

Traducteur : Jérôme QUELIN

25.04.2001 - version 5.3 :

- Mise en forme du code html (titres-hx[verdana], paragraphes-p[Georgia], code-blockquote

14.07.2000 - version 5.2 :

- Corrections apportées par Jean-Pierre Vidal.

08.07.2000 - version 5.1 :

- Première publication sur eGroups.

8 : Interfaces & Classes Internes

Les interfaces et les classes internes sont des manières plus sophistiquées d'organiser et de contrôler les objets du système construit.

C++, par exemple, ne propose pas ces mécanismes, bien que le programmeur expérimenté puisse les simuler. Le fait qu'ils soient présents dans Java indique qu'ils furent considérés comme assez importants pour être intégrés directement grâce à des mots-clefs.

Dans le chapitre 7, on a vu le mot-clef **abstract**, qui permet de créer une ou plusieurs méthodes dans une classe qui n'ont pas de définition - on fournit une partie de l'interface sans l'implémentation correspondante, qui est créée par ses héritiers. Le mot-clef **interface** produit une classe complètement abstraite, qui ne fournit absolument aucune implémentation. Nous verrons qu'une **interface** est un peu plus qu'une classe abstraite poussée à l'extrême, puisqu'elle permet d'implémenter « l'héritage multiple » du C++ en créant une classe qui peut être transtypée en plus d'un type de base.

Les classes internes ressemblent au premier abord à un simple mécanisme de dissimulation de code : on crée une classe à l'intérieur d'autres classes. Cependant, les classes internes font plus que cela - elles connaissent et peuvent communiquer avec la classe principale - ; sans compter que le code produit en utilisant les classes internes est plus élégant et compréhensible, bien que ce soit un concept nouveau pour beaucoup. Cela prend un certain temps avant d'intégrer les classes internes dans la conception.

Interfaces

Le mot-clef **interface** pousse le concept **abstract** un cran plus loin. On peut y penser comme à une classe « purement » **abstract**. Il permet au créateur d'établir la forme qu'aura la classe : les noms des méthodes, les listes d'arguments et les types de retour, mais pas les corps des méthodes. Une **interface** peut aussi contenir des données membres, mais elles seront implicitement **static** et **final**. Une interface fournit un patron pour la classe, mais aucune implémentation.

Une **interface** déclare : « Voici ce à quoi ressemblera toutes les classes qui *implémenteront* cette interface ». Ainsi, tout code utilisant une **interface** particulière sait quelles méthodes peuvent être appelées pour cette **interface**, et c'est tout. Une **interface** est donc utilisée pour établir un « protocole » entre les classes (certains langages de programmation orientés objets ont un mot-clef *protocol* pour réaliser la même chose).

Pour créer une **interface**, il faut utiliser le mot-clef **interface** à la place du mot-clef **class**. Comme pour une classe, on peut ajouter le mot-clef **public** devant le mot-clef **interface** (mais seulement si l'**interface** est définie

dans un fichier du même nom) ou ne rien mettre pour lui donner le statut « amical » afin qu'elle ne soit utilisable que dans le même package.

Le mot-clef **implements** permet de rendre une classe conforme à une **interface** particulière (ou à un groupe d'**interfaces**). Il dit en gros : « **L'interface** spécifie ce à quoi la classe ressemble, mais maintenant on va spécifier comment cela *fonctionne* ». Sinon, cela s'apparente à de l'héritage. Le diagramme des instruments de musique suivant le montre :



Une fois une **interface** implémentée, cette implémentation devient une classe ordinaire qui peut être étendue d'une façon tout à fait classique.

On peut choisir de déclarer explicitement les méthodes d'une **interface** comme **public**. Mais elles sont **public** même sans le préciser. C'est pourquoi il faut définir les méthodes d'une **interface** comme **public** quand on implémente une **interface**. Autrement elles sont « amicales » par défaut, impliquant une réduction de l'accessibilité d'une méthode durant l'héritage, ce qui est interdit par le compilateur Java.

On peut le voir dans cette version modifiée de l'exemple **Instrument**. Notons que chaque méthode de l'**interface** n'est strictement qu'une déclaration, la seule chose que le compilateur permette. De plus, aucune des méthodes d'**Instrument** n'est déclarée comme **public**, mais elles le sont automatiquement.

```
//: c08:music5:Music5.java  
// Interfaces.  
import java.util.*;  
  
interface Instrument {
```

```
// Constante compilée :  
  
int i = 5; // static & final  
  
// Définitions de méthodes interdites :  
  
void play(); // Automatiquement public  
  
String what();  
  
void adjust();  
  
}  
  
class Wind implements Instrument {  
  
    public void play() {  
  
        System.out.println("Wind.play()");  
  
    }  
  
    public String what() { return "Wind"; }  
  
    public void adjust() {}  
  
}  
  
class Percussion implements Instrument {  
  
    public void play() {  
  
        System.out.println("Percussion.play()");  
  
    }  
  
    public String what() { return "Percussion"; }  
  
    public void adjust() {}  
  
}  
  
class Stringed implements Instrument {  
  
    public void play() {  
  
        System.out.println("Stringed.play()");  
  
    }  
  
    public String what() { return "Stringed"; }  
  
    public void adjust() {}  
  
}  
  
class Brass extends Wind {  
  
    public void play() {
```

```
        System.out.println("Brass.play()");
    }

    public void adjust() {
        System.out.println("Brass.adjust()");
    }
}

class Woodwind extends Wind {
    public void play() {
        System.out.println("Woodwind.play()");
    }

    public String what() { return "Woodwind"; }
}

public class Music5 {
    // Le type n'est pas important, donc les nouveaux
    // types ajoutés au système marchent sans problème :
    static void tune(Instrument i) {
        // ...
        i.play();
    }

    static void tuneAll(Instrument[] e) {
        for(int i = 0; i < e.length; i++)
            tune(e[i]);
    }

    public static void main(String[] args) {
        Instrument[] orchestra = new Instrument[5];

        int i = 0;

        // Transtypage ascendant durant le stockage dans le tableau :
        orchestra[i++] = new Wind();
        orchestra[i++] = new Percussion();
        orchestra[i++] = new Stringed();
        orchestra[i++] = new Brass();
        orchestra[i++] = new Woodwind();
    }
}
```

```

        tuneAll(orchestra);
    }
} ///:~

```

Le reste du code ne change pas. Cela ne gêne pas si on transtype vers une classe « normale » appelée **Instrument**, une classe **abstract** appelée **Instrument**, ou une **interface** appelée **Instrument**. Le comportement reste le même. En fait, on peut voir dans la méthode **tune()** qu'on ne peut savoir si **Instrument** est une classe « normale », une classe **abstract**, ou une **interface**. Et c'est bien le but recherché : chaque approche donne au programmeur un contrôle différent sur la manière dont les objets sont créés et utilisés.

« Héritage multiple » en Java

Une **interface** n'est pas simplement une forme « plus pure » d'une classe **abstract**. Elle a un but plus important que cela. Puisqu'une **interface** ne dispose d'aucune implémentation - autrement dit, aucun stockage n'est associé à une **interface** -, rien n'empêche de combiner plusieurs **interfaces**. Ceci est intéressant car certaines fois on a la relation "Un **x** est un **a** et un **b** et un **c**". En C++, le fait de combiner les interfaces de plusieurs classes est appelé *héritage multiple*, et entraîne une lourde charge du fait que chaque classe peut avoir sa propre implémentation. En Java, on peut réaliser la même chose, mais une seule classe peut avoir une implémentation, donc les problèmes rencontrés en C++ n'apparaissent pas en Java lorsqu'on combine les interfaces multiples :



Dans une classe dérivée, on n'est pas forcé d'avoir une classe de base qui soit **abstract** ou « concrète » (i.e. sans méthode **abstract**). Si une classe hérite d'une classe qui n'est pas une **interface**, elle ne peut dériver que de cette seule classe. Tous les autres types de base doivent être des **interfaces**. On place les noms des interfaces après le mot-clef **implements** en les séparant par des virgules. On peut spécifier autant d'**interfaces** qu'on veut - chacune devient un type indépendant vers lequel on peut transtyper. L'exemple suivant montre une classe concrète combinée à plusieurs **interfaces** pour produire une nouvelle classe :

```

//: c08:Adventure.java
// Interfaces multiples.

import java.util.*;

interface CanFight {
    void fight();
}

```

```
interface CanSwim {  
    void swim();  
}  
  
interface CanFly {  
    void fly();  
}  
  
class ActionCharacter {  
    public void fight() {}  
}  
  
class Hero extends ActionCharacter  
    implements CanFight, CanSwim, CanFly {  
    public void swim() {}  
    public void fly() {}  
}  
  
public class Adventure {  
    static void t(CanFight x) { x.fight(); }  
    static void u(CanSwim x) { x.swim(); }  
    static void v(CanFly x) { x.fly(); }  
    static void w(ActionCharacter x) { x.fight(); }  
    public static void main(String[] args) {  
        Hero h = new Hero();  
        t(h); // Le traite comme un CanFight  
        u(h); // Le traite comme un CanSwim  
        v(h); // Le traite comme un CanFly  
        w(h); // Le traite comme un ActionCharacter  
    }  
} ///:~
```

Ici, **Hero** combine la classe concrète **ActionCharacter** avec les interfaces **CanFight**, **CanSwim** et **CanFly**. Quand on combine une classe concrète avec des interfaces de cette manière, la classe concrète doit être spécifiée en premier, avant les interfaces (autrement le compilateur génère une erreur).

Notons que la signature de **fight()** est la même dans l'**interface CanFight** et dans la classe **ActionCharacter**, et que **Hero** ne fournit *pas* de définition pour **fight()**. On peut hériter d'une **interface** (comme on va le voir bientôt), mais dans ce cas on a une autre **interface**. Si on veut créer un objet de ce nouveau type, ce doit être une classe implémentant toutes les définitions. Bien que la classe **Hero** ne fournisse pas explicitement une définition pour **fight()**, la définition est fournie par **ActionCharacter**, donc héritée par **Hero** et il est ainsi possible de créer des objets **Hero**.

Dans la classe **Adventure**, on peut voir quatre méthodes prenant les diverses interfaces et la classe concrète en argument. Quand un objet **Hero** est créé, il peut être utilisé dans chacune de ces méthodes, ce qui veut dire qu'il est transtypé tour à tour dans chaque **interface**. De la façon dont cela est conçu en Java, cela fonctionne sans problème et sans effort supplémentaire de la part du programmeur.

L'intérêt principal des interfaces est démontré dans l'exemple précédent : être capable de transtyper vers plus d'un type de base. Cependant, une seconde raison, la même que pour les classes de base **abstract**, plaide pour l'utilisation des interfaces : empêcher le programmeur client de créer un objet de cette classe et spécifier qu'il ne s'agit que d'une interface. Cela soulève une question : faut-il utiliser une **interface** ou une classe **abstract** ? Une **interface** apporte les bénéfices d'une classe **abstract** *et* les bénéfices d'une **interface**, donc s'il est possible de créer la classe de base sans définir de méthodes ou de données membres, il faut toujours préférer les **interfaces** aux classes **abstract**. En fait, si on sait qu'un type sera amené à être dérivé, il faut le créer d'emblée comme une **interface**, et ne le changer en classe **abstract**, voire en classe concrète, que si on est forcé d'y placer des définitions de méthodes ou des données membres.

Combinaison d'interfaces et collisions de noms

On peut rencontrer un problème lorsqu'on implémente plusieurs interfaces. Dans l'exemple précédent, **CanFight** et **ActionCharacter** ont tous les deux une méthode **void fight()** identique. Cela ne pose pas de problèmes parce que la méthode est identique dans les deux cas, mais que se passe-t-il lorsque ce n'est pas le cas ? Voici un exemple :

```
/// c08:InterfaceCollision.java
```

```
interface I1 { void f(); }

interface I2 { int f(int i); }

interface I3 { int f(); }

class C { public int f() { return 1; } }

class C2 implements I1, I2 {
    public void f() {}

    public int f(int i) { return 1; } // surchargée
}

class C3 extends C implements I2 {
    public int f(int i) { return 1; } // surchargée
```

```

}

class C4 extends C implements I3 {
    // Identique, pas de problème :
    public int f() { return 1; }
}

// Les méthodes diffèrent seulement par le type de retour :
//! class C5 extends C implements I1 {}
//! interface I4 extends I1, I3 {} ///:~

```

Les difficultés surviennent parce que la redéfinition, l'implémentation et la surcharge sont toutes les trois utilisées ensemble, et que les fonctions surchargées ne peuvent différer seulement par leur type de retour. Quand les deux dernières lignes sont décommentées, le message d'erreur est explicite :

```

InterfaceCollision.java:23: f() in C cannot
implement f() in I1; attempting to use
incompatible return type
found   : int
required: void
InterfaceCollision.java:24: interfaces I3 and I1 are incompatible; both define f
(), but with different return type

```

De toutes façons, utiliser les mêmes noms de méthode dans différentes interfaces destinées à être combinées affecte la compréhension du code. Il faut donc l'éviter autant que faire se peut.

Etendre une interface avec l'héritage

On peut facilement ajouter de nouvelles déclarations de méthodes à une **interface** en la dérivant, de même qu'on peut combiner plusieurs **interfaces** dans une nouvelle **interface** grâce à l'héritage. Dans les deux cas on a une nouvelle **interface**, comme dans l'exemple suivant :

```

//: c08:HorrorShow.java
// Extension d'une interface grâce à l'héritage.

interface Monster {
    void menace();
}

```



```
interface DangerousMonster extends Monster {
    void destroy();
}

interface Lethal {
    void kill();
}

class DragonZilla implements DangerousMonster {
    public void menace() {}
    public void destroy() {}
}

interface Vampire
    extends DangerousMonster, Lethal {
    void drinkBlood();
}

class HorrorShow {
    static void u(Monster b) { b.menace(); }
    static void v(DangerousMonster d) {
        d.menace();
        d.destroy();
    }
    public static void main(String[] args) {
        DragonZilla if2 = new DragonZilla();
        u(if2);
        v(if2);
    }
} ///:~
```

DangerousMonster est une simple extension de **Monster** qui fournit une nouvelle **interface**. Elle est implémentée dans **DragonZilla**.

La syntaxe utilisée dans **Vampire** n'est valide *que* lorsqu'on dérive des interfaces. Normalement, on ne peut utiliser **extends** qu'avec une seule classe, mais comme une **interface** peut être constituée de plusieurs autres interfaces, **extends** peut se référer à plusieurs interfaces de base lorsqu'on construit une nouvelle **interface**. Comme on peut le voir, les noms d'**interface** sont simplement séparées par des virgules.

Groupes de constantes

Puisque toutes les données membres d'une **interface** sont automatiquement **static** et **final**, une **interface** est un outil pratique pour créer des groupes de constantes, un peu comme avec le **enum** du C ou du C++. Par exemple :

```
///  
// Utiliser les interfaces pour créer des groupes de constantes.  
  
package c08;  
  
public interface Months {  
    int  
  
    JANUARY = 1, FEBRUARY = 2, MARCH = 3,  
    APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,  
    AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,  
    NOVEMBER = 11, DECEMBER = 12;  
} ///:~
```

Notons au passage l'utilisation des conventions de style Java pour les champs **static final** initialisés par des constantes : rien que des majuscules (avec des underscores pour séparer les mots à l'intérieur d'un identifiant).

Les données membres d'une **interface** sont automatiquement **public**, il n'est donc pas nécessaire de le spécifier.

Maintenant on peut utiliser les constantes à l'extérieur du package en important **c08.*** ou **c08.Months** de la même manière qu'on le ferait avec n'importe quel autre package, et référencer les valeurs avec des expressions comme **Months.JANUARY**. Bien sûr, on ne récupère qu'un **int**, il n'y a donc pas de vérification additionnelle de type comme celle dont dispose l'**enum** du C++, mais cette technique (couramment utilisée) reste tout de même une grosse amélioration comparée aux nombres codés en dur dans les programmes (appelés « nombres magiques » et produisant un code pour le moins difficile à maintenir).

Si on veut une vérification additionnelle de type, on peut construire une classe de la manière suivante [\[38\]](#):

```
///  
// Un système d'énumération plus robuste.  
  
package c08;  
  
public final class Month2 {
```

```

private String name;

private Month2(String nm) { name = nm; }

public String toString() { return name; }

public final static Month2

    JAN = new Month2("January"),
    FEB = new Month2("February"),
    MAR = new Month2("March"),
    APR = new Month2("April"),
    MAY = new Month2("May"),
    JUN = new Month2("June"),
    JUL = new Month2("July"),
    AUG = new Month2("August"),
    SEP = new Month2("September"),
    OCT = new Month2("October"),
    NOV = new Month2("November"),
    DEC = new Month2("December");

public final static Month2[] month = {

    JAN, JAN, FEB, MAR, APR, MAY, JUN,

    JUL, AUG, SEP, OCT, NOV, DEC

};

public static void main(String[] args) {

    Month2 m = Month2.JAN;

    System.out.println(m);

    m = Month2.month[12];

    System.out.println(m);

    System.out.println(m == Month2.DEC);

    System.out.println(m.equals(Month2.DEC));

}

} ///:~

```

Cette classe est appelée **Month2**, puisque **Month** existe déjà dans la bibliothèque Java standard. C'est une classe **final** avec un constructeur **private** afin que personne ne puisse la dériver ou en faire une instance. Les seules instances sont celles **static final** créées dans la classe elle-même : **JAN**, **FEB**, **MAR**, etc. Ces objets sont aussi utilisés dans le tableau **month**, qui permet de choisir les mois par leur index au lieu de leur nom (notez le premier **JAN** dans le tableau pour introduire un déplacement supplémentaire de un, afin que Décembre soit le mois numéro 12). Dans **main()** on dispose de la vérification additionnelle de type : **m** est un objet **Month2** et ne

peut donc se voir assigné qu'un **Month2**. L'exemple précédent (**Months.java**) ne fournissait que des valeurs **int**, et donc une variable **int** destinée à représenter un mois pouvait en fait recevoir n'importe quelle valeur entière, ce qui n'était pas très sûr.

Cette approche nous permet aussi d'utiliser indifféremment **==** ou **equals()**, ainsi que le montre la fin de **main()**.

Initialisation des données membres des interfaces

Les champs définis dans les interfaces sont automatiquement **static** et **final**. Ils ne peuvent être des « finals vides », mais peuvent être initialisés avec des expressions non constantes. Par exemple :

```
//: c08:RandVals.java
// Initialisation de champs d'interface
// avec des valeurs non-constantes.
import java.util.*;

public interface RandVals {
    int rint = (int)(Math.random() * 10);
    long rlong = (long)(Math.random() * 10);
    float rfloat = (float)(Math.random() * 10);
    double rdouble = Math.random() * 10;
} ///:~
```

Comme les champs sont **static**, ils sont initialisés quand la classe est chargée pour la première fois, ce qui arrive quand n'importe lequel des champs est accédé pour la première fois. Voici un simple test :

```
//: c08:TestRandVals.java

public class TestRandVals {
    public static void main(String[] args) {
        System.out.println(RandVals.rint);
        System.out.println(RandVals.rlong);
        System.out.println(RandVals.rfloat);
        System.out.println(RandVals.rdouble);
    }
} ///:~
```

Les données membres, bien sûr, ne font pas partie de l'interface mais sont stockées dans la zone de stockage **static** de cette interface.

Interfaces imbriquées

[39] Les interfaces peuvent être imbriquées dans des classes ou à l'intérieur d'autres interfaces. Ceci révèle nombre de fonctionnalités intéressantes :

```
///  
c08:NestingInterfaces.java  
  
class A {  
    interface B {  
        void f();  
    }  
  
    public class BImp implements B {  
        public void f() {}  
    }  
  
    private class BImp2 implements B {  
        public void f() {}  
    }  
  
    public interface C {  
        void f();  
    }  
  
    class CImp implements C {  
        public void f() {}  
    }  
  
    private class CImp2 implements C {  
        public void f() {}  
    }  
  
    private interface D {  
        void f();  
    }  
  
    private class DImp implements D {  
        public void f() {}  
    }  
  
    public class DImp2 implements D {  
        public void f() {}  
    }  
}
```

```

    public D getD() { return new DImp2(); }

    private D dRef;

    public void receiveD(D d) {
        dRef = d;
        dRef.f();
    }
}

interface E {
    interface G {
        void f();
    }

    // « public » est redondant :
    public interface H {
        void f();
    }

    void g();

    // Ne peut pas être private dans une interface :
    //! private interface I {}
}

public class NestingInterfaces {
    public class BImp implements A.B {
        public void f() {}
    }

    class CImp implements A.C {
        public void f() {}
    }

    // Ne peut pas implémenter une interface private sauf
    // à l'intérieur de la classe définissant cette interface :
    //! class DImp implements A.D {
    //!     public void f() {}
    //! }

    class EImp implements E {

```

```

        public void g() {}
    }

    class EGImp implements E.G {

        public void f() {}
    }

    class EImp2 implements E {

        public void g() {}

        class EG implements E.G {

            public void f() {}
        }
    }

    public static void main(String[] args) {

        A a = new A();

        // Ne peut accéder à A.D :
        //! A.D ad = a.getD();

        // Ne renvoie qu'un A.D :
        //! A.DImp2 di2 = a.getD();

        // Ne peut accéder à un membre de l'interface :
        //! a.getD().f();

        // Seul un autre A peut faire quelque chose avec getD() :

        A a2 = new A();

        a2.receiveD(a.getD());
    }
} ///:~

```

La syntaxe permettant d'imbriquer une interface à l'intérieur d'une classe est relativement évidente ; et comme les interfaces non imbriquées, elles peuvent avoir une visibilité **public** ou « amicale ». On peut aussi constater que les interfaces **public** et « amicales » peuvent être implémentées dans des classes imbriquées **public**, « amicales » ou **private**.

Une nouvelle astuce consiste à rendre les interfaces **private** comme **A.D** (la même syntaxe est utilisée pour la qualification des interfaces imbriquées et pour les classes imbriquées). A quoi sert une interface imbriquée **private** ? On pourrait penser qu'elle ne peut être implémentée que comme une classe **private** imbriquée comme **DImp**, mais **A.DImp2** montre qu'elle peut aussi être implémentée dans une classe **public**. Cependant, **A.DImp2** ne peut être utilisée que comme elle-même : on ne peut mentionner le fait qu'elle implémente l'interface **private**, et donc implémenter une interface **private** est une manière de forcer la définition des méthodes de cette interface sans ajouter aucune information de type (c'est à dire, sans autoriser de transtypage ascendant).

La méthode **getD()** se trouve quant à elle dans une impasse du fait de l'interface **private** : c'est une méthode **public** qui renvoie une référence à une interface **private**. Que peut-on faire avec la valeur de retour de cette méthode ? Dans **main()**, on peut voir plusieurs tentatives pour utiliser cette valeur de retour, qui échouent toutes. La seule solution possible est lorsque la valeur de retour est gérée par un objet qui a la permission de l'utiliser - dans ce cas, un objet **A**, via la méthode **receiveD()**.

L'interface **E** montre que les interfaces peuvent être imbriquées les unes dans les autres. Cependant, les règles portant sur les interfaces - en particulier celle stipulant que tous les éléments doivent être **public** - sont strictement appliquées, donc une interface imbriquée à l'intérieur d'une autre interface est automatiquement **public** et ne peut être déclarée **private**.

NestingInterfaces montre les différentes manières dont les interfaces imbriquées peuvent être implémentées. En particulier, il est bon de noter que lorsqu'on implémente une interface, on n'est pas obligé d'en implémenter les interfaces imbriquées. De plus, les interfaces **private** ne peuvent être implémentées en dehors de leur classe de définition.

On peut penser que ces fonctionnalités n'ont été introduites que pour assurer une cohérence syntaxique, mais j'ai remarqué qu'une fois qu'une fonctionnalité est connue, on découvre souvent des endroits où elle se révèle utile.

Classes internes

Il est possible de placer la définition d'une classe à l'intérieur de la définition d'une autre classe. C'est ce qu'on appelle une classe interne. Les classes internes sont une fonctionnalité importante du langage car elles permettent de grouper les classes qui sont logiquement rattachées entre elles, et de contrôler la visibilité de l'une à partir de l'autre. Cependant, il est important de comprendre que le mécanisme des classes internes est complètement différent de celui de la composition.

Souvent, lorsqu'on en entend parler pour la première fois, l'intérêt des classes internes n'est pas immédiatement évident. A la fin de cette section, après avoir discuté de la syntaxe et de la sémantique des classes internes, vous trouverez des exemples qui devraient clairement montrer les bénéfices des classes internes.

Une classe interne est créée comme on pouvait s'y attendre - en plaçant la définition de la classe à l'intérieur d'une autre classe :

```
///  
// Création de classes internes.  
  
public class Parcel1 {  
    class Contents {  
        private int i = 11;  
        public int value() { return i; }  
    }  
    class Destination {  
        private String label;
```



```

        Destination(String whereTo) {
            label = whereTo;
        }

        String readLabel() { return label; }
    }

    // L'utilisation d'une classe interne ressemble à
    // l'utilisation de n'importe quelle autre classe depuis Parcel1 :
    public void ship(String dest) {
        Contents c = new Contents();
        Destination d = new Destination(dest);
        System.out.println(d.readLabel());
    }

    public static void main(String[] args) {
        Parcel1 p = new Parcel1();
        p.ship("Tanzania");
    }
} ///:~

```

Les classes internes, quand elles sont utilisées dans **ship()**, ressemblent à n'importe quelle autre classe. La seule différence en est que les noms sont imbriqués dans **Parcel1**. Mais nous allons voir dans un moment que ce n'est pas la seule différence.

Plus généralement, une classe externe peut définir une méthode qui renvoie une référence à une classe interne, comme ceci :

```

///: c08:Parcel2.java
// Renvoyer une référence à une classe interne.

```

```

public class Parcel2 {
    class Contents {
        private int i = 11;
        public int value() { return i; }
    }

    class Destination {
        private String label;

        Destination(String whereTo) {
            label = whereTo;
        }
    }
}

```

```

    }

    String readLabel() { return label; }
}

public Destination to(String s) {
    return new Destination(s);
}

public Contents cont() {
    return new Contents();
}

public void ship(String dest) {
    Contents c = cont();
    Destination d = to(dest);
    System.out.println(d.readLabel());
}

public static void main(String[] args) {
    Parcel2 p = new Parcel2();
    p.ship("Tanzania");
    Parcel2 q = new Parcel2();
    // Définition de références sur des classes internes :
    Parcel2.Contents c = q.cont();
    Parcel2.Destination d = q.to("Borneo");
}
} ///:~

```

Si on veut créer un objet de la classe interne ailleurs que dans une méthode **non-*static*** de la classe externe, il faut spécifier le type de cet objet comme *NomDeClasseExterne.NomDeClasseInterne*, comme on peut le voir dans **main()**.

Classes internes et transtypage ascendant

Jusqu'à présent, les classes internes ne semblent pas tellement intéressantes. Après tout, si le but recherché est le camouflage, Java propose déjà un très bon mécanisme pour cela - il suffit de rendre la classe « amicale » (visible seulement depuis un certain package) plutôt que de la déclarer comme une classe interne.

Cependant, les classes internes prennent de l'intérêt lorsqu'on transtype vers une classe de base, et en particulier vers une **interface** (produire une référence vers une interface depuis un objet l'implémentant revient à transtyper vers une classe de base). En effet la classe interne - l'implémentation de **l'interface** - est complètement masquée et indisponible pour tout le monde, ce qui est pratique pour cacher l'implémentation. La seule chose qu'on récupère est une référence sur la classe de base ou **l'interface**.

Tout d'abord, les interfaces sont définies dans leurs propres fichiers afin de pouvoir être utilisées dans tous les exemples :

```
//: c08:Destination.java

public interface Destination {

    String readLabel();

} ///:~
```

```
//: c08:Contents.java

public interface Contents {

    int value();

} ///:~
```

Maintenant **Contents** et **Destination** sont des interfaces disponibles pour le programmeur client (une **interface** déclare automatiquement tous ses membres comme **public**).

Quand on récupère une référence sur la classe de base ou **l'interface**, il est possible qu'on ne puisse même pas en découvrir le type exact, comme on peut le voir dans le code suivant :

```
//: c08:Parcel3.java

// Renvoyer une référence sur une classe interne.

public class Parcel3 {

    private class PContents implements Contents {

        private int i = 11;

        public int value() { return i; }

    }

    protected class PDestination

        implements Destination {

        private String label;

        private PDestination(String whereTo) {

            label = whereTo;

        }

        public String readLabel() { return label; }

    }

    public Destination dest(String s) {
```

```

        return new PDestination(s);
    }

    public Contents cont() {
        return new PContents();
    }
}

class Test {
    public static void main(String[] args) {
        Parcel3 p = new Parcel3();
        Contents c = p.cont();
        Destination d = p.dest("Tanzania");
        // Illégal -- ne peut accéder à une classe private :
        ///! Parcel3.PContents pc = p.new PContents();
    }
} ///:~

```

Notez que puisque **main()** se trouve dans **Test**, pour lancer ce programme il ne faut pas exécuter **Parcel3**, mais :

```
java Test
```

Dans cet exemple, **main()** doit être dans une classe séparée afin de démontrer le caractère **private** de la classe interne **PContents**.

Dans **Parcel3**, de nouvelles particularités ont été ajoutées : la classe interne **PContents** est **private** afin que seule **Parcel3** puisse y accéder. **PDestination** est **protected**, afin que seules **Parcel3**, les classes du packages de **Parcel3** (puisque **protected** fournit aussi un accès package - c'est à dire que **protected** est « amical »), et les héritiers de **Parcel3** puissent accéder à **PDestination**. Cela signifie que le programmeur client n'a qu'une connaissance et des accès restreints à ces membres. En fait, on ne peut faire de transtypage descendant vers une classe interne **private** (ou une classe interne **protected** à moins d'être un héritier), parce qu'on ne peut accéder à son nom, comme on peut le voir dans la classe **Test**. La classe interne **private** fournit donc un moyen pour le concepteur de la classe d'interdire tout code testant le type et de cacher complètement les détails de l'implémentation. De plus, l'extension d'une **interface** est inutile du point de vue du programmeur client puisqu'il ne peut accéder à aucune méthode additionnelle ne faisant pas partie de l'**interface public** de la classe. Cela permet aussi au compilateur Java de générer du code plus efficace.

Les classes normales (non internes) ne peuvent être déclarées **private** ou **protected** - uniquement **public** ou « amicales ».

Classes internes définies dans des méthodes et autres portées

Ce qu'on a pu voir jusqu'à présent constitue l'utilisation typique des classes internes. En général, le code

impliquant des classes internes que vous serez amené à lire et à écrire ne mettra en oeuvre que des classes internes « régulières », et sera simple à comprendre. Cependant, le support des classes internes est relativement complet et il existe de nombreuses autres manières, plus obscures, de les utiliser si on le souhaite : les classes internes peuvent être créées à l'intérieur d'une méthode ou même d'une portée arbitraire. Deux raisons possibles à cela :

1. Comme montré précédemment, on implémente une interface d'un certain type afin de pouvoir créer et renvoyer une référence.
2. On résout un problème compliqué pour lequel la création d'une classe aiderait grandement, mais on ne veut pas la rendre publiquement accessible.

Dans les exemples suivants, le code précédent est modifié afin d'utiliser :

1. Une classe définie dans une méthode
2. Une classe définie dans une portée à l'intérieur d'une méthode
3. Une classe anonyme implémentant une interface
4. Une classe anonyme étendant une classe qui dispose d'un constructeur autre que le constructeur par défaut
5. Une classe anonyme réalisant des initialisations de champs
6. Une classe anonyme qui se construit en initialisant des instances (les classes internes anonymes ne peuvent avoir de constructeurs)

Bien que ce soit une classe ordinaire avec une implémentation, **Wrapping** est aussi utilisée comme une « interface » commune pour ses classes dérivées :

```
///  
c08:Wrapping.java  
  
public class Wrapping {  
    private int i;  
  
    public Wrapping(int x) { i = x; }  
  
    public int value() { return i; }  
  
} ///:~
```

Notez que **Wrapping** dispose d'un constructeur requérant un argument, afin de rendre les choses un peu plus intéressantes.

Le premier exemple montre la création d'une classe entière dans la portée d'une méthode (au lieu de la portée d'une autre classe) :

```
///  
c08:Parcel4.java  
  
// Définition d'une classe à l'intérieur d'une méthode.  
  
public class Parcel4 {  
    public Destination dest(String s) {  
        class PDestination  
            implements Destination {
```

```

    private String label;

    private PDestination(String whereTo) {
        label = whereTo;
    }

    public String readLabel() { return label; }
}

return new PDestination(s);
}

public static void main(String[] args) {
    Parcel4 p = new Parcel4();
    Destination d = p.dest("Tanzania");
}
} ///:~

```

La classe **PDestination** appartient à **dest()** plutôt qu'à **Parcel4** (notez aussi qu'on peut utiliser l'identifiant **PDestination** pour une classe interne à l'intérieur de chaque classe du même sous-répertoire sans collision de nom). Cependant, **PDestination** ne peut être accédée en dehors de **dest()**. Notez le transtypage ascendant réalisé par l'instruction de retour - **dest()** ne renvoie qu'une référence à **Destination**, la classe de base. Bien sûr, le fait que le nom de la classe **PDestination** soit placé à l'intérieur de **dest()** ne veut pas dire que **PDestination** n'est pas un objet valide une fois sorti de **dest()**.

L'exemple suivant montre comment on peut imbriquer une classe interne à l'intérieur de n'importe quelle portée :

```

///: c08:Parcel5.java

// Définition d'une classe à l'intérieur d'une portée quelconque.

public class Parcel5 {
    private void internalTracking(boolean b) {
        if(b) {
            class TrackingSlip {
                private String id;

                TrackingSlip(String s) {
                    id = s;
                }

                String getSlip() { return id; }
            }
        }
    }
}

```

```

        TrackingSlip ts = new TrackingSlip("slip");

        String s = ts.getSlip();
    }

    // Utilisation impossible ici ! En dehors de la portée :
    //! TrackingSlip ts = new TrackingSlip("x");
}

public void track() { internalTracking(true); }

public static void main(String[] args) {

    Parcel5 p = new Parcel5();

    p.track();

}

} ///:~

```

La classe **TrackingSlip** est définie dans la portée de l'instruction **if**. Cela ne veut pas dire que la classe est créée conditionnellement - elle est compilée avec tout le reste. Cependant, elle n'est pas accessible en dehors de la portée dans laquelle elle est définie. Mis à part cette restriction, elle ressemble à n'importe quelle autre classe ordinaire.

Classes internes anonymes

L'exemple suivant semble relativement bizarre :

```

///: c08:Parcel6.java

// Une méthode qui renvoie une classe interne anonyme.

public class Parcel6 {

    public Contents cont() {

        return new Contents() {

            private int i = 11;

            public int value() { return i; }

        }; // Point-virgule requis dans ce cas
    }

    public static void main(String[] args) {

        Parcel6 p = new Parcel6();

        Contents c = p.cont();

    }

} ///:~

```

La méthode **cont()** combine la création d'une valeur de retour avec la définition de la classe de cette valeur de retour ! De plus, la classe est anonyme - elle n'a pas de nom. Pour compliquer le tout, il semble qu'on commence par créer un objet **Contents** :

```
return new Contents()
```

Mais alors, avant de terminer l'instruction par un point-virgule, on dit : « Eh, je crois que je vais insérer une définition de classe » :

```
return new Contents() {  
  
    private int i = 11;  
  
    public int value() { return i; }  
  
};
```

Cette étrange syntaxe veut dire : « Crée un objet d'une classe anonyme dérivée de **Contents** ». La référence renvoyée par l'expression **new** est automatiquement transtypée vers une référence **Contents**. La syntaxe d'une classe interne anonyme est un raccourci pour :

```
class MyContents implements Contents {  
  
    private int i = 11;  
  
    public int value() { return i; }  
  
}  
  
return new MyContents();
```

Dans la classe interne anonyme, **Contents** est créée avec un constructeur par défaut. Le code suivant montre ce qu'il faut faire dans le cas où la classe de base dispose d'un constructeur requérant un argument :

```
///  
// c08:Parcel7.java  
// Une classe interne anonyme qui appelle  
// le constructeur de la classe de base.  
  
public class Parcel7 {  
  
    public Wrapping wrap(int x) {  
  
        // Appel du constructeur de la classe de base :  
  
        return new Wrapping(x) {  
  
            public int value() {  
  
                return super.value() * 47;  
  
            }  
  
        }  
  
    }  
  
}
```



```

    }

    }; // Point-virgule requis
}

public static void main(String[] args) {

    Parcel7 p = new Parcel7();

    Wrapping w = p.wrap(10);

}

} ///:~

```

Autrement dit, on passe simplement l'argument approprié au constructeur de la classe de base, vu ici comme le **x** utilisé dans **new Wrapping(x)**. Une classe anonyme ne peut avoir de constructeur dans lequel on appellerait normalement **super()**.

Dans les deux exemples précédents, le point-virgule ne marque pas la fin du corps de la classe (comme il le fait en C++). Il marque la fin de l'expression qui se trouve contenir la définition de la classe anonyme. Son utilisation est donc similaire à celle que l'on retrouve partout ailleurs.

Que se passe-t-il lorsque certaines initialisations sont nécessaires pour un objet d'une classe interne anonyme ? Puisqu'elle est anonyme, on ne peut donner de nom au constructeur - et on ne peut donc avoir de constructeur. On peut néanmoins réaliser des initialisations lors de la définition des données membres :

```

///: c08:Parcel8.java

// Une classe interne anonyme qui réalise
// des initialisations. Une version plus courte
// de Parcel5.java.

public class Parcel8 {

    // L'argument doit être final pour être utilisé
    // la classe interne anonyme :

    public Destination dest(final String dest) {

        return new Destination() {

            private String label = dest;

            public String readLabel() { return label; }

        };

    }

    public static void main(String[] args) {

        Parcel8 p = new Parcel8();

        Destination d = p.dest("Tanzania");
    }
}

```

```

    }
} ///:~

```

Si on définit une classe interne anonyme et qu'on veut utiliser un objet défini en dehors de la classe interne anonyme, le compilateur requiert que l'objet extérieur soit **final**. C'est pourquoi l'argument de **dest()** est **final**. Si le mot-clef est omis, le compilateur générera un message d'erreur.

Tant qu'on se contente d'assigner un champ, l'approche précédente est suffisante. Mais comment faire si on a besoin de réaliser plusieurs actions comme un constructeur peut être amené à le faire ? Avec l'*initialisation d'instances*, on peut, dans la pratique, créer un constructeur pour une classe interne anonyme :

```

///: c08:Parcel9.java

// Utilisation des « initialisations d'instances » pour
// réaliser la construction d'une classe interne anonyme.

public class Parcel9 {
    public Destination
    dest(final String dest, final float price) {
        return new Destination() {
            private int cost;

            // Initialisation d'instance pour chaque objet :
            {
                cost = Math.round(price);
                if(cost > 100)
                    System.out.println("Over budget!");
            }

            private String label = dest;
            public String readLabel() { return label; }
        };
    }

    public static void main(String[] args) {
        Parcel9 p = new Parcel9();
        Destination d = p.dest("Tanzania", 101.395F);
    }
} ///:~

```

A l'intérieur de l'initialisateur d'instance on peut voir du code pouvant ne pas être exécuté comme partie d'un

initialisateur de champ (l'instruction **if**). Dans la pratique, donc, un initialisateur d'instance est un constructeur pour une classe interne anonyme. Bien sûr, ce mécanisme est limité ; on ne peut surcharger les initialisateurs d'instance et donc on ne peut avoir qu'un seul de ces constructeurs.

Lien vers la classe externe

Jusqu'à présent, les classes internes apparaissent juste comme un mécanisme de camouflage de nom et d'organisation du code, ce qui est intéressant mais pas vraiment indispensable. Cependant, elles proposent un autre intérêt. Quand on crée une classe interne, un objet de cette classe interne possède un lien vers l'objet extérieur qui l'a créé, il peut donc accéder aux membres de cet objet externe - *sans* aucune qualification spéciale. De plus, les classes internes ont accès à tous les éléments de la classe externe [\[40\]](#). L'exemple suivant le démontre :

```
///  
// Contient une séquence d'Objects.  
  
interface Selector {  
    boolean end();  
    Object current();  
    void next();  
}  
  
public class Sequence {  
    private Object[] obs;  
    private int next = 0;  
    public Sequence(int size) {  
        obs = new Object[size];  
    }  
    public void add(Object x) {  
        if(next < obs.length) {  
            obs[next] = x;  
            next++;  
        }  
    }  
    private class SSelector implements Selector {  
        int i = 0;  
        public boolean end() {  
            return i == obs.length;  
        }  
    }  
}
```

```

    }

    public Object current() {
        return obs[i];
    }

    public void next() {
        if(i < obs.length) i++;
    }
}

public Selector getSelector() {
    return new SSelector();
}

public static void main(String[] args) {
    Sequence s = new Sequence(10);

    for(int i = 0; i < 10; i++)
        s.add(Integer.toString(i));

    Selector sl = s.getSelector();

    while(!sl.end()) {
        System.out.println(sl.current());
        sl.next();
    }
}

} ///:~

```

La **Sequence** est simplement un tableau d'**Objects** de taille fixe paqueté dans une classe. On peut appeler **add()** pour ajouter un nouvel **Object** à la fin de la séquence (s'il reste de la place). Pour retrouver chacun des objets dans une **Sequence**, il existe une interface appelée **Selector**, qui permet de vérifier si on se trouve à la fin (**end()**), de récupérer l'**Object** courant (**current()**), et de se déplacer vers l'**Object** suivant (**next()**) dans la **Sequence**. Comme **Selector** est une interface, beaucoup d'autres classes peuvent implémenter l'**interface** comme elles le veulent, et de nombreuses méthodes peuvent prendre l'**interface** comme un argument, afin de créer du code générique.

Ici, **SSelector** est une classe **private** qui fournit les fonctionnalités de **Selector**. Dans **main()**, on peut voir la création d'une **Sequence**, suivie par l'addition d'un certain nombre d'objets **String**. Un **Selector** est alors produit grâce à un appel à **getSelector()** et celui-ci est alors utilisé pour se déplacer dans la **Sequence** et sélectionner chaque item.

Au premier abord, **SSelector** ressemble à n'importe quelle autre classe interne. Mais regardez-la attentivement. Notez que chacune des méthodes **end()**, **current()** et **next()** utilisent **obs**, qui est une référence n'appartenant pas à **SSelector**, un champ **private** de la classe externe. Cependant, la classe interne peut accéder aux méthodes et

aux champs de la classe externe comme si elle les possédait. Ceci est très pratique, comme on peut le voir dans cet exemple.

Une classe interne a donc automatiquement accès aux membres de la classe externe. Comment cela est-il possible ? La classe interne doit garder une référence de l'objet de la classe externe responsable de sa création. Et quand on accède à un membre de la classe externe, cette référence (cachée) est utilisée pour sélectionner ce membre. Heureusement, le compilateur gère tous ces détails pour nous, mais vous pouvez maintenant comprendre qu'un objet d'une classe interne ne peut être créé qu'en association avec un objet de la classe externe. La construction d'un objet d'une classe interne requiert une référence sur l'objet de la classe externe, et le compilateur se plaindra s'il ne peut accéder à cette référence. La plupart du temps cela se fait sans aucune intervention de la part du programmeur.

Classes internes static

Si on n'a pas besoin du lien entre l'objet de la classe interne et l'objet de la classe externe, on peut rendre la classe interne **static**. Pour comprendre le sens de **static** quand il est appliqué aux classes internes, il faut se rappeler que l'objet d'une classe interne ordinaire garde implicitement une référence sur l'objet externe qui l'a créé. Ceci n'est pas vrai cependant lorsque la classe interne est **static**. Une classe interne **static** implique que :

1. On n'a pas besoin d'un objet de la classe externe afin de créer un objet de la classe interne **static**.
2. On ne peut accéder à un objet de la classe externe depuis un objet de la classe interne **static**.

Les classes internes **static** diffèrent aussi des classes internes non **static** d'une autre manière. Les champs et les méthodes des classes internes non **static** ne peuvent être qu'au niveau externe de la classe, les classes internes non **static** ne peuvent donc avoir de données **static**, de champs **static** ou de classes internes **static**. Par contre, les classes internes **static** peuvent avoir tout cela :

```
//: c08:Parcel10.java
// Classes internes static.

public class Parcel10 {
    private static class PContents
    implements Contents {
        private int i = 11;
        public int value() { return i; }
    }

    protected static class PDestination
        implements Destination {
        private String label;
        private PDestination(String whereTo) {
            label = whereTo;
        }
        public String readLabel() { return label; }
    }
}
```

```

// Les classes internes static peuvent
// contenir d'autres éléments static :
public static void f() {}

static int x = 10;

static class AnotherLevel {
    public static void f() {}
    static int x = 10;
}
}

public static Destination dest(String s) {
    return new PDestination(s);
}

public static Contents cont() {
    return new PContents();
}

public static void main(String[] args) {
    Contents c = cont();
    Destination d = dest("Tanzania");
}
} ///:~

```

Dans **main()**, aucun objet **Parcel10** n'est nécessaire ; on utilise à la place la syntaxe habituelle pour sélectionner un membre **static** pour appeler les méthodes qui renvoient des références sur **Contents** et **Destination**.

Comme on va le voir bientôt, dans une classe interne ordinaire (non **static**), le lien avec la classe externe est réalisé avec une référence spéciale **this**. Une classe interne **static** ne dispose pas de cette référence spéciale **this**, ce qui la rend analogue à une méthode **static**.

Normalement, on ne peut placer du code à l'intérieur d'une **interface**, mais une classe interne **static** peut faire partie d'une **interface**. Comme la classe est **static**, cela ne viole pas les règles des interfaces - la classe interne **static** est simplement placée dans l'espace de noms de l'interface :

```

//: c08:IInterface.java

// Classes internes static à l'intérieur d'interfaces.

interface IInterface {
    static class Inner {

```

```

    int i, j, k;

    public Inner() {}

    void f() {}
}
} ///:~

```

Plus tôt dans ce livre je suggérerais de placer un **main()** dans chaque classe se comportant comme un environnement de tests pour cette classe. Un inconvénient de cette approche est le volume supplémentaire de code compilé qu'on doit supporter. Si cela constitue un problème, on peut utiliser une classe interne **static** destinée à contenir le code de test :

```

///: c08:TestBed.java
// Code de test placé dans une classe interne static.

class TestBed {
    TestBed() {}

    void f() { System.out.println("f()"); }

    public static class Tester {
        public static void main(String[] args) {
            TestBed t = new TestBed();
            t.f();
        }
    }
} ///:~

```

Ceci génère une classe séparée appelée **TestBed\$Tester** (pour lancer le programme, il faut utiliser la commande **java TestBed\$Tester**). On peut utiliser cette classe lors des tests, mais on n'a pas besoin de l'inclure dans le produit final.

Se référer à l'objet de la classe externe

Si on a besoin de produire la référence à l'objet de la classe externe, il faut utiliser le nom de la classe externe suivi par un point et **this**. Par exemple, dans la classe **Sequence.SSelector**, chacune des méthodes peut accéder à la référence à la classe externe **Sequence** stockée en utilisant **Sequence.this**. Le type de la référence obtenue est automatiquement correct (il est connu et vérifié lors de la compilation, il n'y a donc aucune pénalité sur les performances lors de l'exécution).

On peut demander à un autre objet de créer un objet de l'une de ses classes internes. Pour cela il faut fournir une référence à l'autre objet de la classe externe dans l'expression **new**, comme ceci :

```

///: c08:Parcel11.java

```

```

// Création d'instances de classes internes.

public class Parcel11 {

    class Contents {

        private int i = 11;

        public int value() { return i; }

    }

    class Destination {

        private String label;

        Destination(String whereTo) {

            label = whereTo;

        }

        String readLabel() { return label; }

    }

    public static void main(String[] args) {

        Parcel11 p = new Parcel11();

        // On doit utiliser une instance de la classe externe

        // pour créer une instance de la classe interne :

        Parcel11.Contents c = p.new Contents();

        Parcel11.Destination d =

            p.new Destination("Tanzania");

    }

} ///:~

```

Pour créer un objet de la classe interne directement, il ne faut pas utiliser la même syntaxe et se référer au nom de la classe externe **Parcel11** comme on pourrait s'y attendre ; à la place il faut utiliser un *objet* de la classe externe pour créer un objet de la classe interne :

```
Parcel11.Contents c = p.new Contents();
```

Il n'est donc pas possible de créer un objet de la classe interne sans disposer déjà d'un objet de la classe externe, parce qu'un objet de la classe interne est toujours connecté avec l'objet de la classe externe qui l'a créé. Cependant, si la classe interne est **static**, elle n'a pas besoin d'une référence sur un objet de la classe externe.

Classe interne à plusieurs niveaux d'imbrication

[\[41\]](#) Une classe interne peut se situer à n'importe quel niveau d'imbrication - elle pourra toujours accéder de manière transparente à tous les membres de toutes les classes l'entourant, comme on peut le voir :


```

//: c08:MultiNestingAccess.java

// Les classes imbriquées peuvent accéder à tous les membres de tous
// les niveaux des classes dans lesquelles elles sont imbriquées.

class MNA {
    private void f() {}

    class A {
        private void g() {}

        public class B {
            void h() {
                g();
                f();
            }
        }
    }
}

public class MultiNestingAccess {
    public static void main(String[] args) {
        MNA mna = new MNA();
        MNA.A mnaa = mna.new A();
        MNA.A.B mnaab = mnaa.new B();
        mnaab.h();
    }
} ///:~

```

On peut voir que dans **MNA.A.B**, les méthodes **g()** et **f()** sont appelées sans qualification (malgré le fait qu'elles soient **private**). Cet exemple présente aussi la syntaxe utilisée pour créer des objets de classes internes imbriquées quand on crée ces objets depuis une autre classe. La syntaxe «**.new**» fournit la portée correcte et on n'a donc pas besoin de qualifier le nom de la classe dans l'appel du constructeur.

Dériver une classe interne

Comme le constructeur d'une classe interne doit stocker une référence à l'objet de la classe externe, les choses sont un peu plus compliquées lorsqu'on dérive une classe interne. Le problème est que la référence « secrète » sur l'objet de la classe externe *doit* être initialisée, et dans la classe dérivée il n'y a plus d'objet sur lequel se rattacher par défaut. Il faut donc utiliser une syntaxe qui rende cette association explicite :

```
//: c08:InheritInner.java
// Inheriting an inner class.

class WithInner {
    class Inner {}
}

public class InheritInner
    extends WithInner.Inner {

    //! InheritInner() {} // Ne compilera pas.
    InheritInner(WithInner wi) {
        wi.super();
    }

    public static void main(String[] args) {
        WithInner wi = new WithInner();
        InheritInner ii = new InheritInner(wi);
    }
} ///:~
```

On peut voir que **InheritInner** étend juste la classe interne, et non la classe externe. Mais lorsqu'on en arrive au constructeur, celui fourni par défaut n'est pas suffisant et on ne peut se contenter de passer une référence à un objet externe. De plus, on doit utiliser la syntaxe :

```
enclosingClassReference.super();
```

à l'intérieur du constructeur. Ceci fournit la référence nécessaire et le programme pourra alors être compilé.

Les classes internes peuvent-elles redéfinies ?

Que se passe-t-il quand on crée une classe interne, qu'on dérive la classe externe et qu'on redéfinit la classe interne ? Autrement dit, est-il possible de redéfinir une classe interne ? Ce concept semble particulièrement puissant, mais « redéfinir » une classe interne comme si c'était une méthode de la classe externe ne fait rien de spécial :

```
//: c08:BigEgg.java
// Une classe interne ne peut être
// redéfinie comme une méthode.
```

```

class Egg {
    protected class Yolk {
        public Yolk() {
            System.out.println("Egg.Yolk()");
        }
    }
    private Yolk y;
    public Egg() {
        System.out.println("New Egg()");
        y = new Yolk();
    }
}

public class BigEgg extends Egg {
    public class Yolk {
        public Yolk() {
            System.out.println("BigEgg.Yolk()");
        }
    }
    public static void main(String[] args) {
        new BigEgg();
    }
} ///:~

```

Le constructeur par défaut est généré automatiquement par le compilateur, et il appelle le constructeur par défaut de la classe de base. On pourrait penser que puisqu'on crée un **BigEgg**, la version « redéfinie » de **Yolk** sera utilisée, mais ce n'est pas le cas. La sortie produite est :

```

New Egg()
Egg.Yolk()

```

Cet exemple montre simplement qu'il n'y a aucune magie spéciale associée aux classes internes quand on hérite d'une classe externe. Les deux classes internes sont des entités complètement séparées, chacune dans leur propre espace de noms. Cependant, il est toujours possible de dériver explicitement la classe interne :

```

///: c08:BigEgg2.java

```

```
// Dérivation d'une classe interne.

class Egg2 {
    protected class Yolk {
        public Yolk() {
            System.out.println("Egg2.Yolk()");
        }
        public void f() {
            System.out.println("Egg2.Yolk.f()");
        }
    }
    private Yolk y = new Yolk();
    public Egg2() {
        System.out.println("New Egg2()");
    }
    public void insertYolk(Yolk yy) { y = yy; }
    public void g() { y.f(); }
}

public class BigEgg2 extends Egg2 {
    public class Yolk extends Egg2.Yolk {
        public Yolk() {
            System.out.println("BigEgg2.Yolk()");
        }
        public void f() {
            System.out.println("BigEgg2.Yolk.f()");
        }
    }
    public BigEgg2() { insertYolk(new Yolk()); }
    public static void main(String[] args) {
        Egg2 e2 = new BigEgg2();
        e2.g();
    }
} ///:~
```

Maintenant **BigEgg2.Yolk** étend explicitement **Egg2.Yolk** et redéfinit ses méthodes. La méthode **insertYolk()** permet à **BigEgg2** de transtyper un de ses propres objets **Yolk** dans la référence **y** de **Egg2**, donc quand **g()** appelle **y.f()**, la version redéfinie de **f()** est utilisée. La sortie du programme est :

```
Egg2.Yolk()
New Egg2()
Egg2.Yolk()
BigEgg2.Yolk()
BigEgg2.Yolk.f()
```

Le second appel à **Egg2.Yolk()** est l'appel du constructeur de la classe de base depuis le constructeur de **BigEgg2.Yolk**. On peut voir que la version redéfinie de **f()** est utilisée lorsque **g()** est appelée.

Identifiants des classes internes

Puisque chaque classe produit un fichier **.class** qui contient toutes les informations concernant la création d'objets de ce type (ces informations produisent une « méta-classe » dans un objet **Class**), il est aisé de deviner que les classes internes produisent aussi des fichiers **.class** qui contiennent des informations pour *leurs* objets **Class**. La nomenclature de ces fichiers / classes est stricte : le nom de la classe externe suivie par un \$, suivi du nom de la classe interne. Par exemple, les fichiers **.class** créés par **InheritInner.java** incluent :

```
InheritInner.class
WithInner$Inner.class
WithInner.class
```

Si les classes internes sont anonymes, le compilateur génère simplement des nombres comme identifiants de classe interne. Si des classes internes sont imbriquées dans d'autres classes internes, leur nom est simplement ajouté après un \$ et le nom des identifiants des classes externes.

Bien que cette gestion interne des noms soit simple et directe, elle est robuste et gère la plupart des situations [\[42\]](#). Et comme cette notation est la notation standard pour Java, les fichiers générés sont automatiquement indépendants de la plateforme (Notez que le compilateur Java modifie les classes internes d'un tas d'autres manières afin de les faire fonctionner).

Raison d'être des classes internes

Jusqu'à présent, on a vu la syntaxe et la sémantique décrivant la façon dont les classes internes fonctionnent, mais cela ne répond pas à la question du pourquoi de leur existence. Pourquoi Sun s'est-il donné tant de mal pour ajouter au langage cette fonctionnalité fondamentale ?

Typiquement, la classe interne hérite d'une classe ou implémente une **interface**, et le code de la classe interne manipule l'objet de la classe externe l'ayant créé. On peut donc dire qu'une classe interne est une sorte de fenêtre

dans la classe externe.

Mais si on a juste besoin d'une référence sur une **interface**, pourquoi ne pas implémenter cette **interface** directement dans la classe externe ? La réponse à cette question allant au coeur des classes internes est simple : « Si c'est tout ce dont on a besoin, alors c'est ainsi qu'il faut procéder ». Alors qu'est-ce qui distingue une classe interne implémentant une **interface** d'une classe externe implémentant cette même **interface** ? C'est tout simplement qu'on ne dispose pas toujours des facilités fournies par les **interfaces** - quelquefois on est obligé de travailler avec des implémentations. Voici donc la raison principale d'utiliser des classes internes :

Chaque classe interne peut hériter indépendamment d'une implémentation. La classe interne n'est pas limitée par le fait que la classe externe hérite déjà d'une implémentation.

Sans cette capacité que fournissent les classes internes d'hériter - dans la pratique - de plus d'une classe concrète ou **abstract**, certaines conceptions ou problèmes seraient impossibles à résoudre. Les classes internes peuvent donc être considérées comme la suite de la solution au problème de l'héritage multiple. Les interfaces résolvent une partie du problème, mais les classes internes permettent réellement « l'héritage multiple d'implémentations ». Les classes internes permettent effectivement de dériver plusieurs non**interfaces**.

Pour voir ceci plus en détails, imaginons une situation dans laquelle une classe doit implémenter deux interfaces. Du fait de la flexibilité des interfaces, on a le choix entre avoir une classe unique ou s'aider d'une classe interne :

```
//: c08:MultiInterfaces.java

// Deux façons pour une classe
// d'implémenter des interfaces multiples.

interface A {}

interface B {}

class X implements A, B {}

class Y implements A {
    B makeB() {
        // Classe interne anonyme :
        return new B() {};
    }
}

public class MultiInterfaces {
    static void takesA(A a) {}
    static void takesB(B b) {}
}
```

```

public static void main(String[] args) {

    X x = new X();

    Y y = new Y();

    takesA(x);

    takesA(y);

    takesB(x);

    takesB(y.makeB());

}

} ///:~

```

Bien sûr, la structure du code peut impliquer une logique pouvant imposer l'une ou l'autre des solutions. La nature du problème fournit généralement aussi des indices pour choisir entre une classe unique ou une classe interne. Mais en l'absence d'aucune autre contrainte, l'approche choisie dans l'exemple précédent ne fait aucune différence du point de vue implémentation. Les deux fonctionnent.

Cependant, si on a des classes **abstract** ou concrètes à la place des **interfaces**, on est obligé de recourir aux classes internes si la classe doit implémenter les deux :

```

///: c08:MultiImplementation.java

// Avec des classes concrètes ou abstract, les classes

// internes constituent le seul moyen de mettre en oeuvre

// « l'héritage multiple d'implémentations ».

```

```

class C {}

abstract class D {}

class Z extends C {

    D makeD() { return new D() {}; }

}

```

```

public class MultiImplementation {

    static void takesC(C c) {}

    static void takesD(D d) {}

    public static void main(String[] args) {

        Z z = new Z();

        takesC(z);

        takesD(z.makeD());

}

```

```

    }

    } ///:~

```

Si le problème de « l'héritage multiple d'implémentations » ne se pose pas, on peut tout à fait se passer des classes internes. Mais les classes internes fournissent toutefois des fonctionnalités intéressantes :

1. Les classes internes peuvent avoir plusieurs instances, chacune avec ses propres informations indépendantes des informations de l'objet de la classe externe.
2. Dans une classe externe on peut avoir plusieurs classes internes, chacune implémentant la même **interface** ou dérivant la même classe d'une façon différente. Nous allons en voir un exemple bientôt.
3. Le point de création d'un objet de la classe interne n'est pas lié à la création de l'objet de la classe externe.
4. Aucune confusion concernant la relation « est-un » n'est possible avec la classe interne ; c'est une entité séparée.

Par exemple, si **Sequence.java** n'utilisait pas de classes internes, il aurait fallu dire « une **Sequence** est un **Selector** », et on n'aurait pu avoir qu'un seul **Selector** pour une **Sequence** particulière. De plus, on peut avoir une autre méthode, **getRSelector()**, qui produise un **Selector** parcourant la **Sequence** dans l'ordre inverse. Cette flexibilité n'est possible qu'avec les classes internes.

Fermetures & callbacks

Une *fermeture* est un objet qui retient des informations de la portée dans laquelle il a été créé. A partir de cette définition, il est clair qu'une classe interne est une fermeture orientée objet, parce qu'elle ne contient pas seulement chaque élément d'information de l'objet de la classe externe (« la portée dans laquelle il a été créé »), mais elle contient aussi automatiquement une référence sur l'objet de la classe externe, avec la permission d'en manipuler tous les membres, y compris les **private**.

L'un des arguments les plus percutants mis en avant pour inclure certains mécanismes de pointeur dans Java était de permettre les *callbacks*. Avec un callback, on donne des informations à un objet lui permettant de revenir plus tard dans l'objet originel. Ceci est un concept particulièrement puissant, comme nous le verrons dans les chapitres 13 et 16. Cependant, si les callbacks étaient implémentés avec des pointeurs, le programmeur serait responsable de la gestion de ce pointeur et devrait faire attention afin de ne pas l'utiliser de manière incontrôlée. Mais comme on l'a déjà vu, Java n'aime pas ce genre de solutions reposant sur le programmeur, et les pointeurs ne furent pas inclus dans le langage.

Les classes internes fournissent une solution parfaite pour les fermetures, bien plus flexible et de loin plus sûre qu'un pointeur. Voici un exemple simple :

```

///: c08:Callbacks.java

// Utilisation des classes internes pour les callbacks

interface Incrementable {

    void increment();

}

// Il est très facile d'implémenter juste l'interface :

```



```
class Callee1 implements Incrementable {

    private int i = 0;

    public void increment() {

        i++;

        System.out.println(i);

    }

}

class MyIncrement {

    public void increment() {

        System.out.println("Other operation");

    }

    public static void f(MyIncrement mi) {

        mi.increment();

    }

}

// Si la classe doit aussi implémenter increment() d'une
// autre façon, il faut utiliser une classe interne :

class Callee2 extends MyIncrement {

    private int i = 0;

    private void incr() {

        i++;

        System.out.println(i);

    }

    private class Closure implements Incrementable {

        public void increment() { incr(); }

    }

    Incrementable getCallbackReference() {

        return new Closure();

    }

}
```

```

class Caller {
    private Incrementable callbackReference;

    Caller(Incrementable cbh) {
        callbackReference = cbh;
    }

    void go() {
        callbackReference.increment();
    }
}

public class Callbacks {
    public static void main(String[] args) {
        Callee1 c1 = new Callee1();
        Callee2 c2 = new Callee2();
        MyIncrement.f(c2);

        Caller caller1 = new Caller(c1);
        Caller caller2 =
            new Caller(c2.getCallbackReference());

        caller1.go();
        caller1.go();
        caller2.go();
        caller2.go();
    }
} ///:~

```

Cet exemple est un exemple supplémentaire montrant les différences entre l'implémentation d'une interface dans une classe externe ou une classe interne. **Callee1** est sans conteste la solution la plus simple en terme de code. **Callee2** hérite de **MyIncrement** qui dispose déjà d'une méthode **increment()** faisant quelque chose de complètement différent que ce qui est attendu par l'interface **Incrementable**. Quand **MyIncrement** est dérivée dans **Callee2**, **increment()** ne peut être redéfinie pour être utilisée par **Incrementable**, on est donc forcé d'utiliser une implémentation séparée avec une classe interne. Notez également que lorsqu'on crée une classe interne, on n'étend pas ni ne modifie l'interface de la classe externe.

Remarquez bien que tout dans **Callee2** à l'exception de **getCallbackReference()** est **private**. L'interface **Incrementable** est essentielle pour permettre *toute* interaction avec le monde extérieur. Les **interfaces** permettent donc une séparation complète entre l'interface et l'implémentation.

La classe interne **Closure** implémente **Incrementable** uniquement pour fournir un point de retour dans **Callee2**

- mais un point de retour sûr. Quiconque récupère la référence sur **Incrementable** ne peut appeler qu'**increment()** (contrairement à un pointeur, qui aurait permis de faire tout ce qu'on veut).

Caller prend une référence **Incrementable** dans son constructeur (bien qu'on puisse fournir cette référence - ce callback - n'importe quand), et s'en sert par la suite, parfois bien plus tard, pour « revenir » dans la classe **Callee**.

La valeur des callbacks réside dans leur flexibilité - on peut décider dynamiquement quelles fonctions vont être appelées lors de l'exécution. Les avantages des callbacks apparaîtront dans le chapitre 13, où ils sont utilisés immodérément pour implémenter les interfaces graphiques utilisateurs (GUI).

Classes internes & structures de contrôle

Un exemple plus concret d'utilisation des classes internes est ce que j'appelle les *structures de contrôle*.

Une *structure d'application* est une classe ou un ensemble de classes conçues pour résoudre un type particulier de problème. Pour utiliser une structure d'application, il suffit de dériver une ou plusieurs de ces classes et de redéfinir certaines des méthodes. Le code écrit dans les méthodes redéfinies particularise la solution générale fournie par la structure d'application, afin de résoudre le problème considéré. Les structures de contrôle sont un type particulier des structures d'application dominées par la nécessité de répondre à des événements ; un système qui répond à des événements est appelé un *système à programmation événementielle*. L'un des problèmes les plus ardues en programmation est l'interface graphique utilisateur (GUI), qui est quasiment entièrement événementielle. Comme nous le verrons dans le Chapitre 13, la bibliothèque Java Swing est une structure de contrôle qui résout élégamment le problème des interfaces utilisateurs en utilisant extensivement les classes internes.

Pour voir comment les classes internes permettent une mise en oeuvre aisée des structures de contrôle, considérons le cas d'une structure de contrôle dont le rôle consiste à exécuter des événements dès lors que ces événements sont « prêts ». Bien que « prêt » puisse vouloir dire n'importe quoi, dans notre cas nous allons nous baser sur un temps d'horloge. Ce qui suit est une structure de contrôle qui ne contient aucune information spécifique sur ce qu'elle contrôle. Voici tout d'abord l'interface qui décrit tout événement. C'est une classe **abstract** plutôt qu'une **interface** parce que le comportement par défaut est de réaliser le contrôle sur le temps, donc une partie de l'implémentation peut y être incluse :

```
///  
// Les méthodes communes pour n'importe quel événement.  
  
package c08.controller;  
  
abstract public class Event {  
    private long evtTime;  
  
    public Event(long eventTime) {  
        evtTime = eventTime;  
    }  
  
    public boolean ready() {  
        return System.currentTimeMillis() >= evtTime;  
    }  
}
```

```

    abstract public void action();

    abstract public String description();
} ///:~

```

Le constructeur stocke simplement l'heure à laquelle on veut que l'**Event** soit exécuté, tandis que **ready()** indique si c'est le moment de le lancer. Bien sûr, **ready()** peut être redéfini dans une classe dérivée pour baser les **Event** sur autre chose que le temps.

action() est la méthode appelée lorsque l'**Event** est **ready()**, et **description()** donne des informations (du texte) à propos de l'**Event**.

Le fichier suivant contient la structure de contrôle proprement dite qui gère et déclenche les événements. La première classe est simplement une classe « d'aide » dont le rôle consiste à stocker des objets **Event**. On peut la remplacer avec n'importe quel conteneur plus approprié, et dans le Chapitre 9 nous verrons d'autres conteneurs qui ne requerront pas ce code supplémentaire :

```

///: c08:controller:Controller.java
// Avec Event, la structure générique
// pour tous les systèmes de contrôle :

package c08.controller;

// Ceci est jsute une manière de stocker les objets Event.
class EventSet {
    private Event[] events = new Event[100];
    private int index = 0;
    private int next = 0;
    public void add(Event e) {
        if(index >= events.length)
            return; // (Normalement, générer une exception)
        events[index++] = e;
    }
    public Event getNext() {
        boolean looped = false;
        int start = next;
        do {
            next = (next + 1) % events.length;
            // Vérifie si on a fait le tour :
            if(start == next) looped = true;

```

```

        // Si on a fait le tour, c'est que la
        // liste est vide :
        if((next == (start + 1) % events.length)
            && looped)
            return null;
    } while(events[next] == null);
    return events[next];
}

public void removeCurrent() {
    events[next] = null;
}
}

public class Controller {
    private EventSet es = new EventSet();

    public void addEvent(Event c) { es.add(c); }

    public void run() {
        Event e;
        while((e = es.getNext()) != null) {
            if(e.ready()) {
                e.action();

                System.out.println(e.description());

                es.removeCurrent();
            }
        }
    }
}
} ///:~

```

EventSet stocke arbitrairement 100 **Events** (si un « vrai » conteneur du Chapitre 9 était utilisé ici, on n'aurait pas à se soucier à propos de sa taille maximum, puisqu'il se redimensionnerait de lui-même). L'**index** est utilisé lorsqu'on veut récupérer le prochain **Event** de la liste, pour voir si on a fait le tour. Ceci est important pendant un appel à **getNext()**, parce que les objets **Event** sont enlevés de la liste (avec **removeCurrent()**) une fois exécutés, donc **getNext()** rencontrera des trous dans la liste lorsqu'il la parcourra.

Notez que **removeCurrent()** ne positionne pas simplement un flag indiquant que l'objet n'est plus utilisé. A la place, il positionne la référence à **null**. C'est important car si le ramasse-miettes rencontre une référence qui est encore utilisée il ne pourra pas nettoyer l'objet correspondant. Si l'objet n'a plus de raison d'être (comme c'est le

cas ici), il faut alors mettre leur référence à **null** afin d'autoriser le ramasse-miettes à les nettoyer.

C'est dans **Controller** que tout le travail est effectué. Il utilise un **EventSet** pour stocker ses objets **Event**, et **addEvent()** permet d'ajouter de nouveaux éléments à la liste. Mais la méthode principale est **run()**. Cette méthode parcourt l'**EventSet**, recherchant un objet **Event** qui soit **ready()**. Il appelle alors la méthode **action()** pour cet objet, affiche sa **description()** et supprime l'**Event** de la liste.

Notez que jusqu'à présent dans la conception on ne sait rien sur *ce que fait* exactement un **Event**. Et c'est le point fondamental de la conception : comment elle « sépare les choses qui changent des choses qui ne bougent pas ». Ou, comme je l'appelle, le « vecteur de changement » est constitué des différentes actions des différents types d'objets **Event**, actions différentes réalisées en créant différentes sous-classes d'**Event**.

C'est là que les classes internes interviennent. Elles permettent deux choses :

1. Réaliser l'implémentation complète d'une application de structure de contrôle dans une seule classe, encapsulant du même coup tout ce qui est unique dans cette implémentation. Les classes internes sont utilisées pour décrire les différents types d'**action()** nécessaires pour résoudre le problème. De plus, l'exemple suivant utilise des classes internes **private** afin que l'implémentation soit complètement cachée et puisse être changée en toute impunité.
2. Empêcher que l'implémentation ne devienne trop lourde, puisqu'on est capable d'accéder facilement à chacun des membres de la classe externe. Sans cette facilité, le code deviendrait rapidement tellement confus qu'il faudrait chercher une autre solution.

Considérons une implémentation particulière de la structure de contrôle conçue pour contrôler les fonctions d'une serre [43]. Chaque action est complètement différente : contrôler les lumières, l'arrosage et la température, faire retentir des sonneries et relancer le système. Mais la structure de contrôle est conçue pour isoler facilement ce code différent. Les classes internes permettent d'avoir de multiples versions dérivées de la même classe de base (ici, **Event**) à l'intérieur d'une seule et même classe. Pour chaque type d'action on crée une nouvelle classe interne dérivée d'**Event**, et on écrit le code de contrôle dans la méthode **action()**.

Typiquement, la classe **GreenhouseControls** hérite de **Controller** :

```

//: c08:GreenhouseControls.java

// Ceci est une application spécifique du
// système de contrôle, le tout dans une seule classe.
// Les classes internes permettent d'encapsuler des
// fonctionnalités différentes pour chaque type d'Event.

import c08.controller.*;

public class GreenhouseControls
    extends Controller {
    private boolean light = false;
    private boolean water = false;
    private String thermostat = "Day";
    private class LightOn extends Event {

```

```
public LightOn(long eventTime) {
    super(eventTime);
}

public void action() {
    // Placer ici du code de contrôle hardware pour
    // réellement allumer la lumière.

    light = true;
}

public String description() {
    return "Light is on";
}
}

private class LightOff extends Event {
    public LightOff(long eventTime) {
        super(eventTime);
    }

    public void action() {
        // Put hardware control code here to
        // physically turn off the light.

        light = false;
    }

    public String description() {
        return "Light is off";
    }
}

private class WaterOn extends Event {
    public WaterOn(long eventTime) {
        super(eventTime);
    }

    public void action() {
        // Placer ici du code de contrôle hardware.

        water = true;
    }

    public String description() {
```

```
        return "Greenhouse water is on";
    }
}

private class WaterOff extends Event {
    public WaterOff(long eventTime) {
        super(eventTime);
    }

    public void action() {
        // Placer ici du code de contrôle hardware.

        water = false;
    }

    public String description() {
        return "Greenhouse water is off";
    }
}

private class ThermostatNight extends Event {
    public ThermostatNight(long eventTime) {
        super(eventTime);
    }

    public void action() {
        // Placer ici du code de contrôle hardware.

        thermostat = "Night";
    }

    public String description() {
        return "Thermostat on night setting";
    }
}

private class ThermostatDay extends Event {
    public ThermostatDay(long eventTime) {
        super(eventTime);
    }

    public void action() {
        // Placer ici du code de contrôle hardware.

        thermostat = "Day";
    }
}
```



```
    }

    public String description() {
        return "Thermostat on day setting";
    }
}

// Un exemple d'une action() qui insère une nouvelle
// instance de son type dans la liste d'Event :
private int rings;

private class Bell extends Event {
    public Bell(long eventTime) {
        super(eventTime);
    }

    public void action() {
        // Sonne toutes les 2 secondes, 'rings' fois :
        System.out.println("Bing!");

        if(--rings > 0)
            addEvent(new Bell(
                System.currentTimeMillis() + 2000));
    }

    public String description() {
        return "Ring bell";
    }
}

private class Restart extends Event {
    public Restart(long eventTime) {
        super(eventTime);
    }

    public void action() {
        long tm = System.currentTimeMillis();

        // Au lieu d'un codage en dur, on pourrait
        // récupérer les informations en parsant un
        // fichier de configuration :

        rings = 5;

        addEvent(new ThermostatNight(tm));
    }
}
```

```

        addEvent(new LightOn(tm + 1000));
        addEvent(new LightOff(tm + 2000));
        addEvent(new WaterOn(tm + 3000));
        addEvent(new WaterOff(tm + 8000));
        addEvent(new Bell(tm + 9000));
        addEvent(new ThermostatDay(tm + 10000));

        // On peut même ajouter un objet Restart !
        addEvent(new Restart(tm + 20000));
    }

    public String description() {
        return "Restarting system";
    }
}

public static void main(String[] args) {
    GreenhouseControls gc =
        new GreenhouseControls();

    long tm = System.currentTimeMillis();

    gc.addEvent(gc.new Restart(tm));

    gc.run();
}
} ///:~

```

Notez que **light**, **water**, **thermostat** et **rings** appartiennent tous à la classe externe **GreenhouseControls**, et donc les classes internes peuvent accéder à ces champs sans qualification ou permission particulière. De plus, la plupart des méthodes **action()** effectuent un contrôle hardware, qui implique certainement des appels à du code non-Java.

La plupart des classes **Event** sont similaires, mais **Bell** et **Restart** sont spéciales. **Bell** sonne, et si elle n'a pas sonné un nombre suffisant de fois, elle ajoute un nouvel objet **Bell** à la liste des événements afin de sonner à nouveau plus tard. Notez comme les classes internes *semblent* bénéficier de l'héritage multiple : **Bell** possède toutes les méthodes d'**Event** mais elle semble disposer également de toutes les méthodes de la classe externe **GreenhouseControls**.

Restart est responsable de l'initialisation du système, il ajoute donc tous les événements appropriés. Bien sûr, une manière plus flexible de réaliser ceci serait d'éviter le codage en dur des événements et de les extraire d'un fichier à la place (c'est précisément ce qu'un exercice du Chapitre 11 demande de faire). Puisque **Restart()** n'est qu'un objet **Event** comme un autre, on peut aussi ajouter un objet **Restart** depuis **Restart.action()** afin que le système se relance de lui-même régulièrement. Et tout ce qu'on a besoin de faire dans **main()** est de créer un objet **GreenhouseControls** et ajouter un objet **Restart** pour lancer le processus.

Cet exemple devrait vous avoir convaincu de l'intérêt des classes internes, spécialement dans le cas des structures de contrôle. Si ce n'est pas le cas, dans le Chapitre 13, vous verrez comment les classes internes sont utilisées pour décrire élégamment les actions d'une interface graphique utilisateur. A la fin de ce chapitre vous devriez être complètement convaincu.

Résumé

Les interfaces et les classes internes sont des concepts plus sophistiqués que ce que vous pourrez trouver dans beaucoup de langages de programmation orientés objets. Par exemple, rien de comparable n'existe en C++. Ensemble, elles résolvent le même problème que celui que le C++ tente de résoudre avec les fonctionnalités de l'héritage multiple. Cependant, l'héritage multiple en C++ se révèle relativement ardu à mettre en oeuvre, tandis que les interfaces et les classes internes en Java sont, en comparaison, d'un abord nettement plus facile.

Bien que les fonctionnalités en elles-mêmes soient relativement simples, leur utilisation relève de la conception, de même que le polymorphisme. Avec le temps, vous reconnaîtrez plus facilement les situations dans lesquelles utiliser une interface, ou une classe interne, ou les deux. Mais à ce point du livre vous devriez à tout le moins vous sentir à l'aise avec leur syntaxe et leur sémantique. Vous intégrerez ces techniques au fur et à mesure que vous les verrez utilisées.

Exercices

Les solutions d'exercices sélectionnés peuvent être trouvées dans le document électronique *The Thinking in Java Annotated Solution Guide*, disponible pour un faible coût sur www.BruceEckel.com.

1. Prouvez que les champs d'une **interface** sont implicitement **static** et **final**.
2. Créez une **interface** contenant trois méthodes, dans son propre **package**. Implémentez cette interface dans un **package** différent.
3. Prouvez que toutes les méthodes d'une **interface** sont automatiquement **public**.
4. Dans **c07:Sandwich.java**, créez une interface appelée **FastFood** (avec les méthodes appropriées) et changez **Sandwich** afin qu'il implémente **FastFood**.
5. Créez trois **interfaces**, chacune avec deux méthodes. Créez une nouvelle **interface** héritant des trois, en ajoutant une nouvelle méthode. Créez une classe implémentant la nouvelle interface et héritant déjà d'une classe concrète. Ecrivez maintenant quatre méthodes, chacune d'entre elles prenant l'une des quatre **interfaces** en argument. Dans **main()**, créez un objet de votre classe et passez-le à chacune des méthodes.
6. Modifiez l'exercice 5 en créant une classe **abstract** et en la dérivant dans la dernière classe.
7. Modifiez **Music5.java** en ajoutant une **interfacePlayable**. Enlevez la déclaration de **play()** d'**Instrument**. Ajoutez **Playable** aux classes dérivées en l'incluant dans la liste **implements**. Changez **tune()** afin qu'il accepte un **Playable** au lieu d'un **Instrument**.
8. Changez l'exercice 6 du Chapitre 7 afin que **Rodent** soit une **interface**.
9. Dans **Adventure.java**, ajoutez une **interface** appelée **CanClimb** respectant la forme des autres interfaces.
10. Ecrivez un programme qui importe et utilise **Month2.java**.
11. En suivant l'exemple donné dans **Month2.java**, créez une énumération des jours de la semaine.
12. Créez une **interface** dans son propre package contenant au moins une méthode. Créez une classe dans un package séparé. Ajoutez une classe interne **protected** qui implémente l'**interface**. Dans un troisième package, dérivez votre classe, et dans une méthode renvoyez un objet de la classe interne **protected**, en le transtypant en **interface** durant le retour.
13. Créez une **interface** contenant au moins une méthode, et implémentez cette **interface** en définissant une classe interne à l'intérieur d'une méthode, qui renvoie une référence sur votre **interface**.
14. Répétez l'exercice 13 mais définissez la classe interne à l'intérieur d'une portée à l'intérieur de la méthode.

15. Répétez l'exercice 13 en utilisant une classe interne anonyme.
16. Créez une classe interne **private** qui implémente une **interface public**. Ecrivez une méthode qui renvoie une référence sur une instance de la classe interne **private**, transtypée en **interface**. Montrez que la classe interne est complètement cachée en essayant de la transtyper à nouveau.
17. Créez une classe avec un constructeur autre que celui par défaut et sans constructeur par défaut. Créez une seconde classe disposant d'une méthode qui renvoie une référence à la première classe. Créez un objet à renvoyer en créant une classe interne anonyme dérivée de la première classe.
18. Créez une classe avec un champ **private** et une méthode **private**. Créez une classe interne avec une méthode qui modifie le champ de la classe externe et appelle la méthode de la classe externe. Dans une seconde méthode de la classe externe, créez un objet de la classe interne et appelez sa méthode ; montrez alors l'effet sur l'objet de la classe externe.
19. Répétez l'exercice 18 en utilisant une classe interne anonyme.
20. Créez une classe contenant une classe interne **static**. Dans **main()**, créez une instance de la classe interne.
21. Créez une **interface** contenant une classe interne **static**. Implémentez cette **interface** et créez une instance de la classe interne.
22. Créez une classe contenant une classe interne contenant elle-même une classe interne. Répétez ce schéma en utilisant des classes internes **static**. Notez les noms des fichiers **.class** produits par le compilateur.
23. Créez une classe avec une classe interne. Dans une classe séparée, créez une instance de la classe interne.
24. Créez une classe avec une classe interne disposant d'un constructeur autre que celui par défaut. Créez une seconde classe avec une classe interne héritant de la première classe interne.
25. Corrigez le problème dans **WindError.java**.
26. Modifiez **Sequence.java** en ajoutant une méthode **getRSelector()** qui produise une implémentation différente de l'**interface Selector** afin de parcourir la séquence en ordre inverse, de la fin vers le début.
27. Créez une **interface U** contenant trois méthodes. Créez une classe **A** avec une méthode qui produise une référence sur un **U** en construisant une classe interne anonyme. Créez une seconde classe **B** qui contienne un tableau de **U**. **B** doit avoir une méthode qui accepte et stocke une référence sur un **U** dans le tableau, une deuxième méthode qui positionne une référence (spécifiée par un argument de la méthode) dans le tableau à **null**, et une troisième méthode qui se déplace dans le tableau et appelle les méthodes de l'objet **U**. Dans **main()**, créez un groupe d'objets **A** et un objet **B**. Remplissez l'objet **B** avec les références **U** produites par les objets **A**. Utilisez **B** pour revenir dans les objets **A**. Enlevez certaines des références **U** de **B**.
28. Dans **GreenhouseControls.java**, ajoutez des classes internes **Event** qui contrôlent des ventilateurs.
29. Montrez qu'une classe interne peut accéder aux éléments **private** de sa classe externe. Déterminez si l'inverse est vrai.

[38] Cette approche m'a été inspirée par un e-mail de Rich Hoffarth.

[39] Merci à Martin Danner pour avoir posé cette question lors d'un séminaire.

[40] Ceci est très différent du concept des *classes imbriquées* en C++, qui est simplement un mécanisme de camouflage de noms. Il n'y a aucun lien avec l'objet externe et aucune permission implicite en C++.

[41] Merci encore à Martin Danner.

[42] Attention cependant, '\$' est un méta-caractère pour les shells unix et vous pourrez parfois rencontrer des difficultés en listant les fichiers **.class**. Ceci peut paraître bizarre de la part de Sun, une entreprise résolument tournée vers unix. Je pense qu'ils n'ont pas pris en compte ce problème car ils pensaient que l'attention se porterait surtout sur les fichiers sources.

[43] Pour je ne sais quelle raison, ce problème m'a toujours semblé plaisant ; il vient de mon livre précédent

C++ Inside & Out, mais Java offre une solution bien plus élégante.