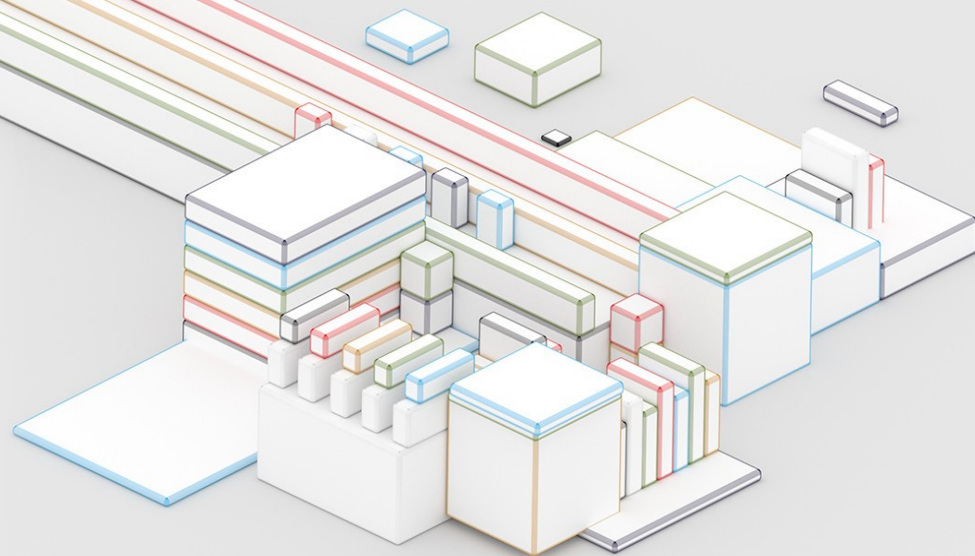
@Entity
@Table(name="suppliers")
public class Supplier {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;

    @Column(nullable=false, unique=true, length=200)
    private String name;

    @ManyToMany(mappedBy="suppliers", cascade=CascadeType.ALL)
    private Set<Product> products;

    ...
}
```

Généricité



Généricité



- Depuis la version 5.0, Java autorise la définition de classes et d'interfaces contenant un (des) paramètre(s) représentant un (des) type(s)
- Cela permet d'écrire une structure qui pourra être personnalisée au moment de l'instanciation à tout type d'objet
- **Motivation** : Homogénéité garantie
- **Inférence de type** :
 - principe d'erasure à la compilation
 - pas de duplication de code
- **Aucune incidence sur la JVM** : les casts restent en interne mais deviennent sûrs (sans levée d'exceptions)

Classe Générique

class name<T1, T2, ... , Tn> { /* ... */ }

```
public class Box<T> {  
    private T t;  
    public void set(T t) { this.t = t; }  
    public T get() { return t; }  
}  
  
public static void main(String[] args){  
    Box<Integer> intBox = new Box<>();  
    intBox.set(42);  
    Box<String> strBox = new Box<>() ;  
    strBox.set("Test");  
}
```

- **Conventions de nommage des paramètres de type**

E : Element **K** : Key **N** : Number

T : Type **V** : Value **S,U,V ...** : 2ème, 3ème, 4ème type

Méthode Générique

- Les méthodes génériques sont des méthodes qui introduisent leurs propres paramètres de type
- Les méthodes d'instance, de classe et les constructeurs peuvent être des méthodes génériques

```
public class GenMethod {  
    public static <T> boolean isEqual(T v1, T v2){  
        return v1.equals(v2);  
    }  
  
    public static void main(String args[]){  
        boolean t1=GenMethod.<String>isEqual("test","hello");  
        boolean t2=GenMethod.<Integer>isEqual(42,30+12);  
    }  
}
```

Contraintes sur les types génériques



- Les génériques permettent d'imposer qu'un type T en étend un autre (classe ou interface)

```
public class MyClass<T extends Comparable>{...}
```

- Contraintes multiples sur un type générique → &

```
public class MyClass<T extends Comparable & Cloneable>{...}
```

La liste des types

- ne peut comporter qu'une unique classe, qui doit être déclarée en premier
- peut comporter plusieurs interfaces

Implémentation des génériques



- **Construction d'une instance**

On ne peut pas construire une nouvelle instance d'un type T avec : `new T()` ou `T.class.newInstance()`

Il faut utiliser :

```
public static <T> T newInstance(Class<T> clazz){  
    return clazz.newInstance();  
}
```

- **Membres statiques**

On ne peut pas référencer un type générique déclaré au niveau de la classe dans des membres statiques

```
public class MyClass<T> {  
    private static T t ; // erreur  
    public static T newInstance(T t) { //... } // erreur
```


Type <?>

- Une classe générique ne peut étendre aucune version d'elle-même

List<T> n'étend jamais **List<U>**, quelle que soit la relation entre **T** et **U**

```
List<String> ls = new ArrayList<String>();  
List<Object> lo = ls; // Ne compile pas
```

- Pour résoudre ce problème, on peut utiliser ? (wildcard)
List<?> correspond à une liste de type inconnue

```
List<String> ls = new ArrayList<String>();  
List<?> lo = ls;
```

- Pour éviter d'ajouter d'autre éléments que des String dans la **List<?>**, On interdit l'utilisation des méthodes qui prennent ? en paramètre

Contraintes sur les type<?>



- **Type ? extension d'un type**

On peut imposer que le type ? en étende un autre

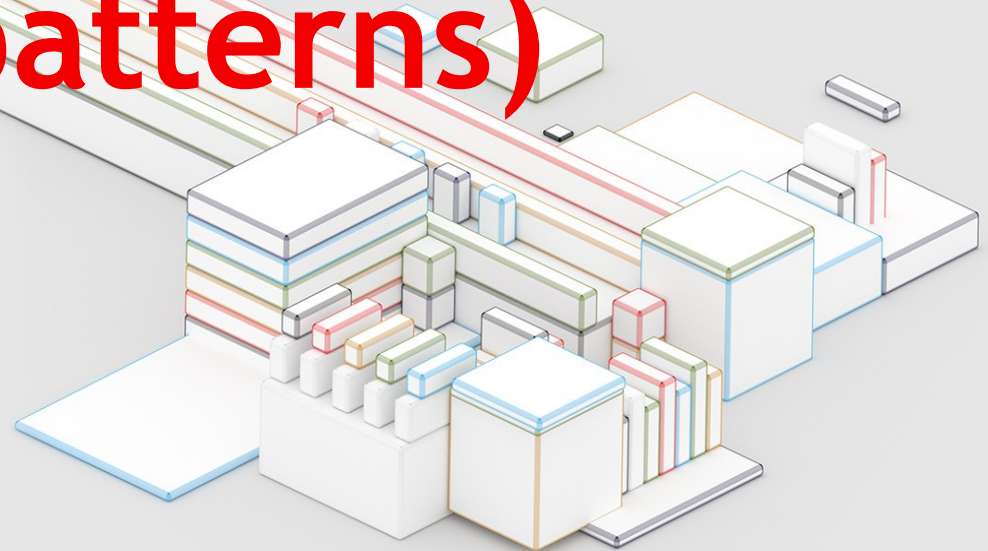
```
List<Integer> lstInt = Arrays.asList(5, 2, 10) ;  
List<? extends Number> lstNum = lstInt ;  
//lstNum.add((Integer)2) ; // Erreur ne compile pas  
int val = (int) lstNum.get(1) ; // 2
```

- **Type ? super-type d'un type**

On peut imposer d'être le super-type d'un type donné, avec
? **super** T (tous les types dont le type T est un type dérivé)

```
List<Integer> lstInt = Arrays.asList(1, 2, 3) ;  
List<? super Integer> lstNum = lstInt ;  
lstNum.add(2) ; //ok  
// Compile, mais n'est pas sûr(le type de retour peut  
int i = (Integer)lstNum.get(0); // être Object)
```

Patrons de conception (Design patterns)



Présentation

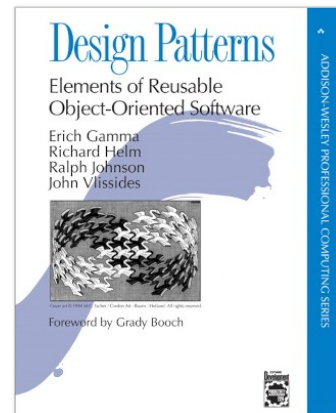
1995 : GoF « Gang of Four »

Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides

Auteurs du livre de référence :

Design Patterns :Elements of Reusable Object-Oriented Software

- Solution standard pour un problème d'architecture
 - Solution à disposition
 - Éprouvée, de qualité
 - Connue, permettant la communication
- à appliquer au code, particulièrement de POO
- 23 patterns répertoriés dans 3 classes



<https://refactoring.guru/fr/design-patterns>

Création d'objets sans instanciation directe d'une classe

Singleton Classe qui ne pourra avoir qu'une seule instance

Factory Introduction d'une méthode abstraite de création d'un objet en reportant aux sous-classes concrètes la création effective

Abstract Factory Création d'objets regroupés en famille sans devoir connaître les classes concrètes destinées à la création de ces objets

Prototype Création de nouveaux objets par duplication d'objets existants appelés prototypes qui disposent de la capacité de clonage

Builder Séparation de la construction d'objets complexes de leur implantation de sorte qu'un client puisse créer ces objets avec des implantations différentes

Patterns de composition (de structure)



Composition de groupes d'objets

Adapter	Conversion de l'interface d'une classe existante pour garantir la collaboration entre clients de cette interface
Bridge	Séparation des aspects conceptuels d'une hiérarchie de classes de leur implantation
Composite	Offrir une profondeur variable à la composition d'objets conception basée sur un arbre
Decorator	Ajout dynamique de fonctionnalités supplémentaires à un objet
Facade	Regroupement des interfaces d'un ensemble d'objets pour le simplifié (interface unifiée)
Flyweight	Faciliter le partage d'un ensemble important d'objet
Proxy	Construction d'un objet se substituant à un autre objet et qui contrôle son accès

Modélisation des communications inter-objets et du flot de données

Interpreter	fournir un cadre objets pour évaluer/interpréter les expressions d'un langage
Iterator	parcours (accès séquentiel) d'une collection d'objets
Memento	sauvegarder et restaurer l'état d'un objet
Observer	construire une dépendance entre un sujet et des observateurs (avec notifications)
State	adaptation du comportement d'un objet en fonction de son état interne
Strategy	adaptation du comportement et des algorithmes d'un objet en fonction d'un besoin sans changer les interactions avec le client
Template Method	report dans les sous-classes d'une étape de création d'un objet
Visitor	construction d'une opération à réaliser sur les éléments d'une collection d'objets ; ajout d'opérations sans modification de la classe de ses objets

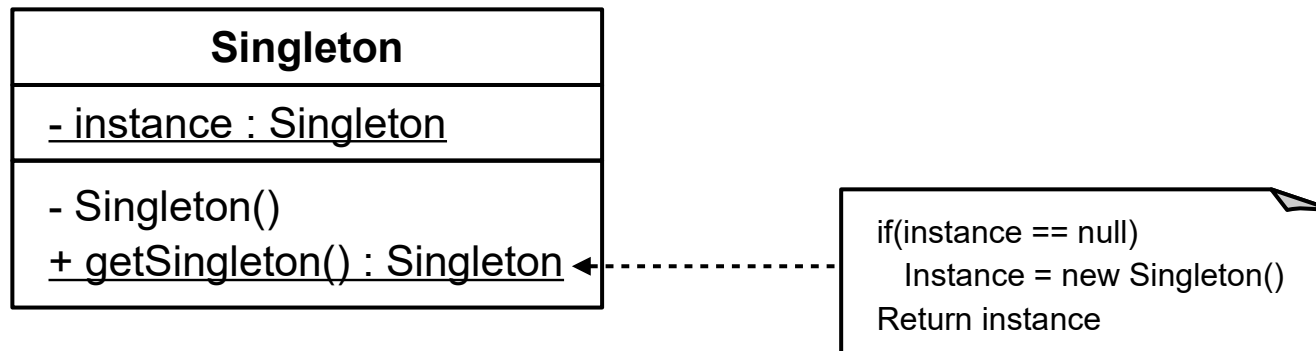
Singleton

- **Objectif**

A tout moment une (et une seule) instance d'une classe existe pour une application

- **Solution**

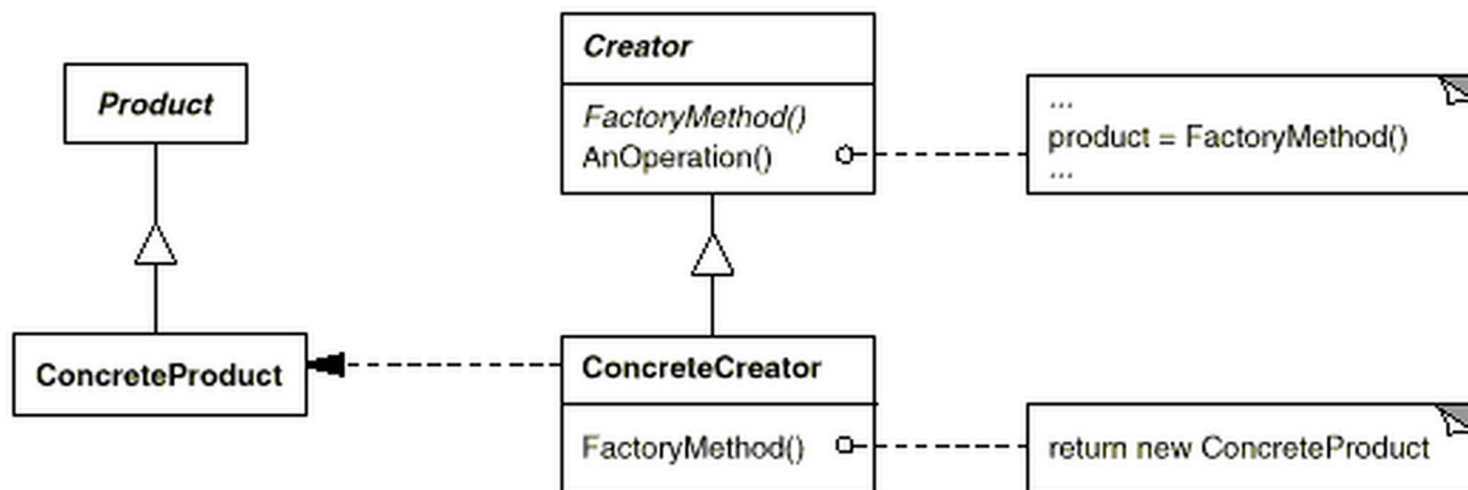
La classe possède une méthode statique **getInstance()**, qui renvoie une instance statique, et a son constructeur privé



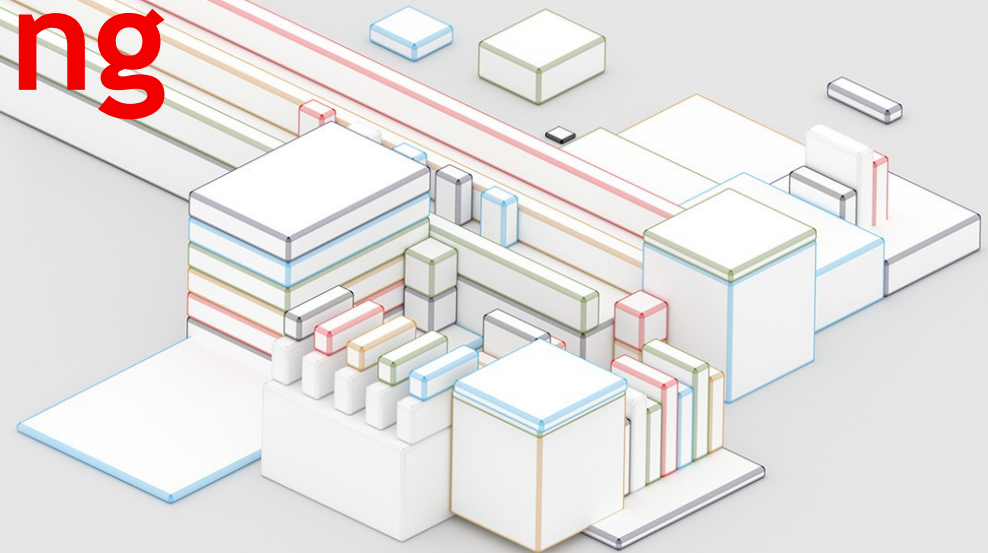
Factory

On utilise le **FactoryMethod** lorsque :

- une classe ne peut anticiper la classe de l'objet qu'elle doit construire
- une classe délègue la responsabilité de la création à ses sous-classes, tout en concentrant l'interface dans une classe unique



Interfaces graphiques Swing



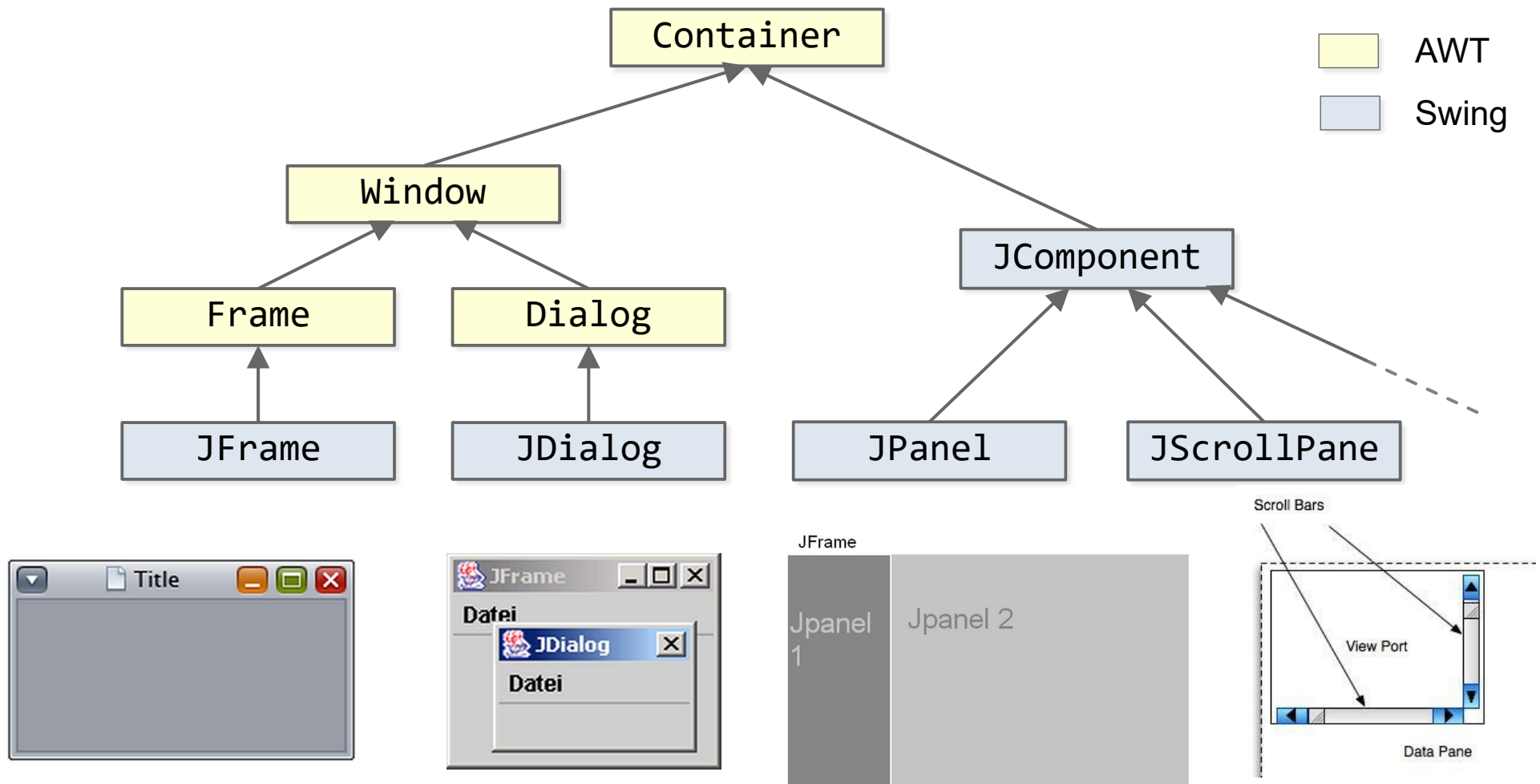
Définition



- Swing est une partie de « Java Foundation Class »
 - Bibliothèques produisant des interfaces graphiques
 - Drag & Drop, i18n, Java 2D, Accessibility, AWT
- Avantages de Swing par rapport à AWT
 - Résout les problèmes de portabilité
 - AWT est utilisé dans les couches basses de Swing

Conteneurs

- Un Conteneur peut regrouper plusieurs composants



JFrame

```
public static void main(String [] args) {  
    // Créer la fenêtre  
    JFrame frame = new JFrame("Ma fenêtre");  
    // Configurer le comportement de la fenêtre à la fermeture  
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    // Créer des composants et les placer dans la fenêtre  
    frame.getContentPane().add(new JButton("OK"));  
    // Dimensionner la fenêtre en fonction de son contenu  
    frame.pack();  
    // Afficher la fenêtre  
    frame.setVisible(true);  
}
```

- Les constructeurs **JFrame()** ou **JFrame(String)** permettent de créer la fenêtre principale , elle est initialement invisible
- Les méthodes **setTitle(String)** et **getTitle()** permettent de modifier et d'obtenir le titre de la fenêtre

- **Définir les dimensions et la position de la fenêtre**
 - **setSize(int,int)** et **getSize()** → définir et obtenir la largeur et la hauteur de la fenêtre
 - **setLocation(int, int)** et **getLocation()** → définir et obtenir la position du coin supérieur gauche de la fenêtre
 - **setBounds(int,int,int,int)** et **getBounds()** → définir et obtenir la largeur, la hauteur et la position de la fenêtre
 - **pack()** → dimensionner la fenêtre pour que tout son contenu soit égal ou supérieur à **preferredSize**
- **setDefaultCloseOperation** → permet de configurer le comportement de la fenêtre à la fermeture
 - **DO_NOTHING_ON_CLOSE** : ne rien faire
 - **HIDE_ON_CLOSE** (par défaut) : cache la fenêtre, mais le programme continue de s'exécuter

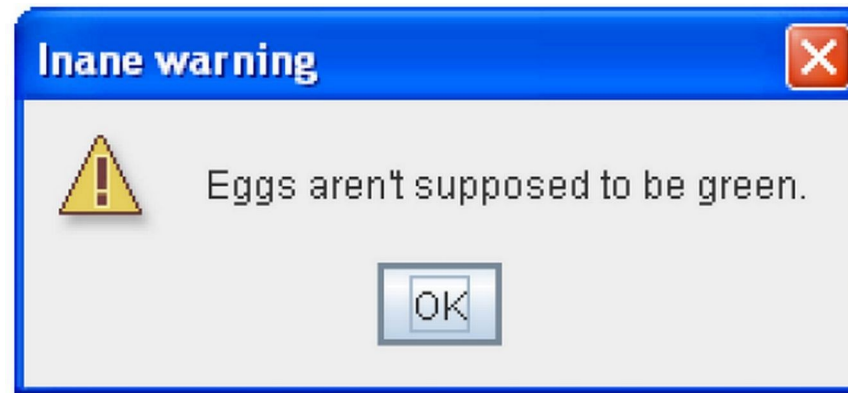
JFrame



- **DISPOSE_ON_CLOSE** → détruit la fenêtre, mais le programme continue de d'executer
- **EXIT_ON_CLOSE** → détruit la fenêtre, et arrête le programme
- **setVisible(boolean)** → afficher (true) ou cacher (false) la fenêtre
- **getContentPane()** → obtenir le content pane qui contient les composants de la fenêtre

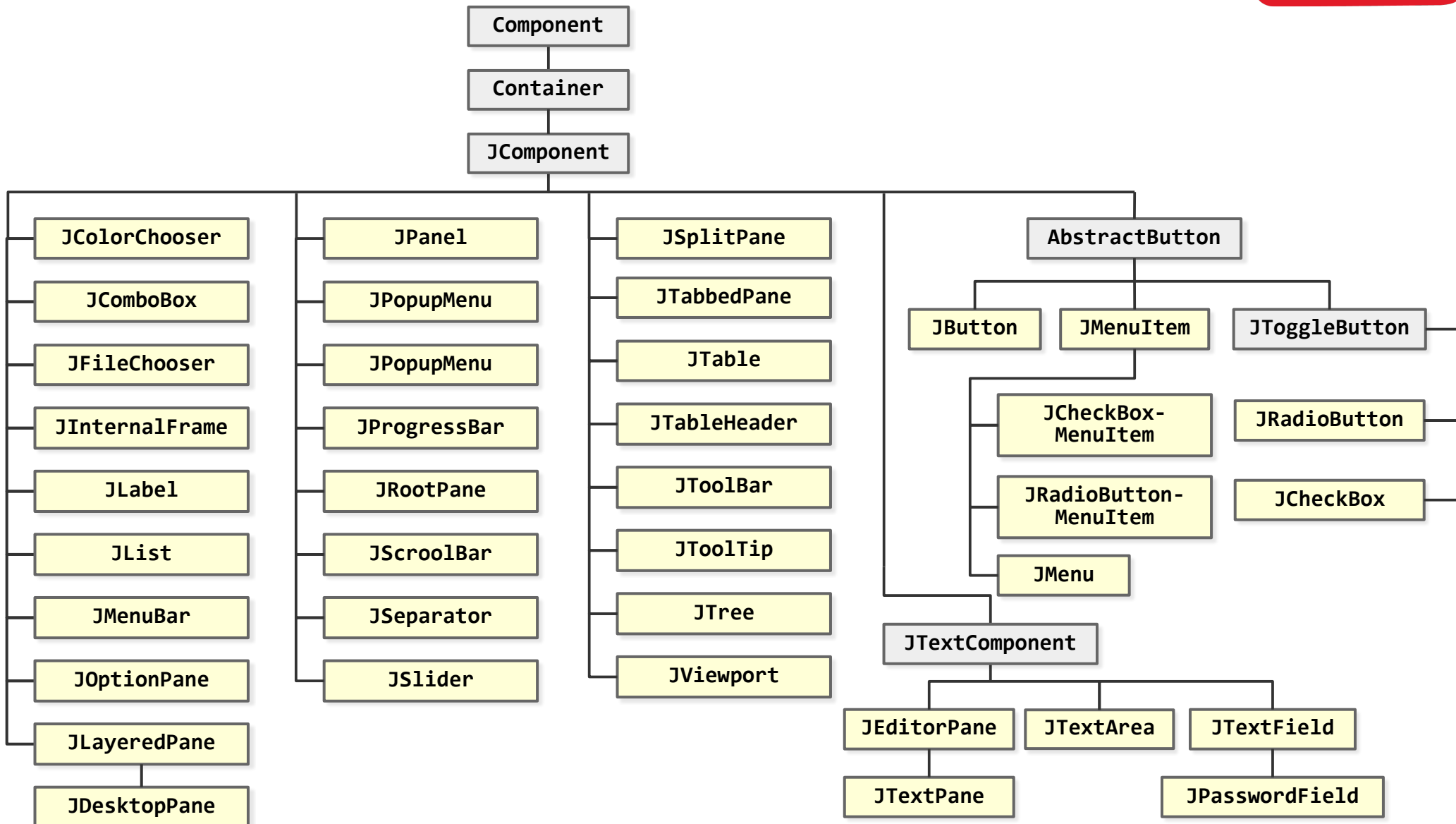
Boites de dialogue

- Parmi les conteneurs
- Déclenchées depuis un conteneur (la fenêtre mère)
- Modales



```
JOptionPane.showMessageDialog(jf,  
    "Les oeufs ne doivent pas être verts.",  
    "Avertissement",  
    JOptionPane.WARNING_MESSAGE);
```


Composants



Composants simples

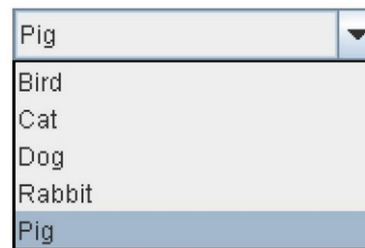
- JButton



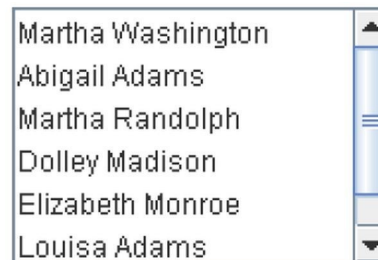
- JLabel et JTextfield



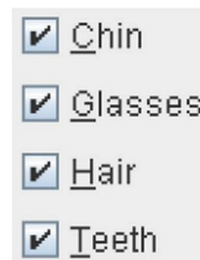
- JComboBox



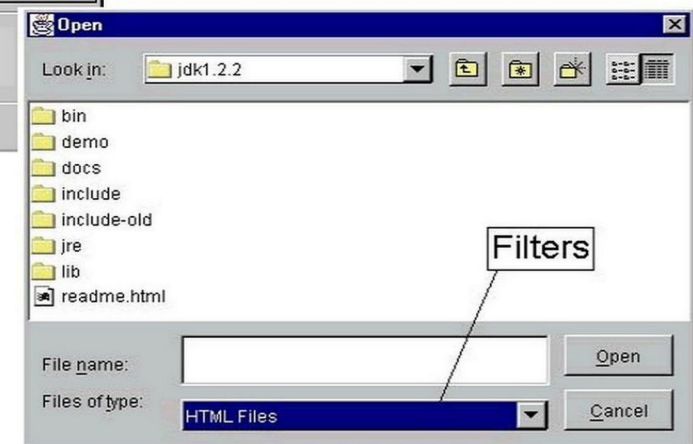
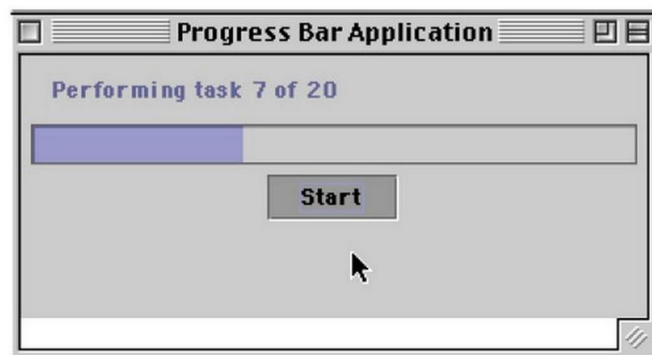
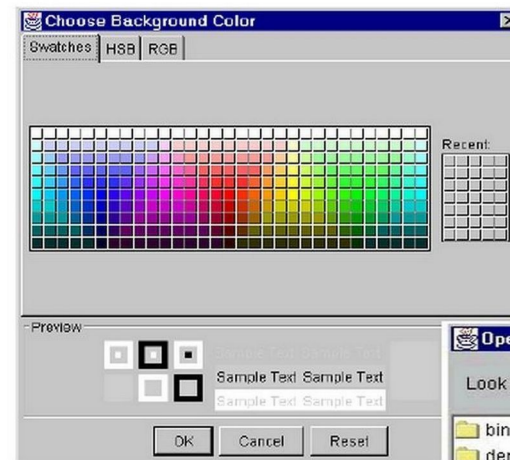
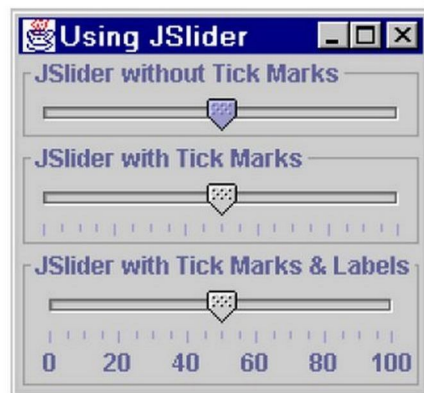
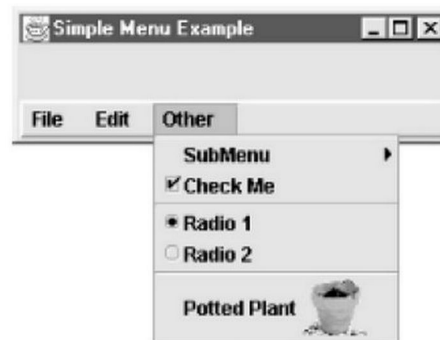
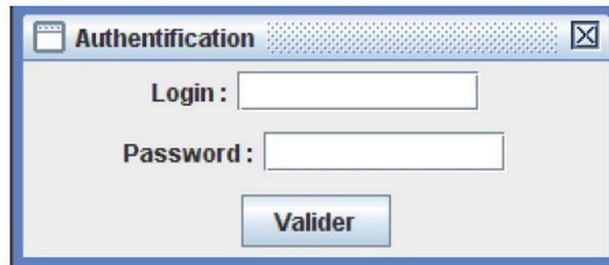
- JList



- JCheckBox



Widgets



Positionnement des composants



- Pour positionner les composants, on peut :
 - le faire manuellement → **positionnement absolue**
 - utiliser des gestionnaires de positionnement
 - ↳ **LayoutManagers** : FlowLayout, BorderLayout, GridLayout, BoxLayout ...

Ils permettent de calculer automatiquement la disposition spatiale des composants dans les conteneurs

- A chaque conteneur est associé un LayoutManager
 - qui dépend du type de conteneur
 - qui peut être changé par la méthode **setLayout()** :
`conteneur.setLayout(unAutreLayoutManager)`

Espacement entre les composants



- Pour tout les Layouts, on peut mettre des espaces entre les composants en ajoutant des bordures vides (EmptyBorder) aux composants (généralement JPanel ou JLabel)

```
JLabel label = new JLabel("Exemple");  
label.setBorder(new EmptyBorder(10,5,10,5));
```

- Pour certain Layout (**FlowLayout**, **BorderLayout**, **GridLayout** ...), on peut définir un espacement verticale ou horizontal entre les composants avec le constructeur ou les méthodes **setVgap(int)**, **setHgap(int)**

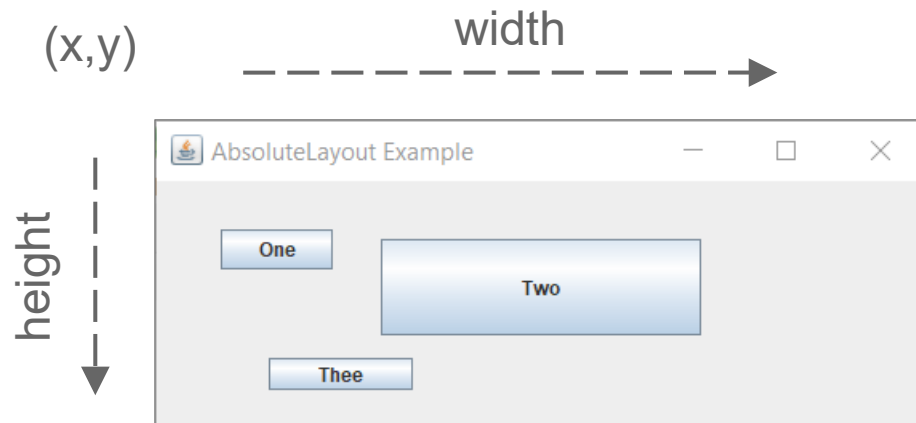
```
JPanel panel = new JPanel();  
BorderLayout l=new BorderLayout(5, 5); // 5 -> vgap, 5 -> hgap  
l.setHgap(20);  
l.setVgap(10);  
panel.setLayout(l);
```

Positionnement Absolu

- Le programmeur détermine manuellement l'emplacement de chaque composant :

Coordonnées :

- x
- y
- width
- height



- Pour passer à un positionnement absolue, on passe **null** à la méthode **setLayout** du conteneur
- Pour obtenir et modifier les dimensions d'un composant

```
Dimension dimension = jComponent.getPreferredSize();  
jComponent.setBounds (x, y, width, height);
```

Positionnement Absolu

```
public void initialise() {  
    // choisir un layout absolu  
    setLayout(null);  
    // observer la taille préférée  
    Dimension size = jLabel.getPreferredSize();  
    // et l'utiliser  
    jLabel.setBounds(25, 5, size.width, size.height);  
    size = getJTextField().getPreferredSize();  
    getJTextField().setBounds(25, 20, size.width, size.height);  
    size = getJButton().getPreferredSize();  
    getJButton().setBounds(25, 40, size.width, size.height);  
    // choisir la taille du conteneur (après pack())  
    setSize(300, 125);  
}
```

FlowLayout

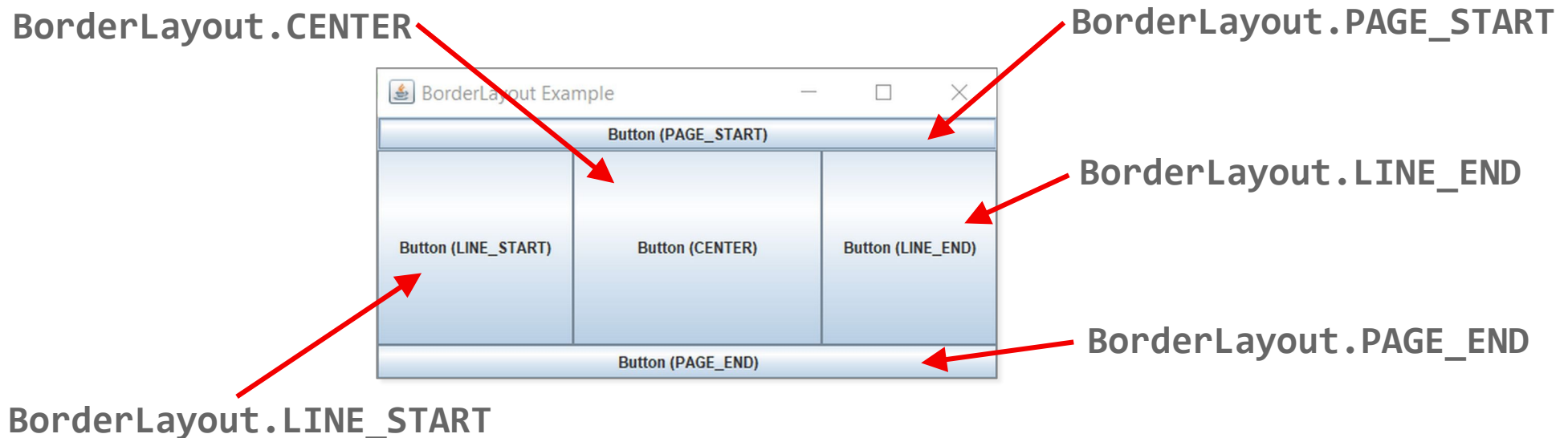
- **FlowLayout** dispose les composants sur une seule ligne et commence une nouvelle ligne si son conteneur n'est pas suffisamment large
- C'est le layout manager par défaut des panneaux (**JPanel**)



- On peut configurer avec le constructeur et la méthode `setAlignment(int)` l'**alignement** des composants
LEFT, **RIGHT**, **CENTER** (par défaut), par rapport au bord arrière **TRAILING** ou avant **LEADING** de l'orientation du conteneur
- L'espacement horizontal et vertical entre les composants est par défaut égal à 5

BorderLayout

- **BorderLayout** divise le conteneur en 5 régions et dans chaque région, un seul composant occupe toute la place
- Layout par défaut du **ContentPane** (**JFrame** et **JDialog**)



- L'emplacement du composant est passé en paramètre à la méthode **add** du conteneur

```
container.add(component, BorderLayout.CENTER);
```

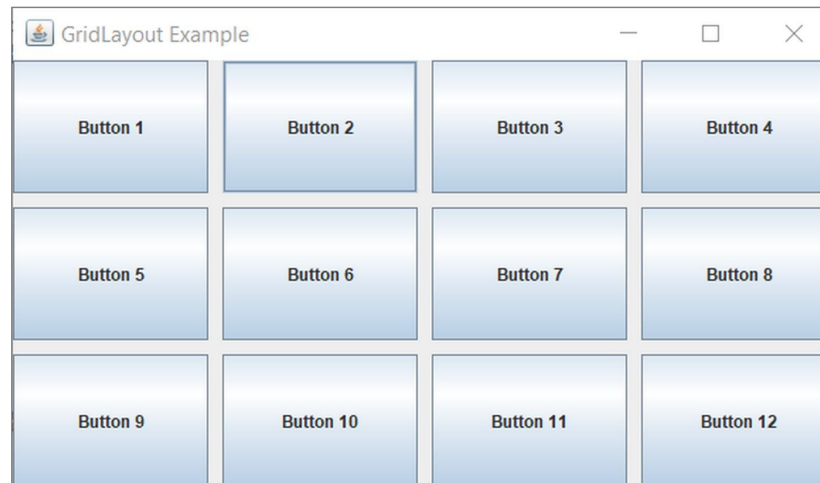
BorderLayout

- Par défaut, il n'y a pas d'espacement horizontal et vertical entre les composants

```
public void initialise( ) {  
    // ...  
    JPanel p = new JPanel();  
    p.setLayout(new BorderLayout());  
    p.add(new JLabel("Gauche"), BorderLayout.LINE_START);  
    p.add(new JLabel("Bas"), BorderLayout.PAGE_END);  
    // ...  
    ... .add(p);  
}
```

GridLayout

- **GridLayout** place les composants dans un tableau, chaque composant prend tout l'espace disponible dans sa cellule



- On définit le nombre de ligne et de colonne avec le constructeur : **GridLayout(int nbCol,int nbRow)**

```
GridLayout gridLayout = new GridLayout(2,3);  
jPanel = new JPanel();  
jPanel.add(new JButton("Ok"));  
jPanel.add(new JButton("Cancel")); // ...
```

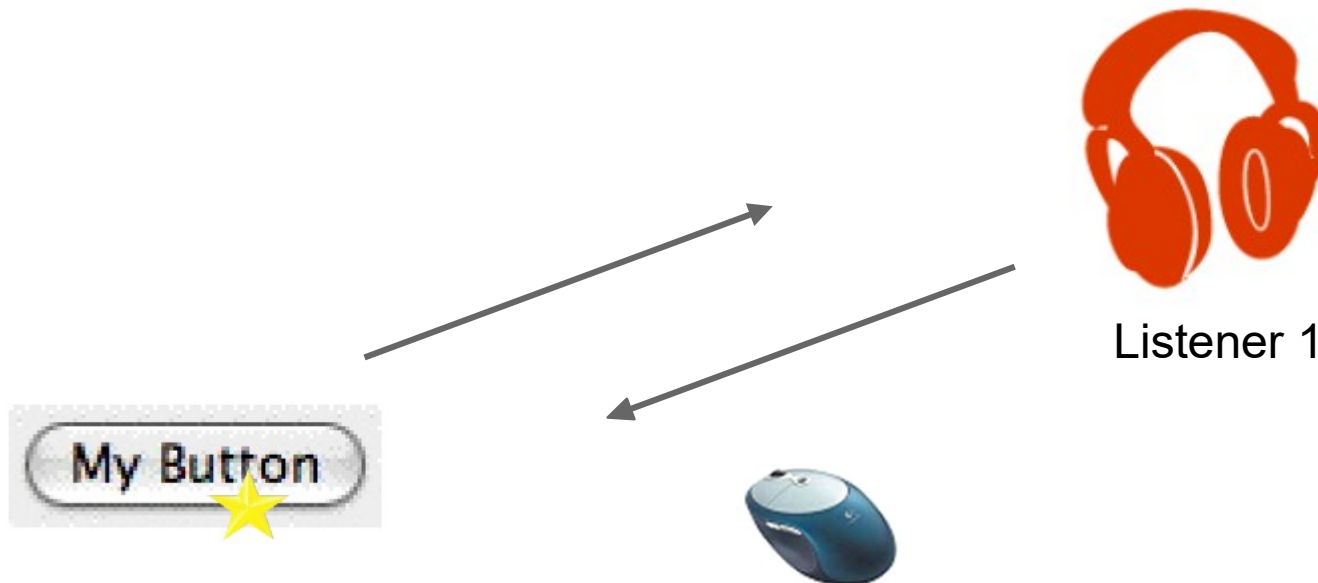
Autres Layouts



- D'autres gestionnaires de positionnement existent mais sont moins utilisés :
 - **CardLayout** → gère les composants de telle manière qu'un seul composant est visible à la fois (empile les composants)
 - **BoxLayout** → pour organiser les composants verticalement ou horizontalement
 - **GridBagLayout** → place les composants dans une grille et permet à ces composants de s'étendre sur plusieurs lignes ou colonnes
 - **SpringLayout** → organise les composants selon un ensemble de contraintes (distance horizontale et verticale entre 2 composants)
 - **GroupLayout, OverlayLayout, ScrollPaneLayout, ViewPortLayout**

Gestion des Évènements

- Pour intercepter un événement, un listener doit être associé au composant
- Le listener est appelé lors de l'évènement
- Il peut y avoir plusieurs listener pour un même événement sur un même composant



Actions sur un JButton



- **ActionListener** est une interface
- Chaque composant proposant des événements “action” dispose de :
 - `addActionListener(ActionListener l)`
 - `removeActionListener(ActionListener l)`
 - `ActionListener[] getActionListeners()`

Classe ActionEvent



- Appartient au package `java.awt.event`
- Permet de gérer les évènements
- Il existe d'autres types de Listener dans `java.awt.event` :
 - `FocusListener`
 - `WindowListener`
 - `ContainerListener`
 - `MouseListener`
 - `KeyListener`
 - ...

Interface ActionListener

- Association du Listener et actions en même temps (classe anonyme) :

```
JButton jButton = new JButton("Cliquer !");  
// Ajout d'un action listener  
jButton.addActionListener(new java.awt.event.ActionListener()  
{  
    public void actionPerformed(java.awt.event.ActionEvent e)  
    {  
        // Faire une action quand il est cliqué  
    }  
});
```


Interface ActionListener

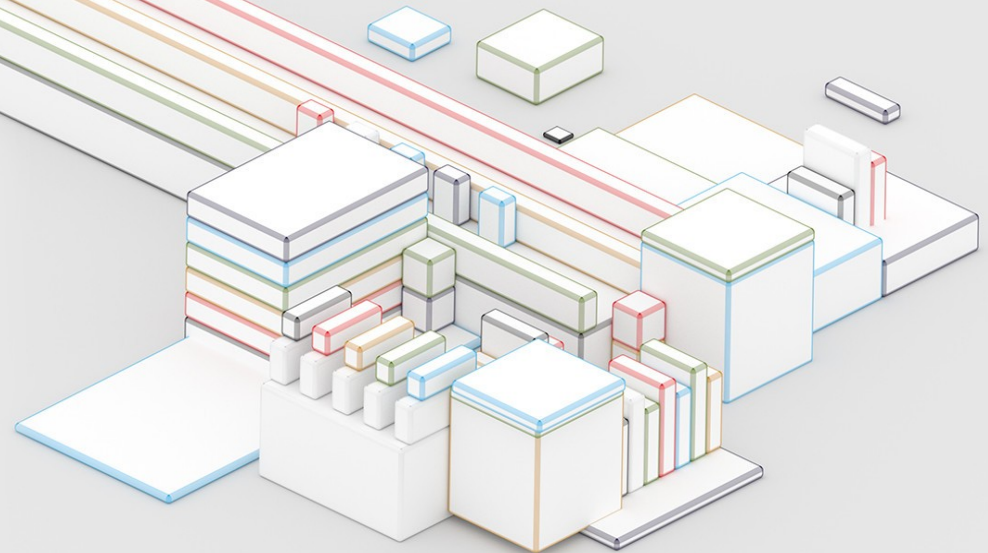
- Plus propre, dans une autre classe

```
public class SubscriptionAction implements ActionListener {  
    public void actionPerformed(ActionEvent e) {  
        // action command permet d'utiliser un  
        // Listener avec plusieurs composants  
        if(e.getActionCommand().equals("quitter"))  
            // Action Exit  
    }  
}
```

- Dans l'interface graphique

```
jButton.setActionCommand("quitter");  
jButton.addActionListener(new  
SubscriptionAction());
```

Threads



Threads



- Un processus léger (unité réelle d'exécution d'un programme)
- Exécute une ou plusieurs tâches
- Partage de mémoire et de temps CPU

Avantages :

- Permettent de faire du multi-tâche
- Faire des boucles infinies
- Création de processus fils

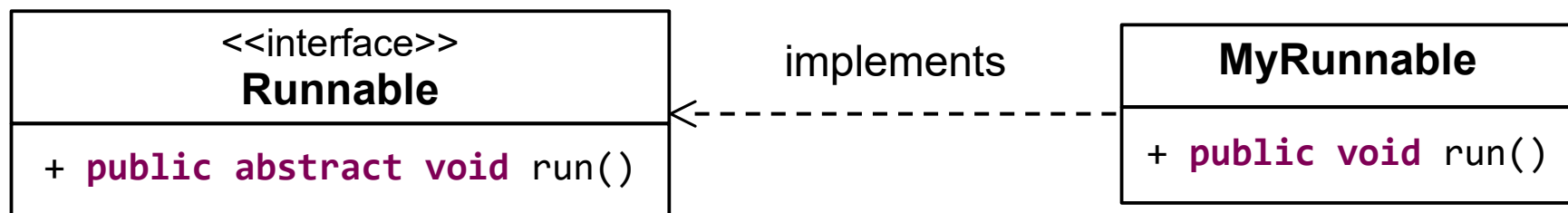
Création et la manipulation de threads (package : java.lang)

- Implémenter l'interface **Runnable**
- Hériter de la classe **Thread**

L'interface Runnable

- L'interface **Runnable** ne contient qu'une seule méthode :
void run()

Toutes les classes à exécuter dans un thread doivent implémenter cette interface et redéfinir la méthode run()



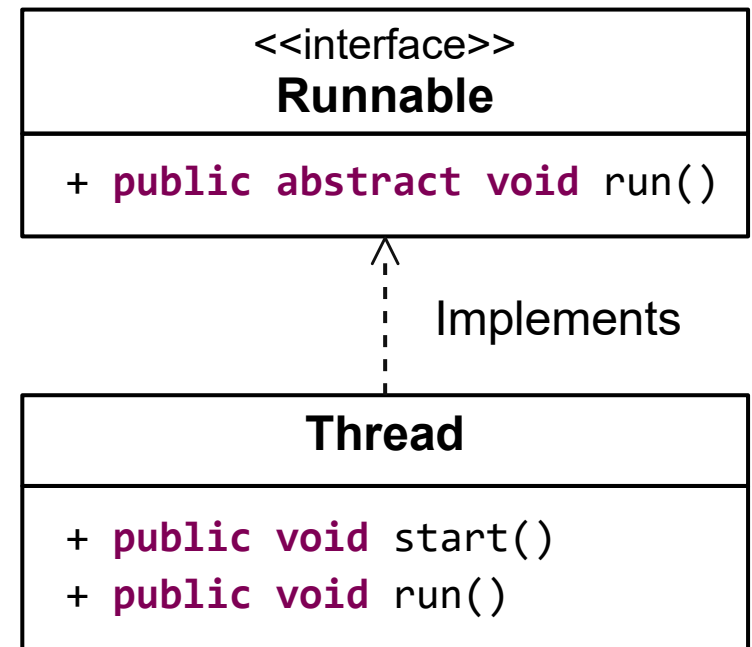
```
public class MyRunnable implements Runnable{
    public void run() { /* Instructions */ }
}
```

- **Pour instancier le Thread**

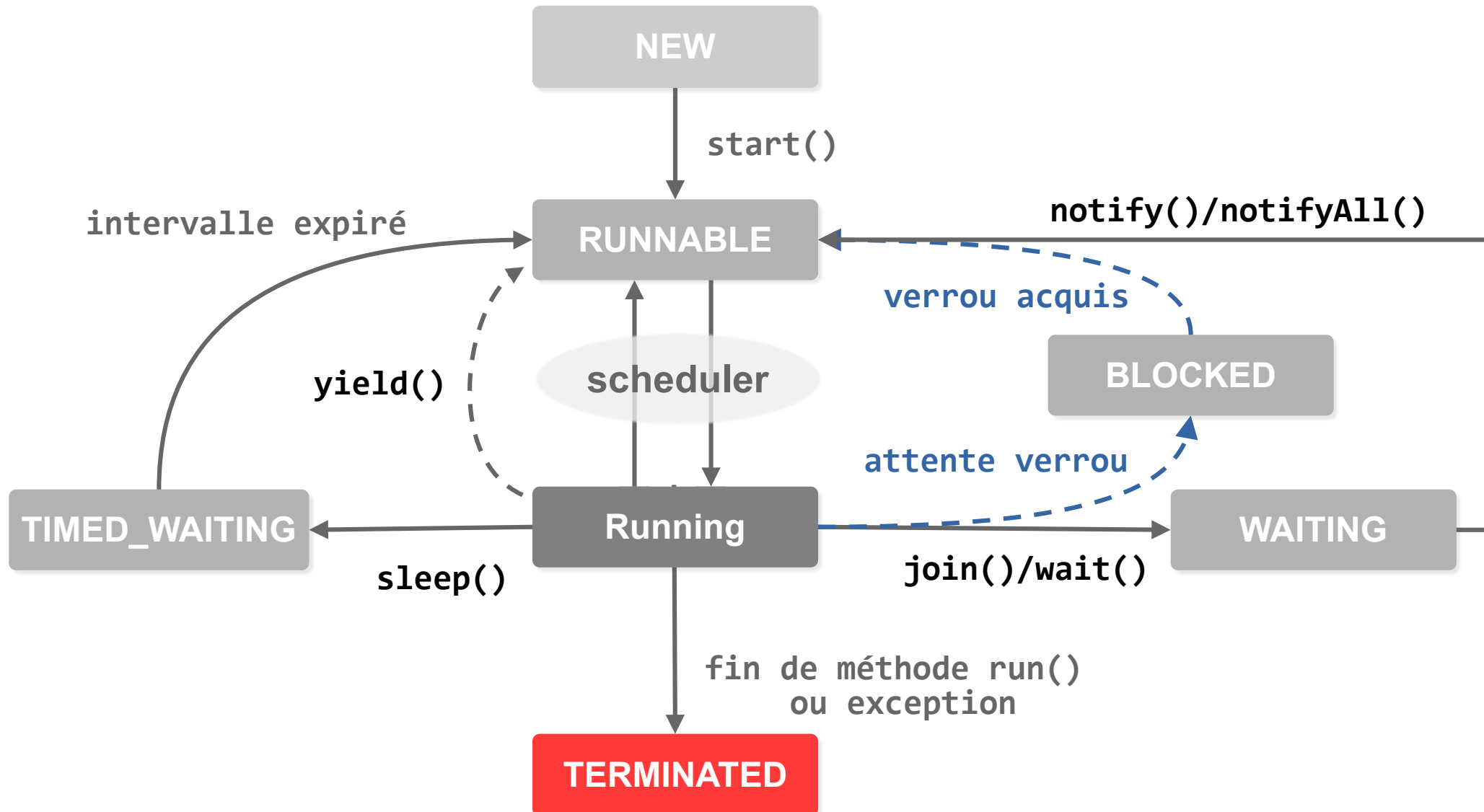
```
Runnable myRunObject = new MyRunnable ();
Thread thread = new Thread (myRunObject);
```

La classe Thread

- Représente un thread d'exécution
- La classe Thread implémente l'interface **Runnable**
- Un thread :
 - Possède un nom et une priorité
 - Peut être un daemon ou non
 - Possède plusieurs états

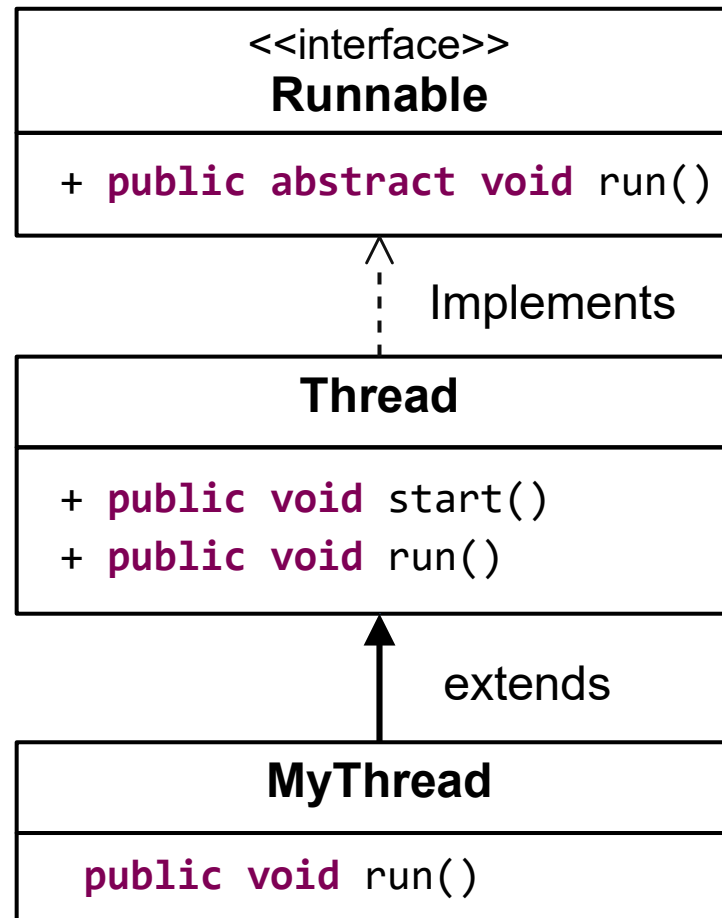


États d'un thread



Écrire ses propres threads

```
public class MyThread extends Thread {...}
```



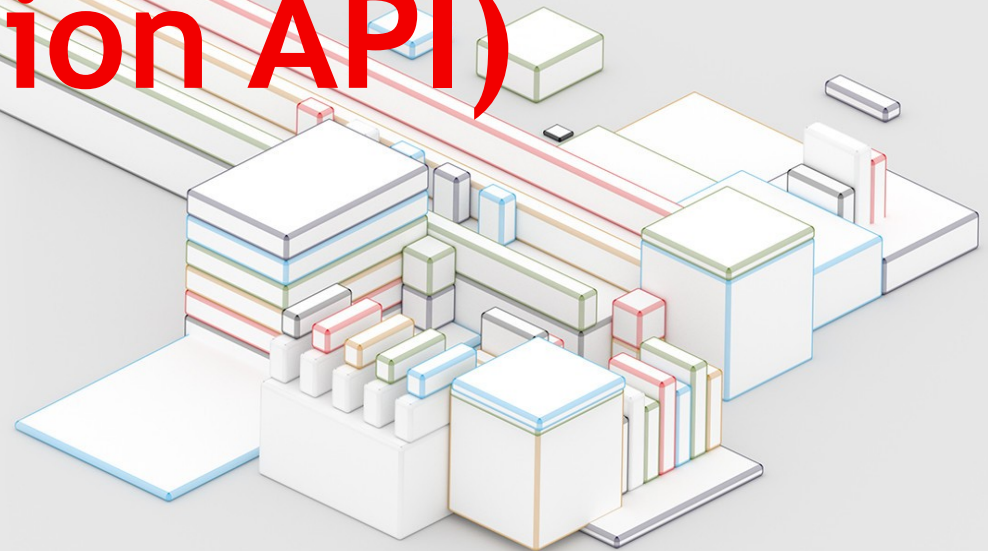
Utilisation des threads

```
MyThread thread1 = new MyThread ();  
MyThread thread2 = new MyThread ();  
thread1.start();  
thread2.start();
```

- Exemple de pause d'un thread en exécution :
RunningThread.java avec **sleep()**

```
public class RunningThread extends Thread {  
    public void run(){  
        try{  
            for (int i=0;i<3;i++){  
                Thread.sleep(1000);  
                System.out.println(getName()+" is running");  
            }  
        }catch(InterruptedException e){}  
    }  
}
```


Introspection (Reflection API)



API Reflection



- L'API Reflection permet de connaître le contenu d'une classe dynamiquement et de savoir ce que contient une classe sans en avoir les sources
- Elle est utilisée dans :
 - IDE (Eclipse, NetBeans ...)
 - debugger
 - outils de test
 - certains Frameworks (Hibernate, Spring, Struts ...)
- Inconvénients
 - réduction des performances
 - diminution de la sécurité
 - accès aux membres privés

Class

- Les instances de la classe Class sont des objets représentant les classes du langage
- On peut l'obtenir avec
 - la méthode **getClass()** de la classe Object

```
String str = "test";  
Class classStr = str.getClass();
```

- le nom de la classe avec la méthode **forName()** de Class

```
Class classStr = Class.forName("java.lang.String");
```

- **.class**

```
Class c = String.class;
```

Class

Méthode	Description
<code>Constructor[] getConstructors()</code>	retourne les constructeurs publics
<code>Constructor[] getDeclaredConstructors()</code>	retourne tous les constructeurs
<code>Field[] getFields()</code>	retourne tous les attributs publics
<code>Field[] getDeclaredFields()</code>	retourne tous les attributs
<code>Method[] getMethod()</code>	retourne toutes les méthodes public
<code>Method[] getDeclaredMethods()</code>	retourne toutes les méthodes
<code>int getModifiers()</code>	retourne un entier représentant les modificateurs de la classe (à utiliser avec Modifier)
<code>Class[] getInterfaces()</code>	retourne des interfaces implémentées par la classe
<code>Classe getSuperClass()</code>	retourne la classe mère
<code>Package getPackage()</code>	retourne le package de la classe
<code>boolean IsInterface()</code>	indique si la classe est une interface
...	

Instanciación dinámica



- **Avec la classe Class**

La méthode **newInstance()** de la classe **Class** permet de créer une instance de la classe en invoquant son constructeur par défaut

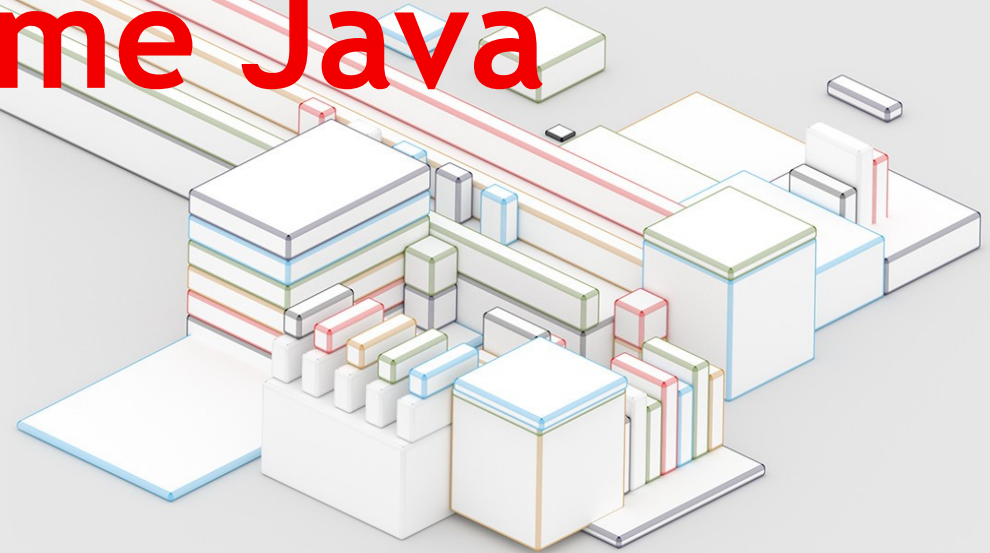
```
Class<MaClasse> classe = (Class<MaClasse>) Class.forName("NomClasse");  
MaClasse instance = classe.newInstance();
```

- **Avec la classe Constructor**

La classe **Constructor** permet de créer des instances en invoquant un constructeur quelconque d'une classe

```
Class<MaClasse> classe = (Class<MaClasse>) Class.forName("NomClasse");  
Constructor<MaClasse> constructeur = classe.getConstructor(new Class[]  
{boolean.class, Class.forName("java.lang.String")});  
MaClasse instance = constructeur.newInstance(new Object[]  
{ Boolean.FALSE, "nom instance" }); }
```

Spécificités de la plate-forme Java



- Outil permettant la génération d'une documentation technique à partir du code source et des commentaires au format :

```
/** commentaire */
```

- Les commentaires peuvent contenir :
 - des annotations : @author, @param, @return ...
 - des élément HTML pour enrichir le formatage de la documentation, généralement : , <i>, <u> et <p>
- La première phrase du commentaire doit être une courte description de la classe et elle doit se terminer par un .
- Les annotations @param et @return doivent être systématiquement indiqués (sauf pour les méthodes sans paramètres ou qui retournent void)

- **Annotations**

@author	nom du développeur
@deprecated	marque la méthode comme dépréciée
@exception ou @throws	documente une exception lancée par une méthode
@param	définit un paramètre de méthode
@return	documente la valeur de retour
@see	documente une association à une autre méthode ou classe
@since	précise à quelle version du JDK une méthode a été ajoutée à la classe
@version	donne la version d'une classe ou d'une méthode

Javadoc

```
/**
 * Calcul de la somme de 2 entiers.
 * @author Mohamed DERKAOUI
 * @param un entier a
 * @param un entier b
 * @return La somme des 2 paramètres a et b*/
public int somme(int a, int b) {
    return a+b;
}
```

- **Générer la documentation avec la commande javadoc**
javadoc a pour paramètre :
 - la liste des packages ou la liste des fichier source à commenter séparer par des espaces
 - des options ([voir documentation](#))
ex : -d permet de spécifier le dossier où seront placé les fichiers générés

Javadoc



```
> javadoc fr.dawan.formation -d doc
```

```
> javadoc fr/dawan/formation/*.java -d doc
```

- **Générer la documentation avec eclipse**

menu → project → generate Javadoc

- **Doclet**

Une doclet permet de préciser le format de la documentation générée par javadoc, par défaut de l'html

Il est possible de définir sa propre doclet pour changer le contenu ou le format de la documentation (pdf, rtf, xml ...)

Liste de doclet : <http://doclet.com/>

JAR : Java ARchive



- Un fichier JAR est un fichier ZIP qui permet de regrouper :
 - Les fichiers **.class**
 - Les images, ressources diverses
 - Les fichiers de configuration

- **Créer un fichier JAR**

```
jar cfv nomDuFichier.jar listeFichiersAIncorporer
```

Les fichiers que l'on veut inclure au jar sont séparés par des espaces

```
> jar cfv game.jar JavaProgram.class Game.class
```

- Avec eclipse :

Menu → File → Export ... → Java → JAR file
ou runnable JAR file

JAR : Java ARchive



- **Exécuter un fichier JAR**

```
java -jar nomDuFichier.jar
```

```
> java -jar game.jar
```

- **Fichier Manifest**

C'est un fichier spécial qui peut contenir des informations sur les fichiers emballés dans un JAR

Il a toujours le chemin d'accès **META-INF/MANIFEST.MF**

Les informations contenues dans le fichier manifeste prennent la forme de paires "**en-tête : valeur**". Le nom d'un en-tête est séparé de sa valeur par deux points. Le manifeste par défaut est conforme à la version 1.0 de la spécification du manifeste et a été créé par la version 1.7.0_06 du JDK.



Plus d'informations sur <http://www.dawan.fr>

**Contactez notre service commercial au
09.72.37.73.73 (prix d'un appel local)**