Chapitre 9

Premiers Pas en Programmation Objet : les Classes et les Objets

Dans la première partie de ce cours, nous avons appris à manipuler des objets de type simple : entiers, doubles, caractères, booléens. Nous avons aussi appris comment utiliser les tableaux pour stocker des collections d'objets de même type : tableaux d'entiers, tableaux de booléens...Cependant, la majorité des programmes manipulent des données plus complexes. Pour ecrire un logiciel bancaire, il faudra représenter dans notre langage de programmation l'ensemble des informations caractérisant un compte et coder les actions qui s'effectuent sur les comptes (retrait, dépot) ; un logiciel de bibliothèque devra représenter l'ensemble des informations caractéristiques des livres et coder les opérations d'ajout ou de retrait d'un livre ....
L'approche Orientée Objet, que nous allons aborder dans ce chapitre, consiste à rendre possible dans le langage de programmation la définition d'objets (des livres, des comptes ...) qui ressemblent à ceux du monde réel, c'est à dire caractérisés par un état et un comportement. L'état d'un compte, pourra être défini par son numéro, le nom de son titulaire, son solde ; son comportement est caractérisé par les opérations de dépot, de retrait et d'affichage du solde.

Dans nos programmes nous aurons plusieurs objets comptes. Chacuns ont un état qui leur est propre, mais ils ont les mêmes caractéristiques : ce sont tous des comptes. En programmation Orientée Objet, nous dirons que ces différents objets comptes sont des objets instances de la classe Compte. Une classe est un prototype qui définit les variables et les méthodes communes à tous les objets d'un même genre. Une classe est un *patron d'objets*. Chaque *classe* définit la façon de créer et de manipuler des *Objets* de ce type.
A l'inverse, un objet est toujours un exemplaire, une instance d'une classe (son patron).
Ainsi, pour faire de la programmation Objet, il faut savoir concevoir des classes, c'est à dire définir des modèles d'objets, et créer des objets à partir de ces classes.
Concevoir une classe, c'est définir :

1. **Les données** caractéristiques des objets de la classe. On appelle ces caractéristiques les *variables d'instance*.
2. **Les actions** que l'on peut effectuer sur les objets de la classe . Ce sont les *méthodes* qui peuvent s'invoquer sur chacuns des objets de la classe.

Ainsi, chaque objet crée possèdera :

1. **Un état**, c'est à dire des valeurs particulières pour les variables d'instances de la classe auquel il appartient.

2. **Des méthodes** qui vont agir sur son état.

9.1 Définir une Classe

Une classe qui définit un type d'objet a la structure suivante :
- Son nom est celui du type que l'on veut créer.
- Elle contient les noms et le type des caractéristiques (les variables d'instances) définissant les objets de ce type.
- Elle contient les méthodes applicables sur les objets de la classe.

Pour définir une classe pour les comptes bancaires on aurait par exemple :

Listing 9.1 – (lien vers le code brut)

```
1  class Compte {
2     int solde;
3     String titulaire;
4     int numero;
5
6     void afficher(){
7       Terminal.ecrireString("solde"+ this.solde);
8     }
9     void deposer(int montant){
10      this.solde = this.solde+ montant;
11    }
12    void retirer(int montant){
13      this.solde=this.solde-montant;
14    }
15 }
```

9.1.1 Les Variables d'instances

La déclaration d'une variable d'instance se fait comme une déclaration de variable locale au main ou à un sous programme : on donne un nom, précédé d'un type. La différence est que cette déclaration se fait **au niveau de la classe** et non à l'intérieur d'un sous programme.

Nous avons dans la classe `Compte` 3 variables d'instance : `solde` destinée à recevoir des entiers, `titulaire` destinée à recevoir une chaine de caractères et `numero` destinée à recevoir des entiers.

Ainsi l'état de chaque objet instance de la classe `compte` que nous créerons par la suite sera constitué d'une valeur pour chacune de ces trois variables d'instances. Pour chaque objet instance de la classe `Compte` nous pourrons connaitre la valeur de son solde, le nom de son titulaire et le numéro du compte. Ces valeurs seront **propres à chaque objet**.

9.1.2 Les méthodes : premier aperçu

Nous n'allons pas dans ce paragraphe décrire dans le détail la définition des méthodes d'objets, mais nous nous contentons pour l'instant des remarques suivantes : une classe définissant un type d'objets comportera autant de méthodes qu'il y a d'opérations utiles sur les objets de la classe.

La définition d'une méthode d'objet (ou d'instance) ressemble à la défintion d'un sous programme : un type de retour, un nom, une liste d'arguments précédés de leur type. Ce qui fait d'elle une méthode d'objet est qu'elle ne comporte pas le mot clé `static`. Ceci (plus le fait que les méthodes

sont dans la classe `Compte`) indique que la méthode va pouvoir être invoquée (appelée) sur n'importe quel objet de type `Compte` et modifier son état (le contenu de ses variables d'instances).

Ceci a aussi des conséquences sur le code de la méthode comme par exemple l'apparition du mot clé `this`, sur lequel nous reviendrons lorsque nous saurons comment invoquer une méthode sur un objet.

9.2 Utiliser une Classe

Une fois définie une classe d'objets, on peut utiliser le nom de la classe comme un nouveau type : déclaration de variables, d'arguments de sous programmes .... On pourra de plus appliquer sur les objets de ce type toutes les méthodes de la classe.

9.2.1 Déclarer des objets instances de la classe

Si la classe `Compte` est dans votre répertoire de travail, vous pouvez maintenant écrire une autre classe, par exemple `test` qui, dans son main, déclare une variable de type `Compte` :

Listing 9.2 – (lien vers le code brut)

```
1 public class test {
2   public static void main (String [] arguments){
3     Compte c1 = new Compte();
4   }
5 }
```

Comme pour les tableaux, une variable référencant un objet de la classe `Compte`, doit recevoir une valeur, soit par une affectation d'une valeur déjà existante, soit en créant une nouvelle valeur avec `new` avant d'être utilisée. On peut séparer la déclaration et l'initialisation en deux instructions :

Listing 9.3 – (lien vers le code brut)

```
1 public class test {
2   public static void main (String [] arguments){
3     Compte c1;
4     c1 = new Compte();
5   }
6 }
```

Après l'exécution de `c1 = new Compte();` chaque variable d'instance de `c1` a une valeur par défaut. Cette valeur est 0 pour `solde` et `numero`, et `null` pour `titulaire`.

9.2.2 Accéder et modifier les valeurs des variables d'instances d'un objet

La classe `Compte` définit la forme commune à tous les comptes. Toutes les variable de type `Compte` auront donc en commun cette forme : un solde, un titulaire et un numéro. En revanche, elles pourront représenter des comptes différents.

Accéder aux valeurs des variables d'instance

Comment connaître le solde du compte `c1` ? Ceci se fait par l'**opérateur noté par un point** :

Listing 9.4 – (lien vers le code brut)

```
1  public class test  {
2    public static void main (String [] arguments){
3      Compte c1 = new Compte();
4      Terminal.ecrireInt(c1.solde);
5    }
6  }
```

La dernière instruction à pour effet d'afficher à l'écran la valeur de la variable d'instance `solde` de `c1`, c'est à dire l'entier 0. Comme le champ `solde` est de type `int`, l'expression `c1.solde` peut s'utiliser partout où un entier est utilisable :

Listing 9.5 – (lien vers le code brut)

```
1  public class test  {
2    public static void main (String [] arguments){
3      Compte c1 = new Compte();
4      int x;
5      int []tab = {2,4,6};
6      tab[c1.solde]= 3;
7      tab[1]= c1.numero;
8      x = d1.solde +34 / (d1.numero +4);
9    }
10 }
```

Modifier les valeurs des variables d'instance

Chaque variable d'instance se comporte comme une variable. On peut donc lui affecter une nouvelle valeur :

Listing 9.6 – (lien vers le code brut)

```
1  public class test  {
2    public static void main (String [] arguments){
3      Compte c1 = new Compte();
4      Compte c2 = new Compte();
5      c1.solde =100;
6      c1.numero=218;
7      c1.titulaire="Dupont";
8      c2.solde =200;
9      c2.numero=111;
10     c2.titulaire="Durand";
11     Terminal.ecrireStringln
12     ("valeur de c1:" + c1.solde + " ," + c1.titulaire + " ," + c1.numero);
13     Terminal.ecrireStringln
14     ("valeur de c2:" + c2.solde + " ," + c2.titulaire + " ," + c2.numero);
15   }
16 }
```

c1 représente maintenant le compte numero 218 appartenant à Dupont et ayant un solde de 100 euros. et c2 le compte numero 111 appartenant à Durand et ayant un solde de 200 euros.

Affectation entre variables référençant des objets

l'affectation entre variables de types `Compte` est possible, puisqu'elles sont du même type, mais le même phenomène qu'avec les tableaux se produit : les 2 variables référencent le même objet et toute modification de l'une modifie aussi l'autre :

Listing 9.7 – (lien vers le code brut)

```
1  public class testBis  {
2    public static void main (String [] arguments){
3      Compte c1 = new Compte();
4      Compte c2 = new Compte();
5      c1.solde =100;
6      c1.numero=218;
7      c1.titulaire="Dupont";
8      c2 = c1;
9      c2.solde = 60;
10     Terminal.ecrireStringln
11     ("valeur de c1:" + c1.solde + " , "+ c1.titulaire + ", " + c1.numero);
12     Terminal.ecrireStringln
13     ("valeur de c2:" + c2.solde + " , "+ c2.titulaire + ", " + c2.numero);
14   }
15 }
```

Trace d'exécution :

```
%> java testBis
valeur de c1: 60 , Dupont, 218
valeur de c2: 60 , Dupont, 218
```

9.2.3 Invoquer les méthodes sur les objets.

Une classe contient des variables d'instances et des méthodes. Chaque objet instance de cette classe aura son propre état, c'est à dire ses propres valeurs pour les variables d'instances. On pourra aussi invoquer sur lui chaque méthode non statique de la classe. Comment invoque-t-on une méthode sur un objet ?

> Pour invoquer la méthode `afficher()` sur un objet `c1` de la classe `Compte` il faut écrire : `c1.afficher();` .

Comme l'illustre l'exemple suivant :

Listing 9.8 – (lien vers le code brut)

```
1  public class testAfficher  {
2    public static void main (String [] arguments){
3       Compte c1 = new Compte();
4       Compte c2 = new Compte();
5       c1.solde =100;
6       c1.numero=218;
7       c1.titulaire="Dupont";
8       c1.afficher();
9       c2.afficher();
10   }
11 }
```

L'expression `c1.afficher();` invoque la méthode `afficher()` sur l'objet `c1`. Cela a pour effet d'afficher à l'écran le solde de `c1` c'est à dire 200. L'expression `c2.afficherDate();` invoque la méthode `afficher()` sur l'objet `c2`. Cela affiche le solde de `c2` c'est à dire 0.

```
> java testafficher
%0 , 0 , 0
%4 , 12 , 2000
```

Ainsi, les méthodes d'objets (ou méthodes non statiques) s'utilisent par invocation sur les objets de la classe dans lesquelles elles sont définies. l'objet sur lequel on l'invoque ne fait pas partie de la liste des arguments de la méthode. Nous l'appellerons l'objet courant.

9.3 Retour sur les méthodes non statiques

Dans une classe définissant un type d'objet, on définit l'état caractéristiques des objets de la classe (les variables d'instances) et les méthodes capables d'agir sur l'état des objets (méthodes non statiques). Pour utiliser ces methodes sur un objet `x` donné, on ne met pas x dans la liste des arguments de la méthode. On utilisera la classe en déclarant des objets instances de cette classe. Sur chacun de ces objets, la notation pointée permettra d'accéder à l'état de l'objet (la valeur de ses variables d'instances) ou de lui appliquer une des méthodes de la classe dont il est une instance.
Par exemple, si `c1` est un objet instance de la classe `Compte` `c1.titulaire` permet d'accéder au titulaire de ce compte, et `c1.deposer(800)` permet d'invoquer la méthode `deposer` sur `c1`.

9.3.1 Les arguments des méthodes non statiques

Contrairement aux sous programmes statique que nous écrivions jusqu'alors, on voit que les méthodes non statiques ont un **argument d'entrée implicite**, qui ne figure pas parmi les arguments de la méthode : l'objet sur lequel elle sera appliqué, que nous avons déjà appelé *l'objet courant*.

Par exemple, la méthode `afficher` de la classe `compte` n'a aucun argument : elle n'a besoin d'aucune information supplémentaire à l'objet courant.
Une méthode d'objet peut cependant avoir des arguments. C'est le cas par exemple de `deposer` : on dépose sur un compte donné (objet courant) un certain `montant`. Ce montant est une information supplémentaire à l'objet sur lequel s'invoquera la méthode et nécessaire à la réalisation d'un dépot.

Les seuls arguments d'une méthode non statique sont les informations nécessaires à la manipulation de l'objet courant (celui sur lequel on invoquera la méthode), jamais l'objet courant lui même.

9.3.2 Le corps des méthodes non statiques

Les méthodes non statiques peuvent consulter ou modifier l'état de l'objet courant. Celui ci n'est pas nommé dans la liste des arguments. Il faut donc un moyen de désigner l'objet courant dans le corps de la méthode.
C'est le role du mot clé `this`. Il fait référence à l'objet sur lequel on invoquera la méthode. A part cela, le corps des méthodes non statiques est du code Java usuel.

Par exemple dans la définition de `afficher` :

Listing 9.9 – (lien vers le code brut)

```
1     void afficher(){
```

```
2         Terminal.ecrireString("solde"+ this.solde);
3       }
```

`this.solde` désigne la valeur de la variable d'instance `solde` de l'objet sur lequel sera invoqué la méthode.

Lors de l'exécution de `c1.afficher()`, `this` désignera `c1`, alors que lors de l'exécution de `c2.afficher()`, `this` désignera `c2`.

En fait, lorsque cela n'est pas ambigu, on peut omettre `this` et écrire simplement le nom de la méthode sans préciser sur quel objet elle est appelée. Pour la méthode `afficher` cela donne :

Listing 9.10 – (lien vers le code brut)

```
1       void afficher(){
2         Terminal.ecrireString("solde"+ solde);
3       }
```

9.3.3 Invocation de méthodes avec arguments

Lorsqu'une méthode d'objet a des arguments, on l'invoque sur un objet en lui passant des valeurs pour chacun des arguments.

Voici un exemple d'invoquation de `deposer` :

Listing 9.11 – (lien vers le code brut)

```
1  public class testDepot  {
2     public static void main (String [] arguments){
3       Compte c1 = new Compte();
4       c1.solde =100;
5       c1.numero=218;
6       c1.titulaire="Dupont";
7       c1.afficher();
8       c1.deposer(800);
9       c1.afficher();
10    }
11 }
```

9.3.4 Lorsque les méthodes modifient l'état de l'objet

La méthode `deposer` modifie l'état de l'objet courant. L'invocation de cette méthode sur un objet modifie donc l'état de cet objet. Dans notre exemple d'utilisation, le premier `c1.afficher()` affiche 100, alors que le second `c1.afficher()` affiche 900. Entre ces deux actions, l'exécution de `c1.deposer(800)` a modifié l'état de `c1`.

9.3.5 Lorsque les méthodes retournent un résultat

Les méthodes non statiques peuvent evidemment retourner des valeurs. On pourrait par exemple modifier `deposer` pour qu'en plus de modifier l'état de l'objet, elle retourne le nouveau solde en résultat :

Listing 9.12 – (lien vers le code brut)

```
1  class Compte {
2     int solde;
3     String titulaire;
4     int numero;
5
6     void afficher(){
7       Terminal.ecrireString("solde"+ this.solde);
8     }
9     int depot(int montant){
10      this.solde = this.solde+ montant;
11      return this.solde;
12    }
13 }
```

Maintenant, `deposer` fait 2 choses : tout d'abord elle modifie l'état de l'objet courant, puis elle retourne l'entier correspondant au nouveau solde. On peut donc utiliser son invocation sur un objet comme n'importe quelle expression de type `int`.

Par exemple

Listing 9.13 – (lien vers le code brut)

```
1 public class testDepot  {
2    public static void main (String [] arguments){
3     Compte c1 = new Compte();
4     c1.solde =100;
5     c1.numero=218;
6     c1.titulaire="Dupont";
7     Terminal.ecrireIntln(c1.deposer(800));
8    }
9 }
```

9.4 Les types des variables d'instances peuvent être des classes

Lorsqu'on définit un classe, on peut choisir comme type pour les variables d'instances n'importe quel type existant.

On peut par exemple définir le classe `Personne` par la donnée des deux variables d'instances, l'une contenant la date de naissance de la personne et l'autre son nom. La date de naissance n'est pas un type prédéfini. Il faut donc aussi définir une classe `Date`

Listing 9.14 – (lien vers le code brut)

```
1  public class Date {
2     int jour;
3     int mois;
4     int annee;
5     public  void afficherDate(){
6         Terminal.ecrireStringln(
7           this.jour + " , " + this.mois + " , " + this.annee);
8     }
9     // On ne montre pas les autres methodes ...
10 }
```

Listing 9.15 – (pas de lien)

```
1  public class Personne{
2     Date naissance;
3     String nom;
4  // on ne montre pas les methodes...
5  }
```

Lorsqu'on déclare et initialise une variable `p2` de type `Personne`, en faisant :
`Personne p2 = new Personne();` l'opérateur `new` donne des valeurs par défaut à `nom` et `naissance`. Mais la valeur par défaut pour les objets est `null`, ce qui signifie que la variable `p2.naissance` n'est pas initialisée.
Dès lors, si vous faites `p2.naissance.jour = 18;` l'éxecution provoquera la levée d'une exception :

```
> java DateTest2
Exception in thread "main" java.lang.NullPointerException
        at DateTest2.main(DateTest2.java:99)
```

Il faut donc aussi initialiser p2.naissance :
`p2.naissance= new Date();` avant d'accéder aux champs jour mois et annee.

On peut alors descendre progressivement à partir d'une valeur de type `Personne` vers les valeurs des champs définissant le champ `Date`. Si par exemple `p1` est de type `Personne` alors :

1. p1.nom est de type `String` : c'est son nom.
2. p1.naissance est de type `Date` : c'est sa date de naissance.
3. p1.naissance.jour est de type `int` : c'est son jour de naissance.
4. p1.naissance.mois est de type `int` : c'est son mois de naissance.
5. p1.naissance.annee est de type `int` : c'est son annee de naissance.

Le champ `naissance` d'une personne n'est manipulable que via l'opérateur **.**, l'affectation et les méthodes définies pour les dates.

Listing 9.16 – (lien vers le code brut)

```
1  public class PersonneTest  {
2    public static void main (String [] arguments){
3      Personne p2 = new Personne();
4      p2.nom="toto";
5      p2.naissance= new Date();
6      p2.naissance.jour = 18;
7      Terminal.ecrireIntln(p2.naissance.jour);
8      Terminal.ecrireString ( p2.nom + "  date naissance:");
9      p2.naissance.AfficherDate();
10   }
11 }
```

9.5 Les classes d'objets peuvent aussi avoir des méthodes statiques

Lorsqu'on définit une classe caractérisant un ensemble d'objets, on définit des variables d'instance et des méthodes non statiques. Ces méthodes définissent un comportement de l'objet courant.

Mais les classe peuvent aussi avoir des méthodes statiques, identiques à celles que nous avons définies dans les chapitres précédents.
Imaginons par exemple que nous voulions définir dans la classe `Date` les méthodes `bissextile` et `longueurMois`. Rien ne nous empêche de les définir comme des méthodes statiques.

Listing 9.17 – (lien vers le code brut)

```
1  public class Date  {
2      int jour;
3      int mois;
4      int annee;
5      public  void afficherDate(){
6          Terminal.ecrireStringln(
7            this.jour + " , " + this.mois + " , " + this.annee);
8      }
9      public static boolean bissextile(int a ){
10       return ((a%4==0) && (!(a%100 ==0) || a%400==0));}
11
12     public static int longueur(int m , int a ){
13       if (m == 1 || m== 3 || m==5 ||m== 7  || m==8|| m==10 || m== 12)
14         {return 31;}
15       else if (m==2) {if ( bissextile(a)){return 29;} else {return 28;}}
16       else {return 30;}
17     }
18 }
```

Déclarer une méthode statique signifie que cette méthode n'agit pas, ni ne connaît en aucune manière, l'objet courant. Autrement dit, c'est une méthode qui n'agit que sur ses arguments et sur sa valeur de retour. Techniquement, cela signifie qu'une telle méthode ne peut jamais, dans son corps, faire référence à l'objet courant, ni aux valeurs de ses variables d'instance.

De la même façon, on ne les invoque pas avec des noms d'objets, mais avec des noms de classe. Pour `longueur`, comme elle est dans la classe `Date`, il faudra écrire :

```
Date.longueur(12,2000);
```

Dans la classe `Date`, on pourra utiliser ces méthodes dans la définition d'autres méthodes, sans faire référence à la classe `Date`.

Dans la pratique, on écrit peu de méthodes statiques et beaucoup de méthodes non statiques.

Méthodes statiques et non statiques :

- Les méthodes non statiques définissent un comportement de l'objet courant auquel elles font, dans leur corps, référence au moyen de `this` :

```
public  void afficherDate(){
 Terminal.ecrireStringln(
   this.jour + " , " + this.mois + " , " + this.annee);
}
```

  On les appelle avec des noms d'objets : `d2.afficherDate();`
- Les méthodes statiques ne connaissent pas l'objet courant. Elles ne peuvent faire référence à `this`, ni aux variables d'instance.

```
public static boolean bissextile(int a ){
  return ((a%4==0) && (!(a%100 ==0) || a%400==0));}
```

  On les appelle avec un nom de classe : `Date.longueur(12,2000);`

Maintenant que nous avons `bissextile` et `longueur`, nous pouvons définir dans `Date` la

méthode `passerAuLendemain`. C'est une méthode d'objet, sans argument qui a pour effet de modifier l'état de l'objet courant en le faisant passer à la date du lendemain :

Listing 9.18 – (lien vers le code brut)

```
1  public class Date  {
2      int jour;
3      int mois;
4      int annee;
5
6      public static boolean bissextile(int a ){
7        return ((a%4==0) && (!(a%100 ==0) || a%400==0));}
8
9      public static int longueur(int m , int a ){
10       if (m == 1 || m== 3 || m==5 ||m== 7  || m==8|| m==10 || m== 12)
11         {return 31;}
12       else if (m==2) {if ( bissextile(a)){return 29;} else {return 28;}}
13       else {return 30;}
14     }
15
16     public  void passerAuLendemain(){
17       if (this.jour < longueur(this.mois,this.annee)){
18         this.jour = this.jour +1;
19       }
20       else if (this.mois == 12) {
21          this.jour = 1; this.annee=this.annee +1 ;this.mois =1;
22       }
23       else {
24           this.jour = 1 ;this.mois =this.mois+1;
25       }
26     }
27 }
```

Listing 9.19 – (lien vers le code brut)

```
1  public class Datetest3  {
2    public static void main (String [] arguments){
3      Date d2  = new Date();
4      d2.annee=2000;
5      d2.jour =28;
6      d2.mois =2;
7      d2.afficherDate();
8      d2.passerAuLendemain();
9      d2.afficherDate();
10   }
11 }
```

Et voici ce que cela donne lors de l'exécution :

```
> java Datetest3
28 , 2 , 2000
29 , 2 , 2000
```

9.6 Les constructeurs.

Revenons un instant sur la création d'objets.
Que se passe-t-il lorsque nous écrivons : `Date d1= new Date()` ?
L'opérateur `new` réserve l'espace mémoire nécessaire pour l'objet `d1` et initialise les données avec des valeurs par défaut. Une variable d'instance de type `int` recevra 0, une de type `boolean` recevra `false`, une de type `char` recevra `'\0'` et les variables d'instances d'un type objet recevra `null`. Nous expliquerons ce qu'est `null` dans le prochain chapitre. l'opérateur `new` réalise cela en s'aidant d'un *constructeur* de la class `Date`. En effet, lorsqu'on écrit `Date()`, on appelle une sorte de méthode qui porte le même nom que la classe. Dans notre exemple, on voit que le constructeur `Date` qui est appelé n'a pas d'arguments.

Un constructeur ressemble à une méthode qui porterai le même nom que la classe, mais ce n'est pas une méthode : la seule façon de l'invoquer consiste à employer le mot clé `new` suivi de son nom, c'est à dire du nom de la classe[1]. Ceci signifie qu'il s'exécute avant toute autre action sur l'objet, lors de la création de l'objet.

9.6.1 Le constructeur par défaut

La classe `Date` ne contient pas explicitement de définition de constructeur. Et pourtant, nous pouvons, lors de la création d'un objet, y faire référence après l'opérateur `new`. Cela signifie que Java fournit pour chaque classe définie, un constructeur par défaut. Ce constructeur par défaut :

– a le même nom que la classe et
– n'a pas d'argument.

Le constructeur par défaut ne fait pratiquement rien. Voilà a quoi il pourrait ressembler :

```
public Date(){

}
```

9.6.2 Définir ses propres constructeurs

Le point intéressant avec les constructeurs est que nous pouvons les définir nous mêmes. Nous avons ainsi un moyen d'intervenir au milieu de `new`, d'intervenir lors de la création des objets, donc avant toute autre action sur l'objet.

Autrement dit, en écrivons nos propres constructeurs, nous avons le moyen, en tant que concepteur d'une classe, d'intervenir pour préparer l'objet à être utilisé, avant toute autre personne utilisatrice des objets.

Sans définir nos propres constructeurs de Date, les objets de types `Date` commencent mal leur vie : il naissent avec 0,0,0 qui ne représente pas une date correcte. En effet, 0 ne correspond pas à un jour, ni à un mois. C'est pourquoi une bonne conception de la classe `Date` comportera des définitions de constructeurs.

attention : Dès que nous définissons un constructeur pour une classe, le constructeur par défaut n'existe plus.

Un constructeur se définit comme une méthode sauf que :

1. Le nom d'un constructeur est toujours celui de la classe.
2. Un constructeur n'a jamais de type de retour.

[1] En fait, on peut aussi l'invoquer dans des constructeurs d'autres classes (cf chapitre sur l'héritage)

Dans une classe, on peut définir autant de constructeurs que l'on veut, du moment qu'ils se différencient par leur nombre (ou le type) d'arguments . Autrement dit, on peut surcharger les constructeurs.

Nous pourrions par exemple, pour notre classe `Date`, définir un constructeur sans arguments qui initialise les dates à 1,1,1 (qui est une date correcte) :

Listing 9.20 – (lien vers le code brut)

```
1  public class Date {
2    int jour;
3    int mois;
4    int annee;
5
6    public Date(){
7      this.jour =1;
8      this.mois=1;
9      this.annee =1;
10     }
11   // ....
12 }
```

Maintenant, toute invocation de `new Date()` exécutera ce constructeur.

Il peut aussi être utile de définir un constructeur qui initialise une date avec des valeurs données. Pour cela, il suffit d'écrire un constructeur avec 3 arguments qui seront les valeurs respectives des champs. Si les valeurs d'entrée ne représentent pas une date correcte, nous levons une erreur :

Listing 9.21 – (lien vers le code brut)

```
1    public class Date {
2      // --- Les variables d'instances ---
3      int jour;
4      int mois;
5      int annee;
6
7      // --- Les constructeurs ---
8      public Date(){
9        this.jour =1;
10       this.mois=1;
11       this.annee =1;
12     }
13
14     public  Date (int j, int m, int a){
15       if (m >0 && m<13 && j <=longueur(m,a)){
16         this.jour=j;
17         this.mois = m;
18         this.annee = a;
19       }
20       else {
21         throw new ErreurDate();
22       }
23     }
24     // --- Les methodes ---
25     public  void afficherDate(){
26       Terminal.ecrireStringln
```

```
27        (this.jour + "␣,␣" + this.mois + "␣,␣" + this.annee);
28      }
29
30      public int getAnnee(){
31        return this.annee;
32      }
33
34      public void setAnnee(int aa){
35        this.annee=aa;
36      }
37
38      public  void passerAuLendemain(){
39        if (this.jour < longueur(this.mois,this.annee)){
40          this.jour = this.jour +1;
41        }
42        else if (this.mois == 12) {
43          this.jour = 1; this.annee=this.annee +1 ;this.mois =1;
44        }
45        else {
46          this.jour = 1 ;this.mois =this.mois+1;
47        }
48      }
49
50      public  static boolean bissextile(int annee ){
51        return ((annee%4==0) && (!(annee%100 ==0) || annee%400==0));
52      }
53
54      public static int longueur(int m , int a ){
55        if (m == 1 || m== 3 || m==5 ||m== 7  || m==8|| m==10 || m== 12){
56          return 31;
57        }
58        else if (m==2) {if ( bissextile(a)){return 29;} else {return 28;}}
59        else {return 30;}
60      }
61    }
62    class ErreurDate extends Error{}
```

Listing 9.22 – (lien vers le code brut)

```
1     public class datetest4  {
2       public static void main (String [] arguments){
3         Date  d1  = new Date ();
4         Date  d2  = new Date (2,12,2000);
5         d1.afficherDate();
6         d2.afficherDate();
7         d2.passerAuLendemain();
8         d2.afficherDate();
```

l'éxécution de `datetest4` donne :

```
> java datetest4
  1 , 1 , 1
  2 , 12 , 2000
```

```
3 , 12 , 2000
```

Répetons encore une fois que lorsqu'on définit ses propres constructeurs, le constructeur par défaut n'existe plus. En conséquence, si vous écrivez des constructeurs qui ont des arguments, et que vous voulez un constructeur sans argument, vous devez l'ecrire vous même.

- Un constructeur :
  - est un code qui s'éxécute au moment de la création de l'objet (avec `new`),
  - porte le même nom que la classe,
  - ne peut pas avoir de type de retour,
  - peut avoir ou ne pas avoir d'arguments,
  - sert à initialiser l'état de chaque objet crée.
- Si on ne définit aucun constructeur, il en existe un par défaut, qui n'a pas d'argument.
- Le constructeur par défaut n'existe plus dès que l'on définit un constructeur même si on ne définit pas de constructeur sans argument.
- 2 constructeurs doivent avoir des listes d'arguments différentes.

9.7 Résumé

- Une classe est un patron d'objet qui définit :
  1. Les données caractéristiques des objets : les variables d'instances.
  2. Les constructeurs permettant d'initialiser les objets lors de leur création. (facultatif)
  3. le comportement des objets : les méthodes que l'on pourra invoquer sur les objets.
- Un objet est une instance d'une classe.
  1. On les créé en faisant `new` suivi d'un appel de constructeur : `Date d1 = new Date();` `Date d2 = new Date(12, 6,2003);`
  2. Ils ont leur propres valeurs pour chacune des variables d'instances.
  3. On peut leur appliquer les méthodes de la classe dont ils sont l'instance : `d1.afficherDate();` .
- Les méthodes non statiques définissent un comportement de l'objet courant auquel elles font, dans leur corps, référence au moyen de `this` :

```
public  void afficherDate(){
 Terminal.ecrireStringln(
   this.jour + " , " + this.mois + " , " + this.annee);
}
```

  On les appelle avec des noms d'objets : `d2.afficherDate();`
- Les méthodes statiques ne connaissent pas l'objet courant. Elles ne peuvent faire référence à `this`, ni aux variables d'instances.

```
public static boolean bissextile(int a ){
  return ((a%4==0) && (!(a%100 ==0) || a%400==0));}
```

  On les appelle avec un nom de classe : `Date.longueur(12,2000);`
- Un constructeur :
  - est un code qui s'éxécute au moment de la création de l'objet (avec `new`)
  - porte le même nom que la classe,
  - ne peut pas avoir de type de retour,

  - peut avoir ou ne pas avoir d'argument,
  - sert à initialiser l'état de chaque objet crée.
- Si on ne définit aucun constructeur, il en existe un par défaut, qui n'a pas d'arguments.
- Le constructeur par défaut n'existe plus dès que l'on définit un constructeur.
- 2 constructeurs doivent avoir des listes d'arguments différentes (par le nombre et/ou par le type des arguments).