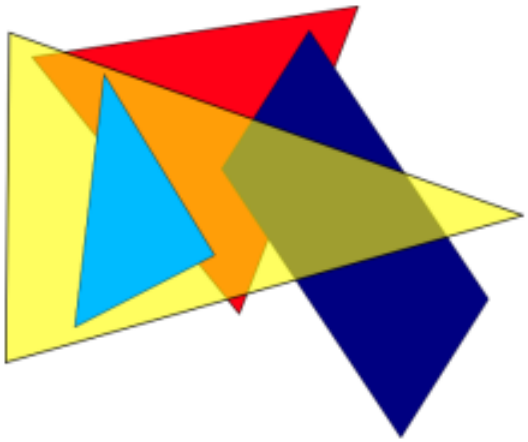
Jacques THOORENS

Modéliser et programmer en Java



Edition 2011

École de Commerce et d'Informatique
Liège

Chapitre 1

Introduction

À travers l'étude de quelques exemples et quelques exercices simples, ce premier chapitre va permettre de maîtriser les points suivants :
- différencier un langage fonctionnel d'un langage objet comme Java
- modéliser une classe en partant de ses responsabilités
- modéliser l'interaction de plusieurs classes entre elles
- utiliser les structures de base du langage Java
- surcharger un constructeur et une méthode
- définir une petite application qui créera plusieurs objets et les fera interagir

1.1 Présentation rapide de Java

1.1.1 Aux origines de Java

Les premières années de la décennie 90 voient Internet prendre son essor. Après l'invention des navigateurs, on se rend compte que les pages statiques n'attireront pas longtemps les utilisateurs. Deux solutions sont envisagées : générer des pages originales en interfaçant des programmes et des modules avec le serveur Web ou rendre la navigation plus dynamique en faisant exécuter du code sur la machine du client. Dans cette seconde technique, deux problèmes apparaissent : le programme exécuté sur le client ne doit pas mettre en péril la machine distante et le client disposant *a priori* d'un système inconnu, il est difficile de lui fournir du code exécutable. À moins de prévoir autant de types de code à envoyer qu'il y a de systèmes dans le monde, il faut se résoudre à envoyer un code universel, compréhensible par tous. Une solution légère intègre un interpréteur dans le navigateur, ce qui limite son action à la mise en page du contenu. Cette solution, qui concilie légèreté et une relative sécurité, obligera les navigateurs à se recycler et aura un succès certain : c'est Javascript. Des solutions lourdes, imposant un interpréteur plus complexe, verront simultanément le jour ou se développeront par la suite : c'est le cas de Java et de Flash. Leurs destins seront différents : Flash, moins gourmand en ressource, reste largement répandu alors que Java a peu à peu déserté les pages Web pour s'installer sur la machine comme un moyen de faire tourner des programmes à part entière. Paradoxalement, Java est allé jusqu'à envahir les serveurs et y sert souvent aujourd'hui à animer les sites par l'autre extrémité.

Java a été développé par la société Sun Microsystems, auparavant connue pour sa maîtrise de stations performantes, basées sur des processeurs RISC et des systèmes UNIX. L'un des maîtres d'œuvre du langage est James GOSLING.

Il existe aujourd'hui trois implémentations complémentaires de Java :
- **Java Standard Edition** permet de réaliser des programmes entièrement portables, tout en offrant toujours la possibilité de les intégrer à une page Web (on parle alors d'*applets*).

- **Java Enterprise Edition** enrichit l'édition standard de fonctionnalités lui permettant de s'interfacer avec le serveur (on parle alors de *servlets*).
- **Java Mobile Edition** est un environnement de développement semblable à Java, mais plus réduit, plus spécialisé. Il est implémenté dans des millions d'appareils électroniques allant du téléphone portable à la machine à laver, souvent d'ailleurs à l'insu de l'utilisateur. Si on peut dire qu'il existe plus d'utilisateurs de Java que de Windows, c'est en se référant à cette version de Java. Elle sort totalement du cadre de cette formation et il n'en sera plus question.

Trois impératifs ont guidé la création et l'évolution de Java : la sécurité, la portabilité et la vitesse d'exécution.

La sécurité

Elle est intégrée dans les structures fondamentales du langage. Par définition, le langage interdit les opérations sources de problème : surcharge des opérateurs, manipulation des pointeurs, héritage multiple. Des mécanismes complexes assurent la cohérence au niveau des instructions exécutées (impossibilité de modifier les types d'un opérande), au niveau de chaque procédure (contrôle de l'intégrité de la pile) et au niveau du programme (contrôle d'intégrité du programme). Les droits concédés au programme ne dépendent pas de lui-même mais de l'environnement hôte. Il appartient donc à l'utilisateur de définir ce que le programme Java sera autorisé à faire. Dans le cas d'une applet, les droits d'accès aux ressources de la machine sont définis par défaut de manière très restrictive (par exemple, une applet ne peut charger du code Java qu'en provenance du site dont elle est issue). Un système de signature, facultatif, permet aussi de vérifier l'authenticité du code reçu.

La portabilité

Elle est assurée de manière optimale par l'implémentation d'une machine virtuelle sur laquelle un code machine spécifique, le *byte code*, s'exécute. La machine virtuelle s'intègre dans le navigateur ou constitue une application de la machine hôte. Si les machines virtuelles respectives sont bien conçues [1], une application Java s'exécutera de manière similaire sur différentes architectures matérielles et logicielles. On constatera seulement quelques petites variations dans l'aspect graphique (à la condition que la programmation de l'interface graphique soit réalisée en respectant les règles de l'art).

La vitesse d'exécution

Ce problème a toujours constitué une faiblesse de Java (en même temps que de grosses exigences en mémoire). Le principe de la machine virtuelle n'est pas la seule cause du problème puisque de nombreux environnements actuels privilégient cette solution (.net). Sun fait de constants efforts pour améliorer les performances des machines virtuelles. Le programmeur doit y mettre du sien également et savoir que certaines techniques mobilisent plus de ressources que d'autres. Dans certains contextes (sites Web fortement sollicités), cet aspect devient critique.

1. Pendant plusieurs années, Microsoft a fourni dans ses systèmes d'exploitation des machines virtuelles Java qui s'écartaient parfois des spécifications techniques de Sun, dans une intention de les améliorer selon les uns, dans le but de discréditer Java selon Sun. L'action légale intentée par Sun s'est terminée par un retrait, suite au versement par Microsoft d'une somme colossale à son adversaire. Microsoft n'a jamais vu d'un bon œil l'engouement pour Java. Sa stratégie J++ a été un cuisant échec. Il semble que C#, largement inspiré par Java, obtienne de meilleurs résultats : c'est dû sans doute à son intégration dans .net.

1.1.2 Historique de Java

La première version de Java (1.0) n'a qu'un intérêt historique. De nombreuses lacunes et des stratégies jugées maladroites ont été corrigées par la suite. La version 1.1 s'est trouvée intégrée dans les versions standard de Netscape et d'Internet Explorer pendant plusieurs années.

La version 1.2 sonne l'apparition de ce qui sera appelé Java 2 (et ses trois déclinaisons J2ME, J2SE et J2EE). Outre de nombreux ajouts et améliorations, cette version se caractérise par l'arrivée d'une nouvelle gestion de l'interface graphique, Swing, qui remplace, sans l'évincer complètement, l'ancienne AWT. Java 2 connaîtra deux mises à jour. La version 1.3 corrige des bugs et améliore les performances, tout en dotant les aspects serveurs de fonctionnalités nouvelles. Cette tendance sera renforcée dans la version 1.4, qui introduira aussi le support d'XML.

La version 1.5, aussi nommée 5.0, constitue une révolution, dont on n'a pas encore perçu toutes les conséquences. Quelques modifications de détails font plaisir aux programmeurs (énumérations, simplification des classes d'emballage), mais l'introduction de la généricité remodèle bien des aspects de la structure du langage.

L'anarchie des numéros de version a trouvé son terme. Tout le monde parle de Java 5 et les versions suivantes seront nommées Java 6 SE, puis Java 7SE. La version courante actuelle, Java 6, a introduit de nouvelles modifications en enrichissant un concept déjà présent dans la version précédente : les annotations. Il s'agit d'innovations pointues qui concernent des programmeurs déjà très expérimentés.

1.1.3 Caractéristiques fondamentales de Java

langage objet

Java est un langage manipulant des classes. À la différence de Javascript, il ne se limite pas à utiliser des objets dérivant de classes prédéfinies. Il autorise et favorise la définition de nouvelles classes par l'utilisateur. En fait, la programmation Java consiste essentiellement à créer des nouvelles classes.

fortement typé

Le typage fort de Java oblige à une grande rigueur de programmation. La syntaxe héritée de C présente pas mal de contextes ambigus où des erreurs peuvent se nicher. Un contrôle strict des types manipulés déclenche deux types d'erreurs : de nombreuses erreurs de compilation visent à la création d'un code plus sûr, des erreurs d'exécution sanctionnent toute utilisation d'un type mal approprié. Cela suppose que les variables restent typées dans le byte code, une originalité par rapport à d'autres langages.

maximisation des erreurs de compilation

L'évolution de Java montre une augmentation de la détection des erreurs au niveau de la compilation. L'introduction de la généricité constitue à cet égard une évolution majeure, notamment, mais pas seulement, dans la gestion des collections.

API [2] Java

Contrairement au C, qui dispose bien sûr de quelques bibliothèques standardisées, Java se définit par ses structures syntaxiques mais aussi par une énorme API. La version 5 compte

2. Application Programming Interface

environ 2800 classes prédéfinies, avec leurs attributs et leurs méthodes. Elles sont fournies avec une documentation précise et standardisée, qui fournit des outils navigationnels et la possibilité de visualiser le code [3].

1.2 Outils nécessaires et installation

Pour exécuter du code Java, il faut disposer d'une machine virtuelle. Intégrée dans les anciens navigateurs, elle est le plus souvent disponible sous la forme d'un programme exécutable et de bibliothèques. On peut se procurer cet ensemble, nommé *Java Runtime Environment* ou JRE sur le site de Sun. Notons qu'il est possible d'installer plusieurs versions différentes du JRE [4]. Pour développer, il faut au moins disposer du compilateur et des outils annexes (bibliothèques de développement et documentation). Le *Java Development Kit* ou JDK contient tout ce qui est nécessaire pour la compilation et l'exécution d'un programme Java (il intègre notamment le JRE qui lui correspond). La documentation complète fait l'objet d'un autre fichier d'archives. Enfin, l'écriture de projet d'envergure nécessite l'emploi d'un environnement de développement intégré (IDE).

À côté des environnements commerciaux fournis par Sun, Borland ou IBM, on pourra utiliser deux environnements gratuits : **NetBeans**, repris maintenant par Sun, ou **Eclipse**, projet Open Source soutenu entre autres par IBM. Le premier permet de créer des environnements graphiques sans nécessiter de plug-in. Tous ces outils sont disponibles sur Internet. La solution la plus simple consiste à télécharger la version conjointe NetBeans+JDK (environ 130 Mo) sur le site *www.netbeans.org*. Le site principal de Sun (*http ://java.sun.com*) offre des JDK pour différentes plates-formes, ce qui permet de mettre à jour son environnement. La version actuelle de NetBeans est 6.9.1 et celle de JDK 6.0 Update 21. Il est conseillé de faire les mises à jour régulièrement. La documentation complète est également disponible sur le site de Sun. On peut l'installer par décompression dans le répertoire du JDK. Les paramètres de l'utilisateur ne sont pas affectés par les changements de version. Par contre, tout ajout de bibliothèque, par exemple un gestionnaire de base de données, devra être réinstallé. En général, cela se réduit à recopier un fichier par bibliothèque.

Le rachat de Sun par Oracle rend particulièrement compliquée la situation de Java. Une bonne partie de l'équipe de développement a quitté la firme à la suite de James Gosling. Une implémentation libre du langage, initialement autorisée par Sun et nommée OpenJDK, n'est pas vue d'un trop bon oeil par Oracle, qui a récemment réussi à liguer contre lui une bonne partie de la communauté des développeurs de logiciels libres. Face à ces querelles, Microsoft, dont le langage phare C# a le vent en poupe, compte les points.

1.3 Langages fonctionnels et langages à objets

La programmation traditionnelle, parfois appelée programmation assertive ou fonctionnelle, modélise la réalité au moyen d'un certain nombre de variables, dont les valeurs successives correspondent à des mesures instantanées de différents aspects du réel. L'assertion y est

3. Java est gratuit et, comme on le voit, lisible. Sun reste toutefois vigilant sur la possibilité de modifier le code source, maintenant disponible. Cette restriction se justifie par le besoin d'un langage unique et standardisé, au comportement prévisible. Java dispose depuis peu de sources libres, mais les tests de conformités à la norme restent l'apanage de Sun.

4. Certains programmes installent leur propre version du JRE, qui n'est pas nécessairement la dernière. C'est le cas d'Oracle. On pourra faire de la place en supprimant les versions qui font double emploi. Il faudra évidemment établir des liens ou modifier des variables d'environnement pour que chaque programme trouve la version qu'il réclame.

l'instruction de base, qui consiste à attribuer une valeur à une variable ou à la modifier. Des fonctions (ou des procédures) regroupent des séquences d'assertions, imbriquées dans des alternatives et des boucles. Chaque variable a un rôle passif, elle se modifie au gré des assertions.

La programmation orientée objet donne un statut autrement plus important à des variables particulières, nommées objets. Un objet appartient à une classe et comme tel, il est capable d'action, on dit qu'il possède des responsabilités. La classe se résume à un ensemble d'opérations que les objets qui en dérivent sont capables de réaliser, seuls ou en interaction avec d'autres objets. Les opérations se déclenchent par l'envoi d'un message à l'objet dont elles dépendent.

Une présentation fréquente des langages à objet compare les structures d'un langage classique comme le C et les objets en disant que les objets sont des structures particulières, qui *encapsulent* les champs, qui les rendent invisibles, par le biais d'une portée privée. Cette vision n'est pas totalement fausse, mais elle donne trop d'importance aux champs, qu'on appelle aussi attributs ou variables d'instance. On dit alors que les méthodes permettent d'accéder aux champs, alors qu'en fait c'est l'inverse : les attributs sont nécessaires aux objets pour assumer leurs responsabilités, qui constituent l'essentiel du service attendu d'un objet.

1.4 Modélisation d'une classe simple

Pour clarifier les choses, prenons un exemple simple, celui d'un compte en banque. Dans une programmation traditionnelle, on insistera probablement sur tous les valeurs informatives liées à un compte en banque : son solde, son numéro, le nom de son titulaire et la liste des mouvements qui l'ont affecté au cours des dernières semaines. En programmation-objet, on s'intéressera plutôt aux responsabilités d'un compte :

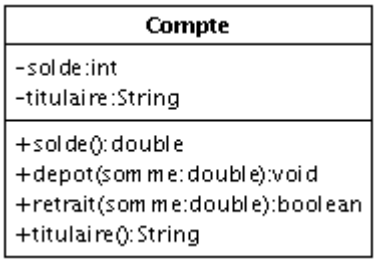
- donner son solde
- accepter un dépôt
- autoriser un retrait
- fournir la liste des dernières opérations qui l'ont affecté
- donner des renseignements sur son titulaire

Cette énumération n'est évidemment pas contradictoire avec l'examen des variables qu'on aurait dû définir dans un programme classique. On remarque pourtant que la liste des opérations et le titulaire peuvent également s'envisager dans une perspective objet et proposer également des services. La liste des opérations bancaire est susceptible de s'agrandir ou peut faire l'objet de recherches selon certains critères, le titulaire peut donner son nom, son adresse ou déménager.

On peut se demander quel avantage réel posséderait l'opération de retrait, par rapport à une procédure classique à laquelle on passerait en argument une structure regroupant tous les paramètres du compte. Outre le fait qu'une procédure pose le problème de l'effet réel d'un appel de procédure au niveau des variables du programme appelant (tout le problème des passages de paramètres), l'opération réalisera le retrait en prenant en compte les caractéristiques de l'objet, non seulement les valeurs de ses attributs, mais sa nature même. C'est ainsi qu'un compte dont le solde est trop bas ne pourra accepter un retrait que si le nouveau solde ne tombe pas sous la valeur du découvert autorisé et qu'un compte d'éparge, un type particulier de compte, ne pourra tolérer aucun découvert. Nous retrouverons ce problème plus tard, lors de l'examen des classes dérivées.

Des logiciels de modélisation fournissent un moyen commode de représenter les classes. La plupart d'entre eux utilisent la notation UML, adoptée par la plupart des acteurs importants de l'industrie logicielle. La présentation complète du *Unified Modeling Language*, et plus encore des méthodes d'analyse qui l'utilisent, dépassent le cadre de la présente formation. Nous

utiliserons simplement quelques éléments des diagrammes de classes. Voici une représentation standard de la classe *Compte* :



1.5 Création d'une classe, constructeurs et instanciation

En Java, une classe correspond en général à un fichier. Nous verrons plus tard que ce fichier est placé dans une arborescence significative pour la structure de l'application. Le fichier contiendra le code permettant de définir la classe :

- quelques instructions pour situer la classe par rapport au contexte dans lequel elle opère
- le nom de la classe, qui commence par une majuscule et doit correspondre au nom du fichier (attention aux majuscules sous Linux)
- le corps proprement dit qui se composera de la déclaration des variables d'instances et du code et des méthodes. Les identificateurs de ces variables et méthodes commencent par des minuscules.

L'usage du point virgule est omniprésent. Chaque bloc est entouré d'accolades.

Voici l'exemple minimal d'une classe, dont chaque instance sera capable de renvoyer la valeur de pi.

```
package divers; // environnement

public class Pi { // nom de la classe
    private float valeurPi = 3.141592F; // variable d'instance

    public float valeur() // méthode
    {
        return valeurPi;
    }
 }
```

On y voit que la classe fait partie du paquetage *divers*, que j'emploie dans mon projet pour placer quelques exemples simples. Elle possède une variable d'instance `valeurPi` de type flottant. On notera l'initialisation qui se fait directement, pour éviter une initialisation à la valeur 0 [5]. La seule méthode `valeur` renvoie la valeur de la seule variable d'instance.

Nous pouvons utiliser notre classe pour créer des objets *Pi*, capables de donner leur valeur. Dans une première version, nous disposerons d'une variable `pi1`. Nous pourrons ensuite initialiser cette variable avec un objet, une instance de *Pi*. Pour créer cette instance, nous allons utiliser l'opérateur `new` et le constructeur `Pi()`. Le constructeur porte toujours le nom de la classe. La variable `p1` est ensuite susceptible de recevoir le message `valeur()`, dont nous imprimons le résultat à l'aide de d'un appel à la classe *System*.

5. Sans le `F` final dans la constante, le compilateur estime la valeur de type `double` et proteste qu'on l'enregistre dans un `float`. Il ne s'agit pas d'un avertissement mais d'une erreur. Cela nous renvoie aux problèmes de sécurité évoqués plus haut.

```
Pi pi1;
pi1 = new Pi();
System.out.println(pi1.valeur());
```

Il est parfaitement possible (et cela se fait fréquemment) de déclarer et d'instancier un objet en une seule instruction.

```
Pi pi2 = new Pi();
System.out.println(pi2.valeur());
```

Dans le cas d'objets simples comme ceux de la classe *Pi*, il n'est pas toujours nécessaire d'utiliser une variable pour les manipuler. Dans l'instruction suivante, je crée un objet anonyme de type *Pi*, je lui envoie un message et le laisse mourir immédiatement. Après l'appel de `valeur`, l'objet cesse tout simplement d'exister.

```
System.out.println(new Pi().valeur());
```

Qu'en est-il du constructeur ? À aucun moment, je ne l'ai défini. Lors de la définition d'une classe, un constructeur est implicitement défini. Il portera le nom de la classe. Son travail sera de créer l'objet, d'initialiser ses variables d'instance à 0 (sauf celles qui disposent d'une initialisation spécifique) et de renvoyer la référence de l'objet créé. Ce constructeur implicite correspond au code suivant, que nous aurions pu écrire :

```
/** créer une instance de Pi */
public Pi() {
}
```

On notera l'absence de valeur de retour. Il est inutile de la préciser puisque le constructeur renvoie toujours un objet de la classe. Le constructeur n'est pas une méthode puisqu'il doit s'utiliser conjointement avec `new`.

Le constructeur que nous venons d'écrire ne sert pas à grand chose, puisqu'il fait ce qu'aurait fait le constructeur implicite. Nous pourrions écrire une nouvelle version [6] de la classe en rendant le constructeur responsable de l'initialisation de la variable `valeurPi`.

```
public class Pi {
    private float valeurPi; // initialisation à 0
    /** création d'une instance de Pi */
    public Pi() {
        valeurPi = 3.141592F;
    }

    public float valeur()
    {
        return valeurPi;
    }
}
```

6. Je laisse tomber la directive `package` pour simplifier. Dans un fichier complet, il ne faut évidemment pas la négliger.

Nous allons voir un autre exemple géométrique d'une classe Cercle où le constructeur devient explicite par nécessité : nous voulons préciser lors de la création d'une instance de cercle la longueur de son rayon. Je vais définir un constructeur qui acceptera un paramètre de type flottant qui me servira à initialiser la variable d'instance `rayon`.

```
package divers;

public class Cercle {

    private float rayon;
    private final float PI = 3.141592F;

    public Cercle(float rayon) {
        this.rayon = rayon;
    }

    public float longueur()
    {
        return 2*PI*rayon;
    }

    public float surface()
    {
        return PI*rayon*rayon;
    }
}
```

On voit que la variable d'instance est mentionnée de deux manières différentes dans le programme : `rayon` et `this.rayon`. Cette deuxième manière d'écrire est nécessitée par le fait que le paramètre du constructeur porte un nom identique et par là même masque la variable d'instance. `this` offre donc un moyen d'accéder à la variable d'instance masquée.

Nous sommes à présent en mesure d'écrire la classe Compte, plus complexe, mais plus utile.

```
public class Compte {
    private double solde;
    private String titulaire;
    private final double decouvert = -1250;

    public Compte() {
    }

    public void depot(double somme) {
        solde += somme;
    }
    public boolean retrait(double somme) {
        if(solde-somme<decouvert)
            return false;
        else
        {
            solde-=somme;
            return true;
```

```
        }
    }
    public String getTitulaire() {
        return titulaire;
    }
    public void setTitulaire(String titulaire) {
        this.titulaire = titulaire;
    }
   public double getSolde() {
        return solde;
    }
}
```

Dans cette version, le découvert fonctionne comme une constante, ce qui est marqué par le modificateur `final`.

1.6 Un programme minimal Java

Les outils de modélisation réalisent en général une grande partie du travail de conception des classes. Le squelette entier de chaque classe peut s'écrire automatiquement, en ce compris la définition des variables d'instance et de classe, en prenant en compte la visibilité, le mécanisme de navigation dans les associations, les entêtes de méthodes avec la définition des paramètres. Les outils bien conçus permettent de compléter le code des opérations, sans qu'il soit endommagé par des modifications mineures de la conception de la classe. Par contre, le programme proprement dit doit souvent être écrit à la main. Dans la plupart des cas, il consiste à créer un objet chargé du contrôle de l'application, celui-ci se chargeant ensuite de créer les autres objets et notamment l'interface homme-machine (IHM [7]).

En Java, un programme peut démarrer de plusieurs manières. Nous nous limiterons à examiner les programmes indépendants, qui ne dépendent donc pas d'une page Web. Ces programmes doivent obligatoirement commencer par l'invocation de la méthode `main` de la classe principale. La classe principale est celle qui est invoquée par la ligne de démarrage de la JVM. Cette méthode doit posséder certaines caractéristiques :

- elle est statique (elle n'est donc pas liée à une instance particulière de la classe principale).
- elle ne renvoie aucun valeur (Java utilise `void` comme en C).
- elle reçoit une série de paramètres chaînes fournis par la ligne de commande ou son équivalent dans Windows ou KDE.
- comme telle, elle ne peut invoquer que ses méthodes statiques. En pratique, elle devrait créer un ou plusieurs objets et leur déléguer la suite du travail. Si la méthode `main` se met à appeler une série de méthodes statiques et à manipuler des variables d'instance statiques, on se retrouve à faire de la programmation fonctionnelle.

Le programme d'exemple que nous allons écrire sera très peu interactif et se contentera de réaliser quelques transactions entre des comptes en affichant des messages sur le canal de sortie (afin de nous permettre de vérifier la validité des résultats). Au laboratoire, il nous arrivera fréquemment d'ajouter une méthode `main` à une classe afin de pouvoir en tester le fonctionnement. Dans un projet conséquent, il sera plus raisonnable de placer les méthodes `main`

7. J'utiliserai l'acronyme IHM, traduction de GUI *graphical user interface*, de préférence au terme d'*interface*, pour éviter la confusion avec la notion d'interface en Java.

destinées aux tests dans des classes spécifiques, qui seront écartées du projet final. On peut aussi utiliser le mécanisme de test prévu dans Java, qui fera l'objet d'explications ultérieures.

```
public static void main(String[] arg) {
    System.out.println("Création␣du␣compte");
    Compte c1 = new Compte();
    System.out.println("Solde:␣"+c1.getSolde()); // 0.0
    c1.depot(1000);
    System.out.println("Solde:␣"+c1.getSolde()); // 1000.0
    c1.retrait(1500);
    System.out.println("Solde:␣"+c1.getSolde()); // -500.0
    // le prochain retrait sera refusé
    c1.retrait(1000);
    System.out.println("Solde:␣"+c1.getSolde()); // -500.0
     System.out.println("");
}
```

La solution que nous avons développée ne fonctionne pas encore de manière optimale. La méthode `retrait()` constitue un progrès par rapport à la programmation traditionnelle puisque le dernier retrait n'est pas effectué, conformément aux règles en vigueur dans la banque. Le programmeur a malheureusement choisi de ne pas tenir compte de la valeur de retour de la méthode `retrait()`. Le programme illustre ici que l'objet possède une relative autonomie, mais dans une situation réelle, il n'est pas désirable que ce soit le programmeur qui doivent penser à vérifier si le retrait s'est bien effectué. La solution idéale dépasse le cadre de ce premier exposé : la méthode devrait lever une exception.

1.7 Surcharge des constructeurs et des méthodes

Notre banque peut modifier le comportement du compte en fonction de la fortune du client. Quelqu'un dont les revenus mensuels dépassent 5000 euros peut se permettre sans problème un découvert supérieur à 1250 euros. Une version élaborée du programme pourrait prendre en compte les revenus du titulaire pour fixer un plafond. Pour simplifier, nous laisserons cette décision à l'employé et tenterons de modifier la classe pour qu'elle soit susceptible de spécifier un découvert personnalisé. Une méthode pourrait se charger de modifier la variable `decouvert` (qui perdrait aussi son modificateur `final`). Nous allons plutôt choisir d'initialiser la constante à l'aide du constructeur (ce qui prouve bien que l'initialisation de toutes les variables d'instance se fait par son entremise). Voici la nouvelle version du constructeur :

```
public Compte(double _decouvert) {
     decouvert = _decouvert;
 }
```

Cependant, nous voici obligé de construire nos comptes en donnant un paramètre explicite :

```
Compte c37 = new Compte(-2000);
```

Il serait plus pratique de garder l'initialisation par défaut pour la plupart des comptes et de spécifier un découvert autorisé différent pour certains comptes. La possibilité de *surcharger* les constructeurs nous y autorise. Nous aurons donc deux constructeurs, dont la liste des paramètres (la signature) sera différente. Pour alléger l'écriture du deuxième constructeur, nous allons lui faire appeler le premier constructeur en fournissant le paramètre par défaut :

```
public Compte(double _decouvert) {
    decouvert = _decouvert;
}
public Compte(){
    this(-1250);
}
```

`this()` prend ici la place du constructeur de l'objet.

Nous allons pouvoir instancier deux comptes à l'aide des deux constructeurs :

```
Compte cNormal = new Compte();
Compte cRiche = new Compte(-5000);
```

La surcharge joue un rôle important dans le langage Java. Il faut évidemment que le compilateur sache quel constructeur employer : dans nos exemples, on voit clairement que le constructeur de la première ligne n'a pas d'argument et que celui de la seconde ligne utilise une valeur numérique (l'entier sera transformé en double).

Il est également possible de surcharger les méthodes. Ici encore, la signature permettra de savoir quelle version de la méthode il faut utiliser. Si nous ajoutons une variable d'instance pour contenir l'adresse du titulaire, nous pourrions distinguer deux méthodes pour initialiser ce dernier.

```
public void setTitulaire(String titulaire) {
        this.titulaire = titulaire;
    }

    public void setTitulaire(String titulaire, String adresse){
        this.adresse = adresse;
        setTitulaire(titulaire);
    }
```

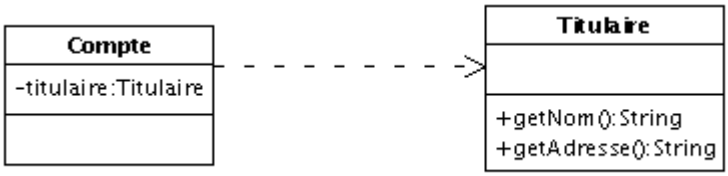
Par contre, une troisième méthode qui permettrait de n'initialiser que l'adresse sera refusée, sa signature la rendant indiscernable de la première version.

```
// surcharge illégale
    public void setTitulaire(String adresse) {
        this.adresse = adresse;
    }
```

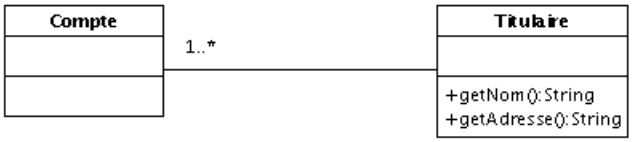
1.8 Liens et associations

En parlant de l'adresse du titulaire, il devient évident que le titulaire ne se réduit pas à une simple chaîne de caractères. Nous avons affaire à une classe à part entière. Le titulaire peut donner son nom, son adresse, énumérer ses comptes... Nous allons voir comment établir une relation entre un compte et son titulaire.

Une première relation, faible, porte le nom de *dépendance* et signifierait que la classe *Compte* dispose d'un attribut de type *Titulaire*. Dans cette optique, chaque titulaire est un moyen commode de rassembler des informations au sujet d'une personne, avec des possibilités de traitements limitées. Les titulaires n'ont pas vraiment d'existence indépendamment de leur compte.



Cette première modélisation ne tient pas compte du fait qu'un titulaire peut avoir plusieurs comptes. Une telle gestion conviendrait mieux pour une classe comme *Date*. On peut donner une meilleure approche des titulaires en établissant une association entre les classes *Compte* et *Titulaire*. L'*association* définit de manière abstraite, au niveau des classes, des *liens* existant entre les objets qu'elle permet d'instancier. Dans la réalité, un objet compte a toujours un et un seul titulaire et un même titulaire peut avoir au moins un compte, mais parfois plusieurs. Dans le diagramme de classe, l'attribut titulaire a disparu. Au niveau du code, on retrouvera une variable d'instance de type *Titulaire*, mais elle résultera du traitement de l'association (en fonction de la multiplicité affichée dans le schéma).



Le deuxième schéma, illustrant une association, permet de se rendre compte que si une variable d'instance de type *Titulaire* pourra suffire du côté du *Compte*, il faudra trouver mieux au niveau du *Titulaire*[8]. L'association définie ici est bidirectionnelle. Elle pourrait être unidirectionnelle, ce qui est plus simple à programmer, mais rend certains traitements plus difficiles, voire impossibles.

On peut raisonnablement supposer que le titulaire d'un compte ne peut pas changer. Si nous nous limitons à examiner la classe *Compte*, voici ce que deviendra notre gestion du titulaire :

```
public class Compte2 {

    private double solde;
    private Titulaire titulaire;
    private final double decouvert;


    public Compte2(Titulaire titulaire, double _decouvert) {
        decouvert = _decouvert;
        this.titulaire = titulaire;
    }

    public Compte2(Titulaire titulaire){
        this(titulaire,-1250);
    }
    ....
    public Titulaire getTitulaire() {
        // your code here
        return titulaire;
    }
```

8. Les tableaux sont peu pratiques pour gérer les associations. On préférera une collection, notion qui sera vue en détails plus tard.

Quelques exemples d'instanciation et de traitement :

```
Titulaire t1 = new Titulaire("Jacques","rue␣de␣l'Eglise");
Compte2 c1 = new Compte2(t1,-3000F);
c1.retrait(1000);
System.out.println("Solde:␣"+c1.getSolde());
System.out.println("Ce␣compte␣appartient␣à␣"
     +c1.getTitulaire().getNom());
Titulaire t2 = new Titulaire("Pierre","rue␣de␣la␣Cathédrale");
Compte2 c2 = new Compte2(t2);
System.out.println("Ce␣compte␣appartient␣à␣"
     +c2.getTitulaire().getNom());
```

1.9 Exercices

1. En reprenant l'exemple du compte, nous allons expliciter un peu la gestion du titulaire. Nous voudrions qu'il soit possible d'obtenir directement l'adresse d'un titulaire de compte et de modifier cette adresse. Il faudra pour cela définir un peu mieux que nous ne l'avons fait la classe *Titulaire*.
2. Quelques exemples de modélisation. Pour chaque description, imaginer une ou plusieurs classes, leurs responsabilités et leurs associations :
   – une compagnie aérienne propose des vols, directs ou avec escales. Le but est d'établir des réservations.
   – une bibliothèque comporte de nombreux livres. Des lecteurs viennent les réserver et les emprunter.

1.10 Remarques

Les notions d'attribut et d'opération correspondent à des concepts de modélisation. En général, un attribut anticipe la création d'une variable d'instance et une opération anticipe une méthode, variable d'instance et méthode relevant du code. L'expérience montre cependant que la correspondance n'est pas automatique.

1. un attribut dérivé détermine sa valeur en réalisant un calcul au départ de valeurs d'autres attributs. Il n'est donc pas souhaitable de lui faire correspondre une variable d'instance, qui risquerait de se désynchroniser par rapport aux attributs utilisés par le calcul.
2. une association, mono ou bidirectionnelle, se matérialise au moyen de variables d'instance qui ne dérivent pas d'attributs de la modélisation.
3. certaines méthodes, dites accesseurs, constituent un moyen de lire et modifier des attributs (méthodes `get` et `set`). Il n'est pas toujours utile de les mentionner lors de la modélisation. Un attribut public peut apparaître dans le code sous la forme d'un attribut privé doté de deux accesseurs.
4. il est plus rare qu'une opération ne soit pas implémentée par une méthode.