Table des matières

1. Introduction

La **programmation orientée objets** encapsule les données, nommées *attributs*, et les méthodes, qui constituent les *comportements* dans des objets. Les données et les méthodes d'un objet sont intimement liées.

Les objets permettent le *masquage de l'information*. Les objets communiquent entre eux par l'entremise d'*interfaces* bien définies, mais l'implémentation des autres objets leur est normalement inconnue.

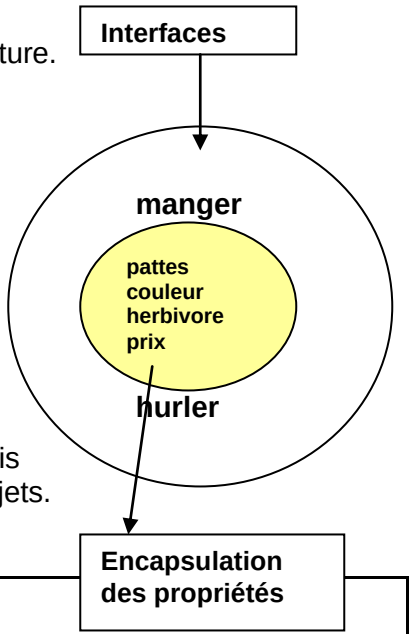
Les programmeurs Java se concentrent sur la création de leurs propres types définis, appelés *classes*, et la réutilisation de classes existantes. Un grand nombre de bibliothèques de classes existent et sont développées mondialement. Les logiciels se construisent à partir de composants existants, qui sont bien définis, testés adéquatement, suffisamment documentés, portables et largement disponibles. Cette réutilisation logicielle accélère le développement d'applications puissantes et de haute qualité.

Chaque classe contient ses données et les méthodes pour les manipuler. La classe représente en fait une définition de type plus sophistiqué que les types primitifs (char, byte, short, int, long, float, double, boolean). Les classes nous permettent de définir nos propres types (par exemple, la classe prédéfinie **String** est le type pour représenter les chaînes de caractères).

Une variable dont le type est une classe n'est pas un objet mais contient une référence à un objet (une *instance* de cette classe).

2. Conception d'une classe

Une classe est un *prototype* ou *modèle abstrait d'objet* qui définit des attributs et des méthodes communs à tous les objets d'une certaine nature.



Par exemple, les attributs de la classe **Animal** pourraient être le *nombre de pattes*, la *couleur*, le booléen *herbivore* et le *prix*.

Les méthodes sont les comportements qui déterminent ce que les objets de la classe peuvent faire. Par exemple, pour la classe **Animal**, nous pourrions avoir la méthode *manger* et la méthode *hurler*.

Selon le concept d'*encapsulation* des données, dont l'objectif est de cacher les détails d'implantation d'un objet pour les autres objets, l'accès aux données se fait par le biais d'interfaces (les méthodes). Les attributs doivent donc être des informations **privées** de l'objet tandis que les méthodes doivent être des interfaces **publiques** aux autres objets.

En Java, une classe se définit comme suit :

```
public class NomClasse
{
    déclaration des attributs (variables et constantes)
    déclaration des méthodes
}
```

Par convention, un nom de classe commence toujours par une majuscule. Chaque mot suivant commence également par une majuscule. La classe doit se trouver dans un fichier dont le nom est **NomClasse.java**.

Exemple de définition de la classe Animal en Java :

```
public class Animal
{
    // ATTRIBUTS
    private int pattes;
    private Color couleur;
    private double prix;
    private boolean herbivore ;

    // MÉTHODES
    public void hurler(){......}
    public void manger(){......}
} // fin Animal
```

3. Attributs (variables)

Attributs d'instance

En Java, un attribut d'instance se définit comme suit :

```
private type nomAttribut;
```

Chaque objet créé de la classe possède ses propres attributs d'instance.

Par convention, un nom d'attribut commence toujours par une minuscule. Chaque mot suivant commence par une majuscule.

Attributs de classe

En Java, un attribut de classe se définit comme suit :

```
static private type nomAttribut;
```

Il n'y a qu'une seule copie d'un attribut de classe, peu importe le nombre d'objets créés de la classe. Une variable de classe est donc une variable commune à tous les objets de la classe et elle peut être manipulée sans qu'un objet de cette classe ne soit créé (instancié).

4. Constantes

En Java, une constante de classe se définit comme suit :

```
public static final type NOM_CONSTANTE = valeurInitiale;
```

Une constante est une variable qu'on ne peut modifier une fois qu'elle a reçu sa valeur initiale. En programmation par objets, les constantes sont généralement **static** (une seule copie de la constante existe peu importe le nombre d'objets créés de la classe - constante *de classe*) et **public** (accessibles de partout).

Par convention, un nom de constante s'écrit toujours en majuscules, chaque mot étant séparé du suivant par le caractère souligné _.

Pour utiliser une constante de classe (à moins d'être dans cette classe), on doit la faire précéder du nom de la classe : **JOptionPane.PLAIN_MESSAGE**, **Font.BOLD**, **Integer.MAX_VALUE**.

Exemple de déclarations d'attributs et de constantes :

```
public class Film
{
    // déclarations des constantes de classe
    public static final char GÉNÉRAL = 'G',
                             ADULTE = 'A';

    // déclarations des attributs de classe
    private static int nbFilms = 0;

    // déclarations des attributs d'instance
    private String titre;
    private char cote;
    private boolean comique;
}
```

5. Méthodes

Méthode d'instance

En Java, une méthode d'instance se définit comme suit :

```
[modificateur d'accès] typeRetourné nomMéthode(liste de paramètres)
{
    corps de la méthode
}
```

modificateur d'accès : de façon générale, on met un accès **public** aux méthodes. Le but premier des méthodes **public** est de présenter les **services** de la classe, c'est-à-dire l'interface public de la classe, à ses clients. Les clients ne se soucient pas de la façon dont la classe accomplit ses tâches. Les méthodes **private**, que l'on qualifie de *méthodes utilitaires* ou *d'assistance*, supportent les opérations des autres méthodes de la classe et ne sont pas accessibles ailleurs.

typeRetourné : type de données primitif de la valeur retournée ou classe de l'objet retourné par la méthode. L'instruction **return** permet de retourner la valeur du type précisé. Une méthode ne peut retourner qu'une seule valeur. Si la méthode ne retourne rien, on met **void** comme type retourné.

liste de paramètres : variables locales à la méthode, elles reçoivent leur valeur seulement quand la méthode est appelée. Une méthode peut avoir aucun, un ou plusieurs paramètres. La transmission des paramètres effectifs, lors de l'appel de la méthode, peut être sous forme de constantes, d'expressions arithmétiques ou sous forme de variables. Il faut respecter le type des paramètres et l'ordre lors de l'appel.

Une méthode d'instance peut manipuler des attributs d'instance et des attributs de classe.

Pour utiliser une méthode d'instance (à moins d'être dans cette classe), on doit la faire précéder d'une référence à un objet de la classe : **chaine.toLowerCase()**, **sortie.setText(chaine)**, **sortie.append(texte)**

Méthode de classe

En Java, une méthode de classe se définit comme suit :

```
[modificateur d'accès] static typeRetourné nomMéthode(liste de paramètres)
{
    corps de la méthode
}
```

Une méthode de classe comporte les caractéristiques suivantes :

- il n'est pas nécessaire de disposer d'un objet de la classe pour l'utiliser
- elle ne peut manipuler que des attributs de classe
- elle est souvent considérée comme une méthode *utilitaire*, plus générale

Pour utiliser une méthode de classe (à moins d'être dans cette classe), on doit la faire précéder du nom de la classe : **Math.random()**, **Character.toUpperCase(carac)**, **Integer.parseInt(chaîne)**.

Méthodes d'accès

Comme les attributs sont privés, il est nécessaire d'offrir des méthodes qui vont permettre l'accès ou la modification des attributs.

Les méthodes de lecture ou accesseurs désignées aussi par méthodes ***« get »*** permettent d'obtenir les valeurs des attributs et comportent les caractéristiques suivantes :

- le nom de la méthode commence par **get** et est suivi du nom de l'attribut désiré (on met une majuscule après get)
- si l'attribut demandé est une variable de type boolean, on écrit souvent **est** plutôt que **get** (**is** en anglais)
- elle ne reçoit aucun paramètre
- elle retourne la valeur de l'attribut demandé
- si l'attribut demandé est **static**, la méthode accesseur l'est aussi

Les méthodes de mutation (mutateurs) qualifiées aussi de méthodes ***« set »*** permettent de modifier les valeurs des attributs et comportent les caractéristiques suivantes :

- le nom de la méthode commence par **set** et est suivi du nom de l'attribut désiré (on met une majuscule après set)
- elle reçoit un ou des paramètres permettant d'établir la nouvelle valeur à donner à l'attribut demandé
- elle ne retourne habituellement aucune valeur
- si l'attribut à modifier est **static**, la méthode de mutation l'est aussi

☞ Le concepteur d'une classe n'a pas à fournir des méthodes ***« get »*** et/ou ***« set »*** pour chacun des attributs privés de la classe. Il fournit ces méthodes après réflexion, seulement si elles sont nécessaires.

Exemple de déclarations d'attributs et de méthodes d'accès :

```
public class Film
{
    // déclarations des constantes de classe
    public static final char GÉNÉRAL = 'G',
                             ADULTE = 'A';

    // déclarations des attributs de classe
    private static int nbFilms = 0;

    // déclarations des attributs d'instance
    private String titre;
    private char cote;
    private boolean comique;

    // déclaration des méthodes d'accès
    public static int getNbFilms()
    { return nbFilms; }

    public String getTitre()
    { return titre; }

    public char getCote()
    { return cote; }

    public boolean estComique()
    { return comique; }

    // déclaration des méthodes de mutation
    public void setTitre(String unTitre)
    { titre = unTitre; }

    public void setCote(char uneCote)
    { cote = uneCote; }

    public void setComique(boolean c)
    { comique = c; }
}
```

Surcharge de méthodes

Quand vous travaillez avec les librairies de classes de Java, vous rencontrez souvent des classes possédant de nombreuses méthodes portant le même nom. A titre d'exemple, la classe String comprend plusieurs méthodes *indexOf()* différentes.

La surcharge de méthode (*overloading*) est le fait de donner le même nom à plus d'une méthode dans une même classe. Cependant, la liste des paramètres de ces méthodes doit être différente. Au moment de l'appel de la méthode, Java détermine laquelle des méthodes sera appelée en fonction des paramètres fournis.

L'opérateur **+** est un exemple d'opérateur surchargé : 3 + 2, 3.0 + 2.0, "bon" + "jour"

Exemples de méthodes surchargées :

```
int somme(int a, int b)
{ return a + b; }

int somme(int a, int b, int c)
{ return a + b + c; }

float somme(float a, float b)
{ return a + b; }
```

```
int x = 1,
    y = 2,
    z = 9;
float xx = 1.3,
      yy = 2.6;

int   iSomme = somme(x, y, z);
float fSomme = somme(xx, yy);
```

Attention ! Il n'est pas possible d'avoir deux méthodes (de même nom) dont tous les paramètres sont identiques, et dont seul le type de retour diffère. Le compilateur ne saurait pas quelle méthode employer. L'exemple ci-dessous ***n'est pas valide*** *:*

```
int somme(float a, float b)
{ return (int)(a + b); }

float somme(float a, float b)
{ return a + b; }
```

Méthode toString

Comme toute classe en Java hérite de la classe Object, toute classe possède une méthode spéciale dont la signature est *String toString()*, qui peut être utilisée pour convertir un objet de cette classe en String. On peut alors l'utiliser pour afficher les informations contenues dans un objet comme ceci :

```
// appel explicite
System.out.println("Informations de l'objet :" + nomObjet.toString());

// appel implicite
System.out.println("Informations de l'objet :" + nomObjet);
```

Si on ne définit pas la méthode toString dans une classe, l'appel de cette méthode donnera des informations comme le nom de la classe et l'adresse de l'objet en mémoire.

6. Constructeurs

Le constructeur est une méthode particulière qui n'a pas de type de retour, pas même void, et qui porte le même nom que celui de la classe. Son rôle consiste à initialiser les attributs d'instance d'un objet lors de sa création. C'est le constructeur d'une classe qu'on appelle lorsqu'on crée un objet avec l'opérateur **new**.

Constructeur par défaut

Si aucun constructeur n'est défini dans la classe, le compilateur crée un constructeur par défaut, aussi appelé **constructeur sans paramètre**. Ce constructeur par défaut initialise à *0* les attributs de type numérique, à *false* les attributs de type boolean, à *\u0000* les attributs de type char et à *null* les attributs de type String et ceux dont le type est une classe.

Lorsque le programmeur définit au moins un constructeur pour la classe, Java ne crée pas automatiquement son constructeur par défaut. Si on a besoin d'un constructeur sans paramètre et qu'on a déjà un autre constructeur, il faut en définir un de façon explicite.

Surcharge de constructeurs

On parle de surcharge de constructeur lorsqu'on fournit plus d'un constructeur dans une classe. Tout comme pour une méthode surchargée, ces constructeurs doivent avoir une liste de paramètres différente.

Exemple de déclaration d'attributs et de constructeurs :

```
public class Film
{
    // déclarations des constantes de classe
    public static final char GÉNÉRAL = 'G',
                             ADULTE = 'A';

    // déclarations des attributs de classe
    private static int nbFilms = 0;

    // déclarations des attributs d'instance
    private String titre;
    private char cote;
    private boolean comique;

    // déclaration du constructeur sans paramètre
    public Film()
    {
        titre = "";
        cote = ' ';
        comique = false;
        nbFilms++;
    }

    // déclaration du constructeur avec 3 paramètres
    public Film(String unTitre, char uneCote, boolean c)
    {
        titre = unTitre;
        cote = uneCote;
        comique = c;
        nbFilms++;
    }
```

Lorsque les méthodes de mutation valident les données, il est d'usage de les appeler à partir du constructeur pour s'assurer de la validité des données reçues.

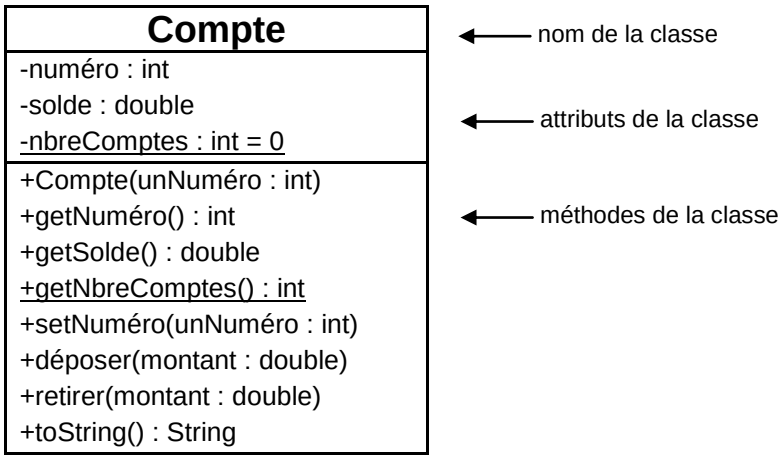
Exemple :

Compte
-numéro : int -solde : double -nbreComptes : int = 0
+Compte(unNuméro : int) +getNuméro() : int +getSolde() : double +getNbreComptes() : int +setNuméro(unNuméro : int) +déposer(montant : double) +retirer(montant : double) +toString() : String

```
public class Compte
{

    // déclaration des attributs
    private int numéro;
    private double solde;
    private static int nbreComptes = 0;

    // déclaration du constructeur
    public Compte(int unNuméro)
    {
        nbreComptes++;
        solde = 0;
        setNuméro(unNuméro);
    }

    // méthodes d'accès (accesseurs)
    public int getNuméro()
    { return numéro; }

    public double getSolde()
    { return solde; }

    public static int getNbreComptes()
    { return nbreComptes; }

    // méthodes de mutation (mutateurs)
    public void setNuméro(int unNuméro)
    {
        if (unNuméro > 0)     // si numéro valide
            numéro = unNuméro;
        else
            System.out.println(unNuméro + " n'est pas un numéro valide");
    }

    // autres méthodes d'instance
    public void déposer(double montant)
    {
        solde = solde + montant;
    }

    public void retirer(double montant)
    {
        if (montant <= solde)
            solde = solde - montant;
        else
            System.out.println("Solde insuffisant");

    }

    public String toString()
    {
        return "Compte : " + numéro + ", solde = " + solde);
    }
} // fin Compte
```

7. Représentation graphique d'une classe

On représente une classe de façon graphique en utilisant la notation UML (***U**nified **M**odeling **L**anguage*).



Le **-** devant un attribut ou une méthode signifie **private**.

Le **+** devant un attribut ou une méthode signifie **public**.

Un élément (attribut ou méthode) souligné indique un élément de **classe**.

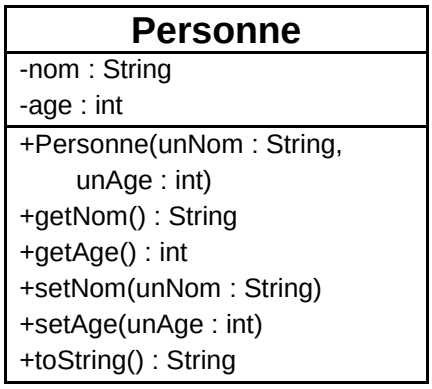
Classe Date

```
public class Date
{
    private int jour;
    private int mois;
    private int an;

    // constructeur par défaut
    public Date()
    {
        jour = 1;
        mois = 1;
        an   = 2007;
    }

    // constructeur d'initialisation
    public Date(int j, int m, int a)
    {
        jour = j;
        mois = m;
        an   = a;
    }

    public String toString()
    {
        String message = jour + "/" + mois + "/" + an;
        return message;
    }
} // fin Date
```

Date
-jour : int -mois : int -an : int
+Date() +Date(j : int, m : int, a : int) +toString() : String

Classe Personne

```
public class Personne
{
    // attributes d'instance
    private String nom;
    private int age;

    // constructeur d'initialisation
    public Personne(String unNom, int unAge)
    {
        nom = unNom;
        age = unAge;
    }

    // méthodes d'accès et de mutation
    public String getNom()
    { return nom; }

    public int getAge()
    { return age; }

    public void setNom(String unNom)
    { nom = unNom; }

    public void setAge(int unAge)
    { age = unAge; }

    // retourne une chaîne contenant les attributs de la personne
    public String toString()
    {
        return "Je m'appelle " + nom + " et j'ai " + age + " ans.";
    }
}
```



8. Utilisation de this

Dans une classe, le mot-clé **this** représente l'objet courant de cette classe.
L'utilisation de **this** permet d'employer comme paramètres (variables locales) les mêmes noms que ceux des attributs.

S'il est suivi de parenthèses, **this()** signifie plutôt l'appel à un autre constructeur de la classe. Si on l'utilise, il doit être la première instruction du constructeur.

Exemple de this :

```
public class Film
{
    // déclarations des constantes de classe
    public static final char GÉNÉRAL = 'G',
                             ADULTE = 'A';

    // déclarations des attributs de classe
    private static int nbFilms = 0;

    // déclarations des attributs d'instance
    private String titre;
    private char cote;
    private boolean comique;

    // déclaration du constructeur sans paramètre
    public Film()
    {
        this("", ' ', false);    // appel au constructeur avec 3 paramètres
    }

    // déclaration du constructeur avec 3 paramètres
    public Film(String titre, char cote, boolean comique)
    {
        this.titre = titre;
        this.cote = cote;
        this.comique = comique;
        nbFilms++;
    }

    public void setComique(boolean comique)
    { this.comique = comique; }
```

9. Création et utilisation d'objets

Création d'un objet

Créer un objet signifie créer une instance d'une classe à l'aide de l'opérateur **new**. Celui-ci appelle le constructeur approprié de la classe. L'opérateur **new** alloue de la place en mémoire pour l'objet tandis que le constructeur initialise les attributs de l'objet.

Exemples de création d'objets :

```
// déclare une référence à un objet de la classe Personne
// l'objet est null pour le moment
Personne enfant;

// crée l'objet enfant en appelant le constructeur
enfant = new Personne("Julie", 2);

// déclare et crée deux objets de la classe Personne
Personne mari = new Personne("Gilles", 27);
Personne femme = new Personne("Marie", 24);
```

```
// déclare 2 références à des objets de la classe Film
Film unFilm,
     unAutreFilm;

// crée le premier film en appelant le constructeur par défaut
unFilm = new Film();

// crée le second film en appelant le constructeur à 3 paramètres
// on notera l'utilisation de la constante de classe GENERAL
unAutreFilm = new Film("Bon cop bad cop", Film.GENERAL, true);

// déclare et crée un troisième film
Film violent = new Film("Aurore", Film.ADULTE, false);
```

```
// déclare et crée 3 objets de la classe Compte
Compte compte1 = new Compte(4687); // correct

Compte compte2 = new Compte(-45);

Compte compte3 = new Compte();     // incorrect
```

-45 n'est pas un numéro valide

On ne peut pas utiliser le **constructeur par défaut** parce qu'il n'a pas été défini dans la classe Compte

```
// crée une zone de texte en appelant le constructeur par défaut
JTextArea sortie = new JTextArea();

// crée une zone de texte avec une taille minimale
// en appelant le constructeur approprié
JTextArea zone = new JTextArea(10, 30);

// crée une zone de texte avec un texte initial
// en appelant le constructeur approprié
JTextArea resultat = new JTextArea("Texte initial");
```

Utilisation d'un objet

Une fois un objet créé par new, on peut appeler ses méthodes d'instance. Pour créer et utiliser des objets d'une classe, on écrit habituellement une classe séparée (application ou applet).

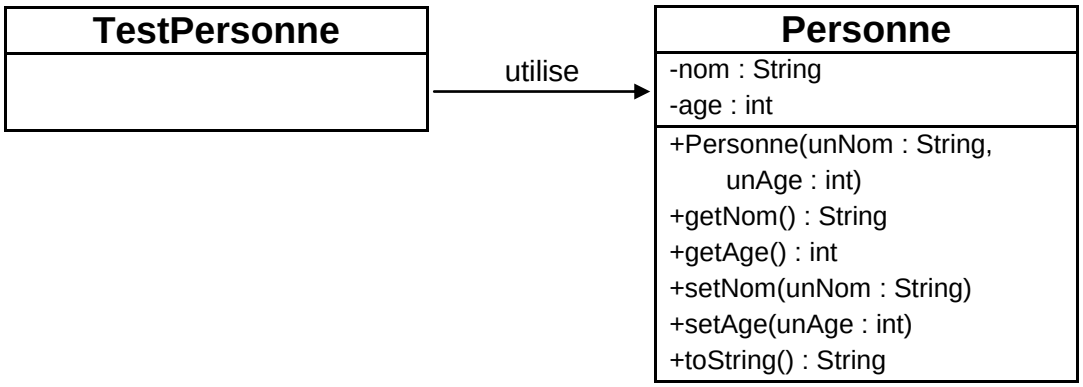
Après la création d'un objet, pour utiliser une méthode d'instance, on doit la faire précéder de la référence de l'objet : **chaine.toUpperCase()**, **sortie.setText(texte)**, **chaine.length()**.

Exemple d'application de test :

```
// Fichier TestPersonne.java
// Application permettant de tester la classe Personne
public class TestPersonne
{
    public static void main(String args[])
    {
        Personne mari = new Personne("Gilles", 27);
        Personne femme = new Personne("Marie", 24);

        mari.setAge(29);
        femme.setNom("Anne");
        femme.setAge(femme.getAge() + 1);   // ajoute 1 à l'âge de la femme

        System.out.println(mari.toString());
        System.out.println(femme);          // appel implicite à toString
    }
}
```

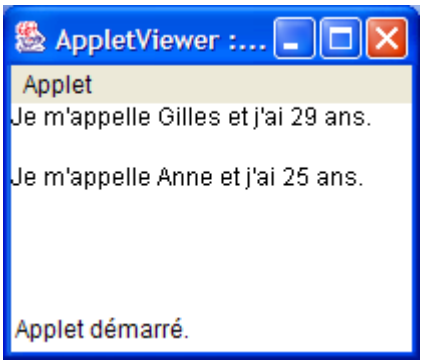
Résultats obtenus :

```
Je m'appelle Gilles et j'ai 29 ans.
Je m'appelle Anne et j'ai 25 ans.
```



Exemple d'applet de test :

```
// Fichier AppletTestPersonne.java
// Applet permettant de tester la classe Personne
import java.applet.*;
import java.awt.Graphics;
public class AppletTestPersonne extends Applet
{
    Personne mari;
    Personne femme;

    public void init()
    {
        mari = new Personne("Gilles", 27);
        mari.setAge(29);

        femme = new Personne("Marie", 24);
        femme.setNom("Anne");
        femme.setAge(femme.getAge() + 1);   // ajoute 1 à l'âge de la femme

        setSize(200, 100);
    }

    public void paint(Graphics g)
    {
        g.drawString(mari.toString(), 0, 10);
        g.drawString(femme.toString(), 0, 40);
    }
}
```



On pourrait également ajouter une méthode **main** dans la classe Personne. Cette méthode ne servirait qu'à tester la classe Personne pour s'assurer de sa validité sans avoir besoin d'une autre classe de test. Une fois la classe testée, la méthode main ne sert plus à rien.

```
public class Personne
{
    // attributs d'instance
    private String nom;
    private int age;

    // constructeur d'initialisation
    public Personne(String unNom, int unAge)
    {
        nom = unNom;
        age = unAge;
    }

    // méthodes d'accès et de mutation
    public String getNom()
    { return nom; }

    public int getAge()
    { return age; }

    public void setNom(String unNom)
    { nom = unNom; }

    public void setAge(int unAge)
    { age = unAge; }

    // retourne une chaîne contenant les attributs de la personne
    public String toString()
    {
        return "Je m'appelle " + nom + " et j'ai " + age + " ans.";
    }

    // méthode main pour tester la classe
    public static void main(String args[])
    {
        Personne test = new Personne("Jean", 37);
        System.out.println("Avant modification :\n" + test);

        test.setNom(test.getNom() + "-Pierre");
        test.setAge(42);

        System.out.println("\nAprès modification :\n" + test);
    }
}
```

Résultats obtenus sur la console :

```
Avant modification :
Je m'appelle Jean et j'ai 37 ans.

Après modification :
Je m'appelle Jean-Pierre et j'ai 42 ans.
```

10. Exemples de classes

Classe Date

```
public class Date
{
    private int jour;
    private int mois;
    private int an;

    // constructeur par défaut
    public Date()
    {
        jour = 1;
        mois = 1;
        an   = 2007;
    }

    // constructeur d'initialisation
    public Date(int j, int m, int a)
    {
        jour = j;
        mois = m;
        an   = a;
    }

    public String toString()
    {
        String message = jour + "/" + mois + "/" + an;
        return message;
    }
} // fin Date
```

Date
-jour : int -mois : int -an : int
+Date() +Date(j : int, m : int, a = int) +toString() : String

```
public class TestDate
{
    public static void main(String args[])
    {
        String texte = "";
        Date d1;                  // pas d'instanciation, seulement déclaration

        Date d2 = new Date();     // avec constructeur sans paramètre

        Date d3 = new Date(26, 12, 1948);   // avec constructeur à 3 paramètres

        texte += "La date d2 : " + d2 +
                 "\nLa date d3 : " + d3;

        d1 = d3;                  // d1 fait référence au même objet que d3

        texte += "\nLa date d1 : " + d1;

        System.out.println(texte);
    }
} // fin TestDate
```

Résultats obtenus :

```
La date d2 : 1/1/2007
La date d3 : 26/12/1948
La date d1 : 26/12/1948
```

Classe Rectangle

```
public class Rectangle
{
    // attributs privés
    private int hauteur;
    private int largeur;

    // constructeur par défaut
    public Rectangle()
    {
        this(0, 0);
    }

    // constructeur d'initialisation
    public Rectangle(int h, int l)
    {
        hauteur = h;
        largeur = l;
    }

    public int calculerAire()
    {
        return hauteur * largeur;
    }

    public String toString()
    {
        String message = "Rectangle [h=" + hauteur + ", l=" + largeur + "]";
        return message;
    }

    // pour tester la classe
    public static void main(String args[])
    {
        Rectangle rect = new Rectangle(12, 35);
        int aire;

        aire = rect.calculerAire();
        System.out.println(rect + ", aire=" + aire);
    }
} // fin Rectangle
```

Résultats obtenus :

```
Rectangle [h=12, l=35], aire=420
```

Classe Dinosaure

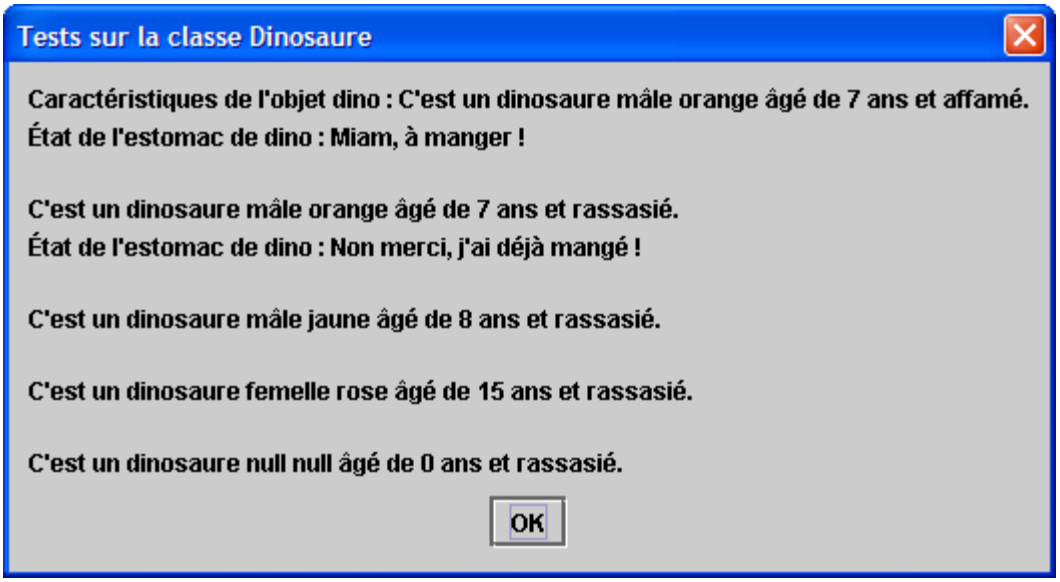
```
// Fichier Dinosaure.java
// Définition des attributs et comportements d'un dinosaure
public class Dinosaure
{
    private String sexe;
    private String couleur;
    private int âge;
    private boolean affamé;

    public Dinosaure()
    {
        sexe = null;
        âge = 0;
        couleur = null;
        affamé = false;
    }

    public Dinosaure(String s, int a, boolean faim)
    {
        sexe = s;
        âge = a;
        couleur = "orange";
        affamé = faim;
    }

    public String toString()
    {
        String texte = "C'est un dinosaure " + sexe + " " +
                       couleur + " âgé de " + âge + " ans et ");
        if (affamé)
            texte += "affamé.";
        else
            texte += "rassasié.";
        return texte;
    }

    public String nourrir()
    {
        String message;
        if (affamé)
        {
            message = "Miam, à manger !";
            affamé = false;
        }
        else
            message = "Non merci, j'ai déjà mangé !";
        return message;
    }

    public void fêter()
    {
        âge++;
    }

    public void setCouleur (String couleur)
    {
        this.couleur = couleur;
    }
}
```

Constructeurs

Méthodes d'instance

```
// Fichier TestDinosaure.java
// Exemple d'utilisation de la classe Dinosaure
import javax.swing.*;
public class TestDinosaure
{
    public static void main(String args[])
    {
        String résultats = "";

        Dinosaure dino = new Dinosaure("mâle", 7, true);
        Dinosaure dinette = new Dinosaure("femelle", 15, false);
        Dinosaure dirien = new Dinosaure();

        résultats += "Caractéristiques de l'objet dino : " + dino;
        résultats += "\nÉtat de l'estomac de dino : " + dino.nourrir();

        résultats += "\n\n" + dino;
        résultats += "\nÉtat de l'estomac de dino : " + dino.nourrir();

        dino.fêter();
        dino.setCouleur("jaune");
        dinette.setCouleur("rose");

        résultats += "\n\n" + dino;
        résultats += "\n\n" + dinette;
        résultats += "\n\n" + dirien;

        JOptionPane.showMessageDialog(null, résultats,
            "Tests sur la classe Dinosaure", JOptionPane.PLAIN_MESSAGE);
        System.exit(0);
    }
}
```

Résultats de l'exécution :



Classe Chien

```
// Fichier Chien.java
// Définition de la classe Chien
public class Chien
{
    public static final char DOBERMAN = 'D',
                             LABRADOR = 'L',
                             TECKEL = 'T';
    private static int nbChiens = 0;
    private String nom;
    private char race;
    private int age;

    // constructeur par défaut
    public Chien()
    {
        this("", DOBERMAN, 0);
    }

    // constructeur d'initialisation
    public Chien(String nom, char race, int age)
    {
        nbChiens++;
        setNom(nom);
        setRace(race);
        setAge(age);
    }

    public String getNom()
    { return nom; }

    public char getRace()
    { return race; }

    public int getAge()
    { return age; }

    public static int getNbChiens()
    { return nbChiens; }

    public void setNom(String nom)
    { this.nom = nom; }

    public void setRace(char race)
    { this.race = race; }

    public void setAge(int age)
    { this.age = (age > 0 ? age : 0); }

    // méthode privée utilisée dans toString seulement
    private String convertirRace()
    {
        String raceAuLong = "chien de race inconnue";
        switch (race)
        {
            case DOBERMAN : raceAuLong = "doberman"; break;
            case LABRADOR : raceAuLong = "labrador"; break;
            case TECKEL   : raceAuLong = "teckel"; break;
        }
        return raceAuLong;
    }

    public String toString()
    {
        return "Le chien " + nom + " est un " + convertirRace() +
            " âgé de " + age + " ans");
    }
}   // fin de la classe Chien
```

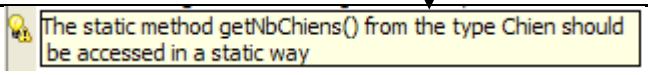
Chien
+DOBERMAN : char = 'D' +LABRADOR : char = 'L' +TECKEL : char = 'T -nbChiens : int = 0' -nom : String -race : char -age : int
+Chien() +Chien(nom : String, race : char, age = int) +getNom() : String +getRace() : char +getAge() : int +getNbChiens() : int +setNom(nom : String) +setRace(race : char) +setAge(age : int) -convertirRace() : String +toString() : String

opérateur conditionnel **?** remplace le if suivant :

```
if (age > 0)
    this.age = age;
else
    this.age = 0;
```

```
// Fichier TestChien.java
// Application permettant de tester la classe Chien

public class TestChien
{
    public static void main(String args[])
    {
        // appel de la méthode de classe, sans que la classe soit instanciée
        System.out.println("Le nombre de chiens est " + Chien.getNbChiens());

        // création de 2 instances (objets) de la classe Chien
        Chien pitou = new Chien();
        Chien grosPitou = new Chien("Chocolat", Chien.LABRADOR, 2);

        pitou.setNom("Vanille");
        grosPitou.setAge(5);

        System.out.println("Le nombre de chiens est " + Chien.getNbChiens());
        System.out.println("Le nombre de chiens est " + pitou.getNbChiens());

        System.out.println(pitou + "\n" + grosPitou);

        System.out.println("Les deux chiens sont " +
            pitou.getNom() + " et " +
            grosPitou.getNom());

        pitou.setRace(Chien.TECKEL);
        System.out.println(pitou);
    }
}
```

L'instruction va s'exécuter mais le compilateur va tout de même émettre un **warning** car on appelle une méthode de classe à partir d'une référence à un objet de la classe



Résultats de l'exécution :

```
Le nombre de chiens est 0
Le nombre de chiens est 2
Le nombre de chiens est 2
Le chien Vanille est un doberman âgé de 0 ans
Le chien Chocolat est un labrador âgé de 5 ans
Les deux chiens sont Vanille et Chocolat
Le chien Vanille est un teckel âgé de 0 ans
```

11. Utilisation de la classe Graphics

C'est un *contexte graphique* qui autorise le dessin à l'écran. Un objet graphique (objet de la classe Graphics) gère un contexte graphique en contrôlant la manière dont les informations sont dessinées. Les objets graphiques possèdent des méthodes de dessin, de manipulation de polices, de couleurs, et toutes choses du genre.

Un applet, par exemple, est un composant possédant un contexte graphique et permettant le dessin.

Gérer les couleurs

La classe **Color** possède des constantes, des constructeurs et des méthodes qu'on peut utiliser pour manipuler les couleurs dans un programme Java.

Constantes	Couleur	Valeur RGB
public final static Color ORANGE	orange	255, 200, 0
public final static Color PINK	rose	255, 175, 175
public final static Color CYAN	cyan	255, 0, 255
public final static Color MAGENTA	fushia	255, 0, 255
public final static Color YELLOW	jaune	255, 255, 0
public final static Color BLACK	noir	0, 0, 0
public final static Color WHITE	blanc	255, 255, 255
public final static Color GRAY	gris	128, 128, 128
public final static Color LIGHTGRAY	gris clair	192, 192, 192
public final static Color DARKGRAY	gris foncé	64, 64, 64
public final static Color RED	rouge	255, 0, 0
public final static Color GREEN	vert	0, 255, 0
public final static Color BLUE	bleu	0, 0, 255
Remarque : les noms des constantes de la classe Color peuvent également s'écrire en minuscules, mais on ne peut pas les écrire avec seulement la première lettre en majuscule.		
Constructeurs		
public Color(int r, int g, int b)	crée une nouvelle couleur basée sur le contenu de rouge, vert et bleu exprimé par des entiers compris entre 0 et 255	
public Color(float r, float g, float b)	crée une nouvelle couleur basée sur le contenu de rouge, vert et bleu exprimé par des valeurs en virgule flottante comprises entre 0.0 et 1.0	
Méthodes		
public int getRed()	retourne une valeur entre 0 et 255 qui représente le contenu de rouge	
public int getGreen()	retourne une valeur entre 0 et 255 qui représente le contenu de vert	
public int getBlue()	retourne une valeur entre 0 et 255 qui représente le contenu de bleu	

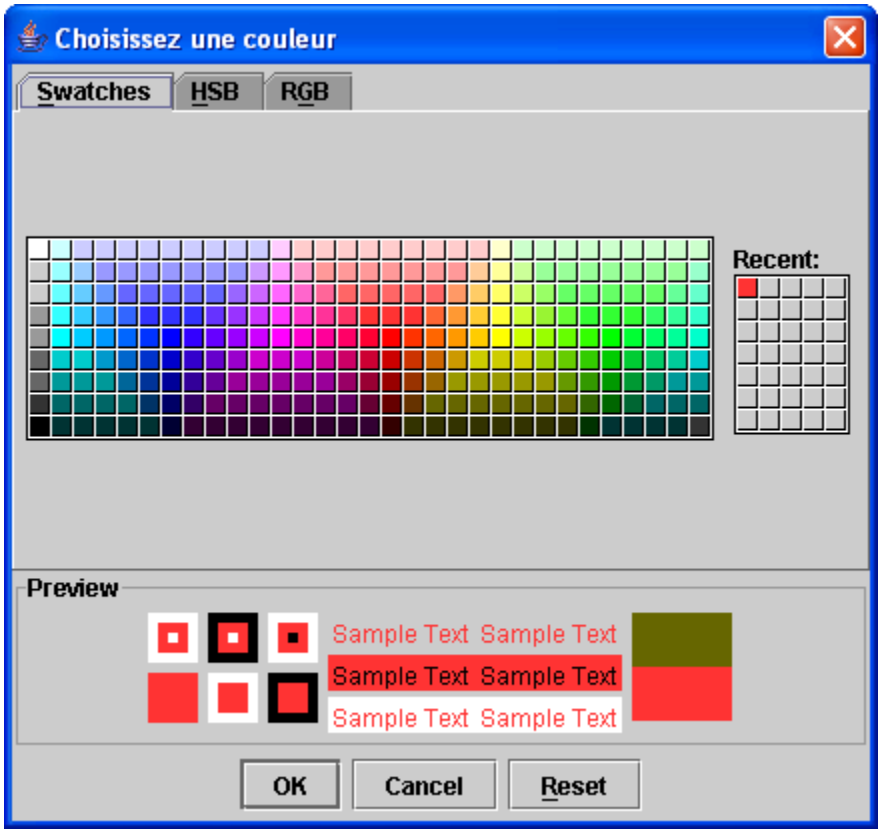
Exemples d'objets de la classe Color :

```
Color c1 = new Color(255, 200, 0);     // orange
Color c2 = Color.YELLOW;               // jaune
```

La classe **JColorChooser** permet d'afficher une boîte de dialogue permettant de choisir une couleur.

Exemple d'utilisation de JColorChooser :

```
// afficher la boîte de dialogue permettant le choix d'une couleur
private Color choisirCouleur(Color couleurActuelle)
{
    Color couleur = JColorChooser.showDialog(null,
                    "Choisissez une couleur", couleurActuelle);

    // vérifie si on a bien choisi une couleur,
    // sinon on garde la couleur actuelle
    if (couleur == null)
        couleur = couleurActuelle;

    return couleur;
}
```



Dessiner des formes

La classe **Graphics** possède des méthodes pour dessiner des formes dans un composant graphique. Ces méthodes sont normalement utilisées dans une méthode nommée **paint** ou appelées par la méthode **paint**.

N.B. On suppose que **g** est le contexte graphique, **col** est le numéro de la colonne (en pixels) et **lig** est le numéro de la ligne (en pixels). Toute **couleur** est un objet ou une constante de la classe Color.

Quelques méthodes de dessin	
g.drawString(chaîne, col, lig)	dessine une chaîne commençant au point donné par col et lig
g.drawLine(col, lig, col2, lig2)	dessine une ligne du point donné par col et lig jusqu'au point donné par col2 et lig2
g.drawOval(col, lig, largeur, hauteur);	dessine un ovale vide dont le coin supérieur droit est donné par col et lig
g.fillOval(col, lig, largeur, hauteur)	comme *drawOval*, mais l'ovale est plein
g.drawRect(col, lig, largeur, hauteur)	dessine un rectangle dont le coin supérieur droit est donné par col et lig
g.fillRect(col, lig, largeur, hauteur)	comme *drawRect*, mais le rectangle est plein
g.clearRect(col, lig, largeur, hauteur)	comme *fillRect* mais le rectangle est dessiné dans la couleur d'arrière-plan en cours
g.drawRoundRect(col, lig, largeur, hauteur, arcLargeur, arcHauteur)	dessine un rectangle vide à coins arrondis dont le coin supérieur droit est donné par col et lig arcLargeur et arcHauteur sont des entiers qui déterminent l'arrondi des coins
g.fillRoundRect(col, lig, largeur, hauteur, arcLargeur, arcHauteur)	comme *drawRoundRect* mais le rectangle est plein
g.draw3dRect(col, lig, largeur, hauteur, relevé)	dessine un rectangle vide dont le coin supérieur droit est donné par col et lig si la booléenne relevé est **true**, le rectangle paraît relevé sinon il paraît enfoncé
g.fill3dRect(col, lig, largeur, hauteur, relevé)	comme *draw3dRect* mais le rectangle est plein
g.drawArc(col, lig, largeur, hauteur, angleDépart, arcAngle)	dessine un arc par rapport au rectangle dont le coin supérieur droit est donné par col et lig le tracé de l'arc débute à l'angle de départ et s'étend sur arcAngle degrés
g.fillArc(col, lig, largeur, hauteur, angleDépart, arcAngle)	comme drawArc mais l'arc est plein, c'est-à-dire un secteur
Autres méthodes de la classe Graphics	
setColor(Color c)	modifie la couleur courante (utilisée dans les instructions de dessin subséquentes)
setFont(Font f)	définit la police courante à la police, au style et à la taille spécifiés dans la référence **f** à un objet Font

Exemple de dessin d'objet

```
import java.awt.*;
public class RectanglePlein
{
    // attributs privés
    private int hauteur;
    private int largeur;
    private int x;
    private int y;
    private Color couleur;

    // constructeur d'initialisation (position et couleur par défaut)
    public RectanglePlein(int h, int l)
    {
        hauteur = h;
        largeur = l;
        x = 100;
        y = 10;
        couleur = Color.BLACK;
    }

    public int getX()
    { return x; }

    public int getY()
    { return y; }

    public void setHauteur(int h)
    { hauteur = h; }

    public void setXY(int nouveauX, int nouveauY)
    {
        x = nouveauX;
        y = nouveauY;
    }

    public void setCouleur(Color c)
    { couleur = c; }

    // méthode de dessin
    public void dessine(Graphics g, int x, int y, Color couleur)
    {
        g.setColor(couleur);
        g.fillRect(x, y, largeur, hauteur);
    }

    public String toString()
    {
        return "RectanglePlein [h=" + hauteur + ", l=" + largeur + "]";
    }
}
```

```
import java.applet.Applet;
import java.awt.*;
public class AppletTestRectanglePlein extends Applet
{
    RectanglePlein rect;
    RectanglePlein carre;

    public void init()
    {
        rect = new RectanglePlein(50, 100);
        carre = new RectanglePlein(30, 30);
        setSize(300, 300);
        setBackground(Color.yellow);
    }

    public void paint(Graphics g)
    {
        rect.setCouleur(Color.BLUE);    // on modifie la couleur
        rect.dessine(g);

        carre.setXY(50, 200);
        carre.setCouleur(new Color(50, 200, 50));  // à peu près vert
        carre.dessine(g);

        rect.setHauteur(150);           // on modifie la hauteur
        rect.setXY(rect.getX(), 140);   // on modifie seulement y
        rect.setCouleur(Color.RED);     // on modifie la couleur
        rect.dessine(g);

        g.setColor(Color.BLACK);
        g.setFont(new Font("Arial", Font.ITALIC, 18));
        g.drawString(rect.toString(), 20, 120);
    }
}
```