Chapitre 2

La programmation orientée objet est une technique très puissante permettant d'exploiter d'énormes ressources stockées sous forme de classes et accessibles librement sur Internet. L'appellation de *programmation* peut ici être remplacé par *développement des applications* basé sur la création et l'intégration des classes.

Dans ce chapitre nous présentons comment développer les classes et créer les objets.

Classe et Objet

Une classe est une collection de données et de méthodes appliquées à ces données.

Prenons comme exemple un disque et définissons une classe *Disque*:

```
public class Disque{
public double x,y;
public double r;
public double circonference() { return r*2*3.14159;}
public double surface() { return r*R*3.14159;}
}
```

Par la définition:

```
Disque d;
```

nous préparons un nom de référence mais l'objet lui même reste à créer dynamiquement par l'opérateur *new*:

```
d = new Disque();  // creation d'une instance dynamique de la classe Disque
 // initialisation des variables public
d.x = 2.0;
d.y = 4.0;
d.r = 1.0;
```

Ensuite la méthode *circonference* peut être appliquée à l'objet *d*:

```
d = new Disque();
double c;
d.r = 1.0;
c = d.circonference();
```

Remarquons que la méthode n'a pas d'argument; elle est appliquée à **cet** (***this***) **objet**!!!

Les données dans une classe peuvent être initialisées au moment de l' instantiation. A cette fin, nous introduisons la méthode de "construction" dit constructeur. Le nom du constructeur doit être le même que celui de la classe.

```
public class Disque{
public double x,y;
public double r;
```

```
public Disque(double x, double y, double r) {        // méthode de constructeur
  this.x=x;
  this.y=y;
  this.r=r;
  }
public double circonference() { return r*2*3.14159;}
public double surface() { return r*r*3.14159;}
}
```

Grâce au constructeur nous pouvons instancier l'objet *Disque* et initialiser ses variables:

```
Disque d = new Disque(2.0, 4.5, 2.0);
```

En général, une classe peut être "équipée" de plusieurs constructeurs permettant de l'instancier de façons différentes. Les méthodes de construction sont gérées par le sur-chargement.

```
public class Disque{
public double x,y;
public double r;
public Disque(double  r) {        // methode de constructeur  avec un argument
  this.x=0.0;  this.y=0.0;
  this.r=r;   }
public Disque(double x, double y, double r) {        // methode de constructeur avec trois arguments
  this.x=x;  this.y=y;
  this.r=r;  }
public double circonference() { return r*2*3.14159;}
public double surface() { return r*r*3.14159;}
}
```

Variables et constantes de classe

En langage Java, toutes les variables doivent être déclarées dans les classes. L'introduction du terme *static* dans la définition d'une classe permet d'attacher une valeur à cette classe et pour l'ensemble des objets instanciés à partir de cette classe. On peut parler ici d'une variable de classe.

```
public class Disque{
public double x,y;
public double r;
static int nombre=0;
public Disque(double x, double y, double r) {    this.x=x;  this.y=y;  this.r=r;
  nombre++;     //variable incrementee pour chaque nouvelle instance
  }
public double circonference() { return r*2*3.14159;}
public double surface() { return r*R*3.14159;}
}
```

Les variables de classe sont initialisées quand la classe est chargée pour la première fois. Les variables d'instance sont initialisées quand un objet est créé.

Une variable de classe peut être modifiée par ces instances. Pour créer une constante à exploiter dans toutes les instances il faut utiliser le terme *final*.

```
public class Disque{
public double x,y;
public double r;
static int nombre=0;
public static final double pi=3.14159;
public Disque(double x, double y, double r) {
  this.x=x;  this.y=y;  this.r=r;
  nombre++;     //variable incrémentée pour chaque nouvelle instance
  }
public double circonference() { return r*2*pi;}
public double surface() { return r*r*pi;}
}
```

Remarque: Les variables *static final* remplacent les constantes définies par la directive *#define* en langage C.

Méthodes statiques

Dans certaines applications, nous avons besoin de méthodes externes définies indépendamment des classes utilisées dans notre programme. Par exemple, nous voulons effectuer un calcul permettant de décider (oui ou non) si le point donné tombe à l'intérieur ou à l'extérieur d'un *Disque*.

```
public class Disque{
public double x,y;
public double r;
static int nombre=0;
public static final double pi=3.14159;
public Disque(double x, double y, double r) {
  this.x=x;  this.y=y;  this.r=r;
  nombre++;     //variable incrementee pour chaque nouvelle instance
  }
public double circonference() { return r*2*pi;}
public double surface() { return r*r*pi;}
public boolean siDedans(double xp, double yp)
  {
  double dx = xp - x;
  double dy = yp - y;
  double distance = Math.sqrt(dx*dx - dy*dy);
  if (distance>r) return false; else reurn true;
}
}
```

Math n'est pas une classe définie dans notre application. La méthode *sqrt()* est une méthode statique définie dans la classe *Math*:

```
public class Math {
public static sqrt(x) { .. }
..}
```

Dans la classe suivante, nous avons défini deux méthodes *pluspetit()*, une comme méthode de classe et une comme méthode d'instance.

```
public class Disque {
public double x,y;
public double r;
public static Disque pluspetit(Disque a, Disque b) {          // les objets a et b sont passes en parametre
  if (a.r<b.r) return a; else return b;
  }
public  Disque pluspetit(Disque a) {                          // l'objet a est passe en parametre
  if (a.r<r) return a; else return this;
  }
public double circonference() { return r*2*pi;}
public double surface() { return r*r*pi;}
}
```

La **méthode de classe** peut être évoquée par :

```
Disque d = Disque.pluspetit(a,b);          // methode statique (de classe Disque) avec arguments a et b
```

La **méthode d'instance** peut être évoquée par:

```
Disque d = a.pluspetit(b);                 // methode d'istance applique a l'objet a avec l'argument b
```

Organisation des classes

Les classes sont regroupées dans les ensembles appelés packages. Par défaut, toutes les classes regroupées dans un package sont visibles entre elles. Les classes déclarées comme *public* sont accessibles à partir d'autres packages. Les classes sans cette déclaration ne sont pas accessibles d'un autre package.

Les variables ou méthodes sont accessibles aux autres classes dans le même package. Les variables ou méthodes déclarées comme *private* sont accessibles seulement à l'intérieur de la classe. Les variables et les méthodes déclarées comme *public* peuvent être accessibles à partir d'autres packages à condition que leurs classes soient déclarées également en *public*.

A l'intérieur des méthodes, Java permet l'utilisation de variables locales. Ces variables se comportent comme les variables locales en langage C; elles sont inaccessibles de l'extérieur des méthodes.

Héritage

Une classe peut être étendue afin d'intégrer d'autres variables et méthodes dont nous avons besoin. Java permet de définir une nouvelle classe comme une extension, ou une sous-classe de la classe de

base. Par exemple, nous pouvons ajouter à notre classe *Disque* une nouvelle méthode *Dessiner* en créant une nouvelle classe *DisqueDessin.*

```
import java.awt.*;                                    // awt est un package graphique
public class DisqueDessin extends Disque {
Color bordure,remplissage;
public void Dessiner (DrawWindow df) {
  df.drawCircle(x,y,r,bordure,remplissage)
  }
}
```

Le terme *extends* indique que la classe *DisqueDessin* est une sous-classe de *Disque* et qu'elle hérite des variables et des méthodes de cette classe.

La nouvelle classe dérivée peut être complétée par un constructeur:

```
import java.awt.*;                // package graphique
public class DisqueDessin extends Disque {
Color bordure,remplissage;
public  DisqueDessin (double x, double y, double r, Color bordure, Color replissage) {
this.x = x;
this.y = y;
this.r = r;
this.bordure = bordure;
this.remplissage = remplissage;
}
public void Dessiner (DrawWindow df) {
```

Ce constructeur peut être simplifié moyennant la notion de **super classe** qui est la classe d'origine:

```
public  DisqueDessin(double x, double y, double r, Color bordure, Color replissage) {
  super(x,y,r);   // initialisation de la super classe
  this.bordure = bordure;
  this.remplissage = remplissage;
  }
}
```

L'appel à la super classe doit être placé avant la première instruction et/ou variable du constructeur.

Remarque: Dans le cas ou une classe est déclarée sans son constructeur, le compilateur Java ajoute un constructeur par défaut avec l'appel *super()*.

Variables cachées

Prenons comme exemple notre classe étendue *DisqueDessin* en lui ajoutant une nouvelle méthode *Resolution*. Le même nom de variable peut être utilisé

```
import java.awt.*;
public class DisqueDessin extends Disque {
Color bordure,remplissage;
float r;  // variable d'instance de resolution
public  DisqueDessin (double x, double y, double r, Color bordure, Color remplissage) {
super(x,y,r);  this.bordure = b; this.remplissage = remplissage;  }
public void Resolution(float resolution) { r = resolution; }
public void Dessiner (DrawWindow df) { df.drawCircle(x,y,r,bordure,remplissage)
}
```

Le nom de la variable d'instance de résolution et de la variable de classe pour le rayon est le même - *r*. Pour y accéder nous utilisons deux références différentes:

```
this.r                  // pour la variable d'instance
super.r                 // pour la variable de classe
```

Surcharge et coercition des méthodes

La surcharge (*overloading*) de méthode permet l'usage du même nom pour plusieurs méthodes, si leurs listes de paramètres diffèrent selon le nombre, les types et l'ordre de leurs paramètres. Lors de l'appel d'une méthode surchargée, le compilateur Java sélectionne la méthode appropriée en examinant les listes de paramètres.

```
import java.awt.*;
public class DisqueDessin extends Disque {
Color bordure,remplissage;
float sr;                       // variable d'instance de faible resolution
double dr;                      // variable d'instance de forte resolution
public  DisqueDessin (double x, double y, double r, Color bordure, Color remplissage) {
  super(x,y,r);  this.bordure = b; this.remplissage = remplissage;
 }
public void Resolution(float resolution) { sr = resolution; }
public void Resolution(double resolution) { dr = resolution; }
}
```

La coercition (*overriding*) de méthode permet de remplacer une méthode "supérieure" par une méthode dérivée. Par exemple:

```
classe Positive {
int i = 5;
int m() { return i;}
}
```

```
classe Negative extends Positive {              // methode derivee
int i = 10;
int m() { return -i;}   // “coerce” la méthode m() dans la classe Positive
}

public class Coercition {
public static void main(String args[]) {
Negative n = new Negative();
System.out.println(Negative.n);      // imprime 10
System.out.println(Negative.m());   // imprime -10

Positive p = (Positive) n;              // coercition de l’instance n en instance “supérieure” de la classe Positive
System.out.println(Positive.p);       // imprime 5
System.out.println(Positive.m());    // toujours en reference de Negative : imprime  -10
```

Visibilité des méthodes et variables

Java possède plusieurs modificateurs d’accessibilité permettant d’ouvrir ou de restreindre l’accès aux méthodes et aux variables. Par défaut, une méthode peut être accessible à l’intérieur d’un package et inaccessible en dehors de son package.

Si-dessous nous présentons la signification des modificateurs d’accessibilité:

- *public*
- *protected*
- *private*

TABLE 2. **Visibilité des variables et des méthodes Java**

	par défaut	***public***	***protected***	***private protected***	***private***
accessible à une classe dans le même package	**oui**	**oui**	**oui**	**non**	**non**
accessible à une sous-classe dans le même package	**oui**	**oui**	**oui**	**non**	**non**
accessible à une classe dans un autre package	**non**	**oui**	**non**	**non**	**non**
accessible à une sous-classe dans un autre package	**non**	**oui**	**non**	**non**	**non**
héritée par une sous-classe dans le même package	**oui**	**oui**	**oui**	**oui**	**non**
héritée par une sous-classe dans un autre package	**non**	**oui**	**oui**	**oui**	**non**

Remarquons qu'une variable ou une méthode marquée par *private* est accessible seulement dans sa classe. Les variables et les méthodes privées ne sont pas visibles dans les sous-classes et elles ne son pas héritées dans sous-classes.

Une variable et/ou une méthode protégée (*protected*) est visible pour les sous-classes et les classes dans le même package.

Tableaux (*arrays*) et *Strings*

Les tableaux

Les tableaux et les *strings* (chaînes) sont manipulés par référence. Ils sont crées dynamiquement par l'instruction *new* et ils sont effacés automatiquement quand ils ne sont plus utilisés.

Attention: Dans certaines situations, l'effacement automatique d'un tableau ou d'une partie du tableau peut provoquer le mauvais fonctionnement du programme qui n'a pas re-alloué l'espace nécessaire.

Il y a deux manières de créer un tableau; la première utilise l'instruction *new*,

```
byte tampon[] = new byte[512];  // initialises avec null
int valeurs[] = new int[32];    // initialises avec 0
object objets[] = new object[4];  // initialises avec null
byte tab2dim[][] = new byte[16][32];    // tableau bidimensionnel
```

la deuxième crée un tableau par l'initialisation statique.

```
int tabint[] = {1,2,3,4,5,6,7,8,9,0}:
```

L'accès à un élément du tableau est effectué par l'insertion de son indice entre les crochets [indice]:

```
int valeurs[] = new int[32];    // initialises avec 0
for(int i; i<valeurs.length;i++)   valeurs[i]  = i;
```

L'attribut *length* permet d'extraire la taille du tableau. Dans le cas où l'indice proposé dans le tableau dépasse la taille du tableau, l'exception du type *ArrayIndexOutOfBoundsException* est levée.

Les Strings

Les *Strings* du Java ne sont pas terminées par '*\0*'; elles sont des instances de la classe *java.lang.String*. Les *Strings* sont traitées comme objets constants; il n'est pas possible de modifier le contenu d'une *String*.

Une *String* peut être est initialisée de façon statique ou dynamique par l'instruction *new*:

```
String chaine = "Ceci est un test";
String[] titres ={"resistance", "capacité"};
String donnee = new String(paquet.getData());      // la taille de donnees dans le paquet doit etre connue
int taille = donnee.length();
```

La gestion de *String* peut être effectuée par plusieurs méthodes proposées dans le package *java.lang*. Par exemple, la taille d'une *String* est donnée par la méthode *length()*.

Classes et interfaces abstraites

Java permet de définir des classes abstraites. Une classe abstraite peut être considérée comme une classe générique (super-classe). Par exemple nous pouvons concevoir une classe *Figures* dans laquelle nous allons mettre progressivement notre classe *Disque*, puis une classe *Rectangle*, *Triangle*, etc. Toutes ces classes peuvent contenir les méthodes pour calculer la *circonference()* et la *surface()*.

```
public abstract  class Forme {
public abstract double circonference();              // methode abstraite
public abstract double surface();
}
public class Disque extends Forme{
protected final double pi= 3.14159;
public double x,y, r;
public Disque(double x, double y, double r) {  this.x=x;  this.y=y;  this.r=r;  }
public double circonference() { return r*2*pi;}
public double surface() { return r*r*pi;}
}
public class Rectangle extends Forme{
public double x,y;
public Rectangle(double x, double y) {  this.x=x;  this.y=y;  }
public double circonference() { return 2*(x+y);}
public double surface() { return x*y;}
}
```

Les méthodes déclarées dans la classe abstraite *Forme* sont des méthodes abstraites. Remarquons qu'une méthode abstraite n'a pas du corps encadré par {}.

Une classe qui contient une méthode abstraite devient automatiquement une classe abstraite. Une classe abstraite ne peut pas être instanciée. Une sous-classe qui n'implémente pas toutes les méthodes de classe abstraite reste également une classe abstraite.

Une classe abstraite peut être utilisée pour instancier les sous-classes.

```
Forme[] formes= new Forme[2];              // tableau pour futures instances, pas une instanciation !!!
formes[1] = new Rectangle(5.2,8.4);
formes[2] = new Disque(2.5,5.0,1.0);
double surface_totale = 0.0;
for(int i=0;i<formes.length;i++) surface_totale += formes[i].surface();
```

Interfaces

Le langage Java n'autorise pas l'héritage multiple - une classe hérite d'au plus une classe - mais il fournit la notion d'interface qui permet de simuler l'héritage multiple. Les interfaces sont abstraites. Cela signifie que, à l'instar des classes abstraites, elles ne peuvent pas être instanciées.

Les interfaces ne fournissent que des définitions de méthodes abstraites. Elles doivent être implémentées en utilisant les mêmes noms et paramètres dans la classe d'implémentation. Une interface peut contenir la définition de constantes (*final static*).

Une classe peut implémenter une ou plusieurs interfaces, tout en héritant éventuellement d'une autre classe. Ce mécanisme permet de simuler les propriétés de sous-typage de l'héritage multiple: tous les types liés à une classe ne sont pas des sous-types des uns les autres.

Prenons comme exemple nos classes *Disque* et *Rectangle* et essayons de les élargir par les méthodes de dessin: *DessinerDisque()*, *DessinerRectangle()*.

Nous pouvons donc créer une classe abstraite *DessinerForme*. Nous voulons garder également nos classes et les méthodes *circonference()* et *surface()*. Mais nous avons déjà une super-classe- *Forme*. Pour résoudre ce problème nous pouvons définir une interface *DessinerForme*.

```
public interface DessinerForme {
public void Couleur(Color c);
public void Position(double x, double y);
public void Dessiner(DrawWindow dw);
}
```

Une classe peut implémenter une interface.

```
public class DessinerDisque extends Disque implements DessinerForme {
private Color c;
private double x,y,r;
public DessinerDisque(double x, double y, double r) {super(c,x,y,r);}
public void Couleur(Color c)
  {this.c = c;}
public void Position(double x, double y)
 { this.x = x; this.y = y;}
public void Dessiner(DrawWindow dw)
  { dw.drawCircle(x,y,r,c);}
}
```

Exemple d'utilisation:

```
Forme[] formes= new Forme[2];                        // tableau pour futures instances
DessinerForme[] dessins = new DessinerForme[2];   // tableau pour futures instances d'implementation
DessinerDisque dd = new DessinerDisque(2.5,5.0,1,green);
DessinerRectangle dr = new DessinerRectangle(5.2,8.4);
formes[0] = dd; dessin[0] = dd;
formes[1] = dr; dessin[2] = dr;
```

```
double surface_totale = 0.0;
for(int i=0;i<formes.length;i++) {
surface_totale += formes[i].surface();
DessinerForme[i].Position(i*20.0,i*5.0);DessinerForme[i].Dessiner(fenetre)
// fenetre doit etre predefinie dans le programme
}
```

De la même manière qu'une classe peut avoir des sous-classes, une interface peut avoir des sous-interfaces. Une sous-interface hérite toutes les méthodes abstraites d'une ou plusieurs interfaces.

```
public interface Flexible extends Extensible, Rotative, Amovible {}
```

Une interface qui étend plusieurs interfaces hérite l'ensemble de méthodes abstraites définies dans ses super-interfaces et peut définir ses propres méthodes abstraites. Une classe qui implémente une telle interface doit implémenter toutes les méthodes présentes dans toutes les super-interfaces ainsi que les méthodes de cette interface.

Résumé

Dans ce chapitre nous avons présenté un nombre important de concepts et de mécanismes propres à la programmation orientée objet en langage Java. Nous avons étudié la notion de la classe et de l'objet (instantiation), la notion de visibilité des variables et des méthodes, et le mécanisme d'héritage. Pour terminer nous avons illustré les classes abstraites et les interfaces.