# NFA023/Android
# Programmation orientée objet/Plan du cours

2011-2012

## 1 Notion de structure

Insuffisance des types primitifs et des tableaux en java :
- on veut représenter des produits, qui peuvent être périmés ;
- un produit a une désignation et une date limite de péremption

Solution pour une liste de produits : deux tableaux, un de String (les désignations), l'autre d'entiers (?) pour les dates limites.

C'est difficile à manipuler : on doit toujours passer les deux tableaux. Si on décide de rajouter une caractéristique, il faut ajouter un troisième tableau et revoir tout le code.

⇒ notion de structure (dès langages de programmation des années 60) : on crée notre propre type de donnée, représentant un produit, et, dans un produit on place toutes les informations intéressantes : désignation et date limite.

En java :

```
class Produit {
        String designation;
        Date peremption;
}
```

- `designation` et `peremption` sont des *variables d'instances* (ou *attributs*). Elles sont placées en dehors des méthodes.
- la date de péremption est une date, hors Java propose un type (Date, justement) pour cela.

Quand on va créer un nouvel objet `Produit,` il aura sa propre valeur des différents attributs.

Utilisation :

```
Produit p; // on déclare un produit
p= new Produit(); // on crée un objet produit.
                  // p contient l'adresse du produit.
p.designation= "lait"; // on fixe la désignation
p.date= new Date(); // on fixe la date...
```

Pour accéder à la valeur d'une variable d'instance `designation` d'un produit p, on écrit `p.designation`.

# 2 Constructeurs

Pour mieux contrôler *l'initialisation* des objets, on introduit la notion de constructeur. Dans une ligne comme :

```
p= new Produit();
```

Il se passe deux choses : `new` réserve de l'espace pour un nouveau produit, et celui-ci est initialisé grâce au **constructeur** `Produit()`. Le constructeur qui ne prend pas d'argument est nommé *constructeur par défaut.*

Le constructeur par défaut initialise les attributs à 0, false ou `null` selon leurs types.

## 2.1 Écriture de constructeurs

On écrit généralement un ou plusieurs constructeurs pour une classe. Leur travail est de permettre l'initialisation des variables d'instances d'un nouvel objet.

Dans le cas de notre classe `Produit,` ce sera

```
class Produit {
  String designation;
  Date peremption;

  Produit(String des, Date per) {
    designation= des;
    peremption= per;
  }
}
```

- Le constructeur a le même nom que la classe, et est identifié par le *type* de ses arguments
- dans le constructeur, on a accès aux variables d'instances de l'objet qu'on est en train d'initialiser.
- son travail c'est justement de leur donner une valeur.

Utilisation :

```
// Appel du constructeur de date
// (voir la doc de Date)
Date date= new Date(2012, 6, 20);
// appel de notre nouveau constructeur.
Produit p= new Produit("lait", date);
```

## 2.2 "this"

Dans le constructeur, (et plus tard dans les méthodes), le mot-clef `this` désigne l'objet auquel s'applique le constructeur (ou la méthode). On dit parfois « l'objet courant ». On peut donc aussi écrire

```
class Produit {
  String designation;
  Date peremption;
```

```
  Produit(String des, Date per) {
    this.designation= des;
    this.peremption= per;
  }
}
```

Ça permet une petite acrobatie : dans notre constructeur, nous avons utilisé des variables `des` et *per* pour les arguments du constructeur, en inventant de nouveaux noms, parce que sinon, nous aurions écrit :

```
  Produit(String designation, Date peremption) {
    designation= designation;
    peremption= peremption;
  }
```

Qui ne fonctionne pas : les paramètres « cachent » les variables d'instance, et

```
designation= designation;
```

signifie « copier la valeur du paramètre `designation` dans ... le paramètre `designation` », ce qui ne fait strictement rien.

En revanche, on écrira :

```
  Produit(String designation, Date peremption) {
    this.designation= designation;
    this.peremption= peremption;
  }
```

Qui ne pose pas de problème :

- `this.designation` est la variable d'instance `designation` du nouvel objet ;
- `designation` tout court est le paramètre passé au constructeur.

# 3 Méthodes

En programmation objet, une classe associe :

- des données (les variables d'instance) ;
- et des comportements, les *méthodes*.
- Les méthodes décrivent ce que l'objet *sait faire.*
- En pratique, utiliser une classe revient à connaître ses constructeurs et ses méthodes.
- Une méthode est une procédure ou une fonction qui sera appliquée à un objet. On « demande à l'objet d'exécuter la méthode ».

Exemple : supposons qu'on veuille savoir si un produit est périmé. Regardons la classe `Date.`

- Son constructeur par défaut crée une date représentant « aujourd'hui » ;
- la classe Date a une méthode

Nous pouvons maintenant ajouter une méthode `estPerime()` à la classe `Produit` :

```
class Produit {
  String designation;
  Date peremption;
```

```
  Produit(String des, Date per) {
    this.designation= des;
    this.peremption= per;
  }

  boolean estPerime() {
    Date aujourdhui= new Date();
    boolean perime= aujourdhui.after(this.peremption);
    return perime;
  }
}
```

Noter que le nom de la méthode n'est pas précédé de `static`. On appelle la méthode *sur un objet* `Produit` en séparant l'objet de la méthode par un point :

```
Produit p= .... ;
if (p.estPerime()) {
  ...
}
```

Une méthode non statique s'exécute *toujours* sur un produit. La méthode a automatiquement accès aux variables d'instances de l'objet sur lequel elle s'exécute (qu'on ne passe donc pas comme paramètre). Ici, `estPerime` utilise `this.peremption.`

L'utilisation du mot clef `this` est optionnelle. On aurait pu écrire `peremption` tout court :

```
  boolean estPerime() {
    Date aujourdhui= new Date();
    boolean perime= aujourdhui.after(peremption);
    return perime;
  }
```

Une méthode très pratique : `toString()` : dès que java veut avoir une représentation textuelle d'un objet sous forme de `String`, comme par exemple dans une concaténation (ou pour l'affichage des élément dans un `ListView` sous `Android`), il appelle la méthode `toString`. On peut la redéfinir pour une classe, et elle sera alors utilisée :

```
class Produit {
  ....
  public String toString() {
    return designation + " consommer avant : " + peremption;
  }
}
```

(le mot clef `public` est expliqué ci-dessous).

# 4 Encapsulation et Visibilité

Supposons que nous écrivions notre propre classe `MaDate` pour représenter une date. Une date comporte une année, un mois, et un jour. Nous les représenterons

par trois entiers. Cependant, il y a des contraintes sur ceux-ci : le mois doit être compris entre 1 et 12, et le jour entre 1 et 31 (en réalité, le jour maximal dépend du mois et de l'année, mais nous laissons cela en exercice au lecteur intéressé).

L'une des grandes idées de la programmation est qu'une classe doit fonctionner comme une « boîte noire ». Si on veut réaliser de grands programmes, il faut cacher la complexité internes des classes, et en faire des *abstractions* signifiantes.

Par exemple, pour la classe `MaDate`, la représentation interne ne nous intéresse pas tant que ça. On veut pouvoir faire des opérations comme calculer la date du lendemain ou de la veille. Hors, ce n'est pas si simple (on peut par exemple changer de mois). Bref, on aimerait que le programmeur qui *utilise* la nouvelle classe :

- la manipule exclusivement à travers ses constructeurs et ses méthodes ;
- n'ait ni le besoin, ni même la possibilité de manipuler *directement* les variables d'instance (la plomberie interne de la classe, quoi).

On peut écrire :

```
class MaDate {
        int annee, mois, jour;
        MaDate(int a, int m, int j) {
          if (m < 1 || m > 12 || j < 1 || j > 31)
                throw new IllegalArgumentException();
          annee= a;
          mois= m;
          jour= j;
        }
}
```

Alors, toute nouvelle date sera « correcte », car le constructeur garantira que le mois sera dans [1,12] et le jour dans [1,31].

Mais rien n'interdira d'écrire :

```
MaDate fin= new MaDate(2011,12,31);
// Le lendemain, tentative "naive" :
fin.jour = fin.jour + 1;
// le 32 décembre ????
```

Car le programmeur pourra manipuler directement les variables d'instances.

Pour prévenir cela, on introduit les mots clefs `public` et `private`.

- `public` devant une variable d'instance, une méthode ou un constructeur, signifie que celui-ci est utilisable librement sur un objet de la classe.
- `private` signifie que la variable d'instance, le constructeur ou la méthode ne peut être appelée directement que par les méthodes de la classe elle-même.
- Devant une classe, `public` la rend utilisable depuis les autres *packages* du programme. En pratique, vous mettrez `public` devant la plupart des classes.

En pratique :

- une variable d'instance doit être `private` ;
- une méthode ou un constructeur est généralement `public` ;
- une méthode `private` est une méthode auxiliaire utilisée pour la « plomberie interne » de votre classe.

Exemple :

```
public class MaDate {
        private int annee, mois, jour;
        public MaDate(int a, int m, int j) {
          if (m < 1 || m > 12 || j < 1 || j > 31)
                throw new IllegalArgumentException();
          annee= a;
          mois= m;
          jour= j;
        }
}
```

## 4.1 Accesseurs

Cependant, si annee, jour et mois sont private, on ne peut plus les consulter :

```
MaDate d= new MaDate(2012,11, 10);
int a= d.annee; // NE COMPILE PAS.
```

On écrit donc des méthodes, appellées *accesseurs*, pour pouvoir consulter ou modifier les variables d'instance. Java propose des conventions pour cela :

### 4.1.1 Getters

les « getters » permettent de consulter les valeurs des variables d'instance. Conventionnellement leur nom est « `get` » + le nom de la variable. Par exemple, nous aurons `getAnnee(), getMois()` et `getJour()` (notez la majuscule après `get`).

La classe `MaDate` devient :

```
public class MaDate {
        private int annee, mois, jour;
        public MaDate(int a, int m, int j) {
          ...
        }
        public int getAnnee() {
          return annee;
        }
        public int getMois() {
          return mois;
        }
        public int getJour() {
          return jour;
        }
}
```

Il est maintenant possible d'accéder aux variables d'instance, mais pas de les modifier. Un objet de classe `MaDate,` n'ayant pas de méthode qui modifient les variables d'instances, est dit « immuable ». Une fois créé, il ne peut changer de valeur :

```
MaDate d= new MaDate(2012,11, 10);
int a= d.getAnnee(); // ok
// mais je ne peux pas modifier annee, mois ou jour !
```

Quand la valeur est booléenne, la convention est de remplacer « `get` » par « `is` ». Nota : si le nom des variables est en français, on arrive rapidement à un franglais illisible... on peut envisager de tout nommer en anglais dès le départ.

### 4.1.2 Setters

Quand on veut permettre la *modification* d'une variable d'instance, on écrit un « *setter* ». La méthode en question peut, *si nécessaire,* faire des vérifications sur la valeur à donner, mais, *dans tous les cas,* on aura intérêt à passer par ce mécanisme, afin d'avoir le code le plus uniforme possible.

Le setter est une *procédure* qui prend comme argument la nouvelle valeur de la variable d'instance. Le nom du *setter* est « `set` » + le nom de la variable.

Exemple :

```
public class Personne {
  private String nom, prenom, adresse;
  public Personne (String nom, String prenom, String adresse) {
        // exercice laissé au lecteur intéressé.
  }

  public void setAdresse(String adresse) {
        this.adresse= adresse;
  }
  public String getNom() {return nom;}
  public String getPrenom() {return prenom;}
  public String getAdresse() {return adresse;}
}
```

Dans la classe Personne, le nom et le prénom ne changent plus une fois donnés par le constructeur. La personnne peut cependant changer d'adresse. Utilisation :

```
Personnne p= new Personnne("Baggins", "Frodo", "Shire");
...
p.setAdresse("Gray Havens");
```

# 5 ArrayList

Voir une introduction aux arrayList sur le site de `NFA001` :

`http://deptinfo.cnam.fr/Enseignement/CycleA/APA/docs/chap-arraylist.pdf`