Support de cours Java

Structures de données
Notions en Génie Logiciel
et Programmation Orientée Objet

H. Mounier

Université Paris Sud

Table des matières

Table des matières **i**

I Threads **1**
- I.1 Modèle de threads de Java 1
- I.2 Création et utilitaires 3
- I.3 Synchronisation 7
- I.4 Groupes de threads 15
- I.5 États d'une thread et ordonnancement 20

II Génie logiciel appliqué **25**
- II.1 Implantation d'une pile en Java 25
- II.2 Minimisation de couplage entre méthodes 32
- II.3 Maximisation de cohésion des méthodes 39

III Programmation orientée objet en Java **49**
- III.1 Exemple 1 : balles 49
- III.2 Exemple 2 : Flipper 56
- III.3 Formes d'héritage 63
- III.4 Exemple 3 : jeu de solitaire 71

IV Boîtes à outils awt et Swing **85**
- IV.1 Aperçu des classes 85
- IV.2 Gestion des événements 90
- IV.3 Gestionnaire de disposition 93
- IV.4 Composants d'interface utilisateur spécifiquement Awt 95
- IV.5 Un exemple 103
- IV.6 Boîtes de dialogue 106
- IV.7 Menus 108

V Classe `Url` **111**
- V.1 Classe `URL` 111

VI Paquetage java.net : Sockets **115**
VI.1 Sockets : Rappels TCP/IP 115
VI.2 Sockets : Clients 116
VI.3 Sockets serveurs 125
VI.4 Serveurs itératifs 128
VI.5 Serveurs non bloquants 130
VI.6 Serveurs concurrents 132

VII Invotaion de méthodes distantes : RMI **135**
VII.1 Notion d'invocation de méthode distante 135
VII.2 Sécurité – Sérialisation 136
VII.3 Brève description du fonctionnement des RMI 138
VII.4 Implantation 140
VII.5 Paquetage java.rmi 144
VII.6 Paquetage java.rmi.registry 148
VII.7 Paquetage java.rmi.server 149

Bibliographie **151**

I – Threads

Références bibliographiques

- *The Java Language Specification*, J. Gosling, B. Joy et G. Steele [GJS96],
- *Java Threads*, S. Oaks et J. Wong [OW97],
- *Concurrent Programming in Java. Design Principles and Patterns*, D. Lea [Lea97].

I.1 Modèle de threads de Java

1.1 Threads et processus

- Le multi-tâches de threads occasionne bien moins de surcharge que le multi-tâches de processus
- *Processus* :
  - Un *ensemble de ressources* ; par ex. sous UNIX, segment de code, de données utilisateur, de données systèmes (répertoire de travail, descripteurs de fichiers, identificateurs de l'utilisateur ayant lancé le processus, du groupe dont il fait partie, infos. sur l'emplacment des données en mémoire, . . .)
  - Une *unité d'exécution* (avec, par ex. sous UNIX, des informations nécessaires à l'ordonnanceur, la valeur des registres, le compteur d'instructions, la pile d'exécution, des informations relatives aux signaux)
- *Thread* : on ne retient que l'aspect d'**unité d'exécution**.
- Contrairement aux processus, les threads sont légères :
  - elles partagent le même espace d'adressage,
  - elles existent au sein du même processus (lourd),
  - la communication inter-thread occasionne peu de surcharge,
  - le passage contextuel (context switching) d'une thread à une autre est peu coûteux.

- Le multi-tâches de processus n'est pas sous le contrôle de l'environnement d'exécution java. Par contre, il y a un mécanisme interne de gestion multi-threads.
- Le peu de surcharge occasionné par le système de multi-threads de Java est spécialement intéressant pour les applications distribuées. Par ex., un serveur multi-tâches (avec une tâche par client à servir)

1.2 Etats d'une thread

- Dans un environnement mono-thread, lorsqu'une thread se bloque (voit son exécution suspendue) en attente d'une ressource, le programme tout entier s'arrête. Ceci n'est plus le cas avec un environnement multi-threads
- Une thread peut être :
  - en train de s'exécuter (*running*) ;
  - prète à s'exécuter (*ready to run*), dès que la CPU est disponible ;
  - suspendue (*suspended*), ce qui stoppe temporairement son activité ;
  - poursuivre l'exécution (*resumed*), là où elle a été suspendue ;
  - bloquée (*blocked*) en attendant une ressource.

  Une thread peut se terminer à tout moment, ce qui arrête son exécution immédiatement. Une fois terminée, une thread ne peut être remise en route

1.3 Priorités d'une thread

- Une thread peut
  - soit volontairement laisser le contrôle. Ceci est réalisé en passant le contrôle explicitement, dormant ou bloquant en E/S. La thread prète à s'exécuter et de plus haute priorité est alors lancée.
  - soit être supplantée par une autre de plus haute priorité. Si c'est systématiquement le cas, on parle de *multi-tâches préemptif.*
- Dans le cas où 2 threads de même priorité veulent s'exécuter, le résultat est **dépendant du système d'exploitation** (de son algorithme d'ordonnancement).
- Sous Windows 95, le même quantum de temps est tour à tour alloué aux threads de même priorité.
- Sous Solaris 2.x, des threads de même priorité doivent explicitement passer le contrôle à leurs pairs ...

1.4 Synchronisation

- Synchronisation inter-threads *via* des *moniteurs*. Notion définie par C.A.R Hoare.

– Moniteur : mini boîte ne pouvant contenir qu'une thread. Une fois qu'une thread y est entrée, **les autres doivent attendre qu'elle en sorte**.
– Pas de classe " `Monitor`". Chaque objet a son propre moniteur implicite dans lequel on entre automatiquement lorsqu'une méthode synchronisée de (l'instance de) l'objet est appelée.
– Lorsqu'une thread est dans une méthode synchronisée, aucune autre thread ne peut appeler de méthode synchronisée de la même instance de l'objet.
– Permet un code clair et compact, la synchronisation étant construite dans le langage.
– Il est également possible pour 2 threads de communiquer par un mécanisme simple et souple.

I.2 Création et utilitaires

2.1 Classe `Thread` et interface `Runnable`

– Le système de multi-threads de Java est construit autour de la classe `Thread` et de son interface compagnon `Runnable`.
– Liste des méthodes les plus courantes :

`methode()`	But
`getName()`	Obtenir le nom d'une thread
`getPriority()`	Obtenir la priorité d'une thread
`isAlive()`	Déterminer si une thread est toujours en cours d'exécution
`join()`	Attendre la terminaison d'une thread
`resume()`	Poursuivre l'exécution d'une thread suspendue
`run()`	Point d'entrée (d'exécution) de la thread
`sleep()`	Suspendre la thread pour une période de temps donnée
`start()`	Débuter une thread en appelant sa méthode `run()`
`suspend()`	Suspendre une thread

2.2 Création d'une thread

– Deux possibilités :
  – Implanter l'interface `Runnable`,

  - Créer une sous-classe de `Thread`
- Pour **implanter** `Runnable`, il y a seulement besoin d'implanter la méthode `run()` :

```
public abstract void run()
```

- Dans `run()`, on place les instructions de la thread à lancer.
- Ayant créé une classe implantant `Runnable`, on créé une instance de `Thread` dans cette classe :

```
Thread(Runnable objThread, String nomThread)
```

- La thread **ne débute pas** tant qu'on n'appelle pas `start()`.
- En fait `start()` effectue un appel à `run()`. C'est un *cadre logiciel*.

```
synchronized void start()
```

```
class NouvelleActivite implements Runnable {
    Thread th;

    NouvelleActivite() {
        th = new Thread(this, "Activite demo");
        System.out.println("Activite enfant : " + th);
        th.start();                  // Debuter l'activite
    }

    // Point d'entree de la nouvelle activite
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Activite enfant : " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Enfant interrompu.);
        }
        System.out.println("Sortie de l'activite enfant");
   }// run()
}// class NouvelleActivite

class ActiviteDemo {
    public static void main(String args[]) {
        new NouvelleActivite();

        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Activite parente : " + i);
```

```
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Parent interrompu.);
        }
        System.out.println("Sortie de l'activite parent");
   }// run()
}// class Activite
```

2.3 Création d'une thread : hériter de Thread

– Deuxième possibilité : **créer une sous-classe de** `Thread`, qui redéfinit la méthode `run()`. Même exemple :

```
class NouvelleActivite extends Thread {

    NouvelleActivite() {
        super("Activite demo");
        System.out.println("Activite enfant : " + this);
        start();                 // Debuter l'activite
    }

    // Point d'entree de la nouvelle activite
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Activite enfant : " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Enfant interrompu.);
        }
        System.out.println("Sortie de l'activite enfant");
   }// run()
}// class NouvelleActivite

class ActiviteDemo {
    public static void main(String args[]) {
        new NouvelleActivite();

        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Activite parente : " + i);
```

```
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Parent interrompu.);
        }
        System.out.println("Sortie de l'activite parent");
    }// run()
}// class Activite
```

- Que choisir entre sous-classer `Thread` et implanter `Runnable` ? La classe `Thread` définit plusieurs méthodes pouvant être redéfinies par des classes dérivées ; la seule qui **doit l'être** est `run()`.
- Divers programmeurs java estiment qu'on étend (`extends`) une classe seulement pour la modifier ou la compléter. ⇒ Si l'on ne doit pas **redéfinir d'autres méthodes que** `run()`, **plutôt implanter** `Runnable`.

2.4 Méthodes utilitaires

- Ci-dessus figure `sleep()` :

  ```
  static void sleep(long millisec) throws InterruptedException
  ```

  qui suspend l'exécution de la thread appelante pendant `millisec` milisecondes. Elle peut déclencher une exception
- Pour obtenir et changer le nom d'une thread, on dispose de :

  ```
  final void setName(String nomThread)
  final String getName()
  ```

- Deux moyens pour savoir quand une thread est terminée :

  ```
  final boolean isAlive() throws InterruptedException
  ```

  qui renvoie `true` si la thread appelante est toujours en cours d'exécution ; elle renvoie `false` sinon ;

  ```
  final void join() throws InterruptedException
  ```

  qui attend que la thread appelante se termine.
- Suspension et poursuite d'exécution :

  ```
  final void suspend()
  final void resume()
  ```

- Obtention et modification de priorité :

  ```
  final void setPriority(int niveau)
  ```

  qui modifie la priorité à `niveau`, ce dernier étant entre `MIN_PRIORITY` et `MAX_PRIORITY` (qui sont souvent 1 et 10) ; la priorité par défaut est `NORM_PRIORITY` (généralement 5) ;

```
final int getPriority()
```

- Sous Solaris, il y a un comportement multi-tâches non préemptif (les tâches plus prioritaires doivent laisser le contrôle aux moins prioritaires pour que ces dernières aient une chance de tourner). ⇒ Pour avoir un comportement prédictible sur diverses plates-formes, utiliser des threads qui laissent volontairement le contrôle aux autres.

I.3 Synchronisation

3.1 Synchronisation

- Lorsque 2 threads ou plus ont besoin d'une même ressource au même moment, il y a besoin de s'assurer que la ressource ne sera utilisée que par une thread en même temps : on utilise alors un procédé de *synchronistaion*.
- Un moyen d'obtenir une synchronisation : les moniteurs.
- Un moniteur est un mécanisme utilisé comme un verrou d'exclusion mutuelle, un *mutex*.
- Lorsqu'une thread acquiert un verrou, on dit qu'elle *entre dans* le moniteur.
- Les autres threads sont dites *en attente* du moniteur.
- Analogie avec des cabines d'essayage de vêtements.
- Deux moyens de synchronisation : les méthodes synchronisées et les blocs synchronisés.

3.2 Méthodes synchronisées

- Pour entrer dans le moniteur d'un objet, appeler une méthode modifiée avec le mot clé `synchronized`.
- Lorsqu'une thread est dans une méthode synchronisée, toute autre thread qui essaie de l'appeler (elle ou toute autre méthode synchronisée) doit attendre.
- Pour sortir du moniteur, celui qui y est entré revient simplement de la méthode synchronisée.
- Exemple de programme non synchronisé. Dans une classe `Callme`, la méthode `call()` essaie d'afficher une chaîne `String` passée en paramètre et ce, entre crochets. Elle fait une pause d'une seconde (`sleep(1000)`) entre la fin de la chaîne et le crochet fermant.

```
// Programme non synchronise
class Callme {
    void call(String msg) {
        System.out.print("[" + msg);
```

```
        try {
            Thread.sleep(1000);
        } catch(InterruptedException e) {
            System.out.println("Interrompue");
        }
        System.out.println("]");
    }
}// class Callme

class Caller implements Runnable {
    String msg;
    Callme cible;
    Thread th;

    public Caller(Callme cib, String ch) {
        cible = cib;
        msg = ch;
        th = new Thread(this);
        th.start();
    }

    public void run() {
        cible.call(msg);
    }
}// class Caller

class Synch {
    public static void main(String args[]) {
        Callme cible = new Callme();
        Caller activ1 = new Caller(cible, "Bonjour");
        Caller activ2 = new Caller(cible, "monde");
        Caller activ1 = new Caller(cible, "synchronise");

        // On attend que les activites se terminent
        try {
            activ1.th.join();
            activ2.th.join();
            activ3.th.join();
        } catch(InterruptedException e) {
          System.out.println("Interrompue");
        }
    }
}// class Synch
```

- La sortie du programme est :

```
[Bonjour[monde[synchronise]
]
]
```

  En appelant `sleep()`, la méthode `call()` permet que l'exécution passe à une autre thread. Ce qui donne des affichages inter-mêlés.
- Ceci est une *condition de course*, les threads étant en compétition pour arriver à la fin de la méthode `call()`.
- Pour sérier les accès à `call()`, il faut restreindre son accès à une thread à la fois.
- Ceci se fait *via* le mot clé `synchronized` :

```
class Callme {
    synchronized void call(String msg) {
...
```

  La sortie du programme est alors

```
[Bonjour]
[monde]
[synchronise]
```

3.3 Instruction `synchronized`

- Supposez vouloir **synchroniser l'accès aux objets** d'une classe qui n'a pas été construite pour un accès multi-threads. En outre, vous n'avez pas au code source.
- Un appel synchronisé peut être réalisé dans un *bloc synchronisé* :

```
synchronized(objet) {
    // instructions a synchroniser
}
```

  où `objet` est une référence à l'objet à synchroniser.
- Un bloc synchronisé assure qu'un **appel à une méthode** quelconque d' `objet` **ne pourra se faire qu'une fois que la thread courante sera entrée dans le moniteur**.
- Autre version de l'exemple précédent, avec un bloc synchronisé à l'intérieur de la méthode `run()`

```
// Programme non synchronise
class Callme {
    void call(String msg) {
        System.out.print("[" + msg);
        try {
```

```
            Thread.sleep(1000);
        } catch(InterruptedException e) {
            System.out.println("Interrompue");
        }
        System.out.println("]");
    }
}// class Callme

class Caller implements Runnable {
    String msg;
    Callme cible;
    Thread th;

    public Caller(Callme cib, String ch) {
        cible = cib;
        msg = ch;
        th = new Thread(this);
        th.start();
    }

    public void run() {
        synchronized(cible) {   // bloc synchronise
            cible.call(msg);
        }
    }
}// class Caller


class Synch {
    public static void main(String args[]) {
        Callme cible = new Callme();
        Caller activ1 = new Caller(cible, "Bonjour");
        Caller activ2 = new Caller(cible, "monde");
        Caller activ1 = new Caller(cible, "synchronise");

        // On attend que les activites se terminent
        try {
            activ1.th.join();
            activ2.th.join();
            activ3.th.join();
        } catch(InterruptedException e) {
          System.out.println("Interrompue");
        }
```

```
    }
}// class Synch
```

3.4 Communication inter-threads

- Pas besoin de scrutation entre threads. Exemple d'un producteur et d'un consommateur avec la contrainte que le producteur doit attendre que le consommateur ait terminé avant qu'il puisse générer des données supplémentaires.
- Dans un système avec scrutation, le consommateur perd des cycles CPU à attendre que le producteur produise ; lorsque le producteur a terminé, il perd des cycles CPU à attendre que le consommateur ait terminé, et ainsi de suite.
- La scrutation est évitée avec un mécanisme de communication inter-threads accessible à travers 3 méthodes :
  - La méthode `wait()` : fait sortir la tâche appelante du moniteur et la met en sommeil jusqu'à ce qu'une autre thread entre dans le même moniteur et appelle `notify()` ;
  - La méthode `notify()` : réveille la première thread ayant appelé `wait()` sur le même objet ;
  - La méthode `notifyAll()` : réveille toutes les threads ayant appelé `wait()` sur le même objet ; la thread de priorité la plus élevée s'exécutera en premier.
- Ces méthodes sont implantées comme `final` dans `Object`, de sorte que toutes les classes en disposent. Ces trois méthodes ne peuvent être appelées qu'à l'intérieur d'une méthode synchronisée.
- Leur déclaration est :

  ```
  final void wait()
  final void notify()
  final void notifyAll()
  ```

  Il y a également la forme `final void wait(long timeout)` qui permet d'attendre un nombre donné de millisecondes.
- Exemple d'**implantation incorrecte** d'une forme simple du problème producteur/consommateur.
- Il y a 4 classes :
  - La classe `FileAttente`, la file dont on veut synchroniser les accès ;
  - La classe `Producteur`, l'objet thread qui produit les entrées de la file ;
  - La classe `Consommateur` qui les consomme ;
  - La classe `ProdCons`, la classe qui crée la file, le producteur et le consommateur.

```
// Implantation incorrecte d'un producteur et d'un consommateur
class FileAttente {
    int n;

    synchronized int prendre() {
        System.out.println("Pris : " + n);
        return n;
    }

    synchronized void mettre(int n) {
        this.n = n;
        System.out.println("Mis : " + n);
    }
}// class FileAttente

class Producteur implements Runnable {
    FileAttente f;

    Producteur(FileAttente f) {
        this.f = f;
        new Thread(this, "Producteur").start();
    }

    public void run() {
        int i = 0;

        while(true) {
            f.mettre(i++);
        }
    }
} // class Producteur

class Consommateur implements Runnable {
    FileAttente f;

    Consommateur(FileAttente f) {
        this.f = f;
        new Thread(this, "Consommateur").sart();
    }

    public void run() {
        while(true) {
            f.prendre();
```

```
        }
    }
} // class Consommateur

class ProdCons {
    public static void main(String args[]) {
        FileAttente f = new FileAttente();
        new Producteur(f);
        new Consommateur(f);

        System.out.println("Ctrl-C pour arreter.");
}// class ProdCons
```

– Bien que les méthodes `put()` et `get()` de `FileAttente` soient synchronisées, rien n'arrête ici le producteur de dépasser le consommateur et rien n'arrête non plus le consommateur de consommer la même valeur 2 fois. On pourra ainsi obtenir à l'écran, par exemple

```
Mis : 1
Pris : 1
Pris : 1
Pris : 1
Mis : 2
Mis : 3
Mis : 4
Pris : 4
```

– Le moyen de corriger cela est d'utiliser `wait()` et `notify()` pour signaler dans les deux directions :

```
// Implantation correcte d'un producteur et d'un consommateur
class FileAttente {
    int n;
    boolean valeurMise = false; // A-t-on mis une valeur dans la file ?

    synchronized int prendre() {
        if(valeurMise == false)
            try {
                wait();
            } catch(InterruptedException e) {
                System.out.println("InterruptedException interceptee");
            }
        System.out.println("Pris : " + n);
        ValeurMise = false;
        notify();
        return n;
```

```
    }

    synchronized void mettre(int n) {
        if(valeurMise == true)
            try {
                wait();
            } catch(InterruptedException e) {
                System.out.println("InterruptedException interceptee");
            }
        this.n = n;
        ValeurMise = true;
        System.out.println("Mis : " + n);
        notify();
    }
}// class FileAttente

class Producteur implements Runnable {
    FileAttente f;

    Producteur(FileAttente f) {
        this.f = f;
        new Thread(this, "Producteur").start();
    }

    public void run() {
        int i = 0;

        while(true) {
            f.mettre(i++);
        }
    }
} // class Producteur

class Consommateur implements Runnable {
    FileAttente f;

    Consommateur(FileAttente f) {
        this.f = f;
        new Thread(this, "Consommateur").sart();
    }

    public void run() {
        while(true) {
```

```
                f.prendre();
            }
        }
    } // class Consommateur

    class ProdCons {
        public static void main(String args[]) {
            FileAttente f = new FileAttente();
            new Producteur(f);
            new Consommateur(f);

            System.out.println("Ctrl-C pour arreter.");
    }// class ProdCons
```

Et l'on obtient ainsi à l'écran

```
Mis : 1
Pris : 1
Mis : 2
Pris : 2
Mis : 3
Pris : 3
Mis : 4
Pris : 4
```

I.4 Groupes de threads

4.1 Groupes de threads de l'API Java

- Les groupes de threads sont organisées en arbre. La racine de cet arbre est le groupe System (System thread group) ; son enfant est le groupe Main, qui contient la thread dite Main, exécutant la méthode `main()`.
- Dans le **groupe System**, on a 4 threads démons :
  - Gestionnaire d'horloge (Clock handler) Il gère les timeouts créés par des appels aux méthodes `sleep()` et `repaint()` (qui redessine le contenu d'une fenêtre).
  - Thread oisive (Idle thread) C'est une thread de priorité exceptionnellement basse : 0 (en fait, on ne peut fixer la priorité d'une thread à cette valeur *via* la méthode `setPriority()`). Elle ne tourne donc que lorsque toutes les autres threads sont bloquées. Cela permet en fait au ramasse-miettes de savoir quand il peut faire la chasse aux objets qui ne sont plus référencés.

  - Ramasse-miettes (Garbage collector) Il inspecte les objets au sein de la machine virtuelle, teste s'ils sont toujours référencés et libère la place mémoire correspondante si ce n'est pas le cas. Par défaut, le ramasse-miettes dort pendant une seconde ; lorsqu'il se réveille, il regarde si la thread oisive est en cours d'exécution. Si oui, il suppose que le système n'a rien à faire et il commence à inspecter la mémoire ; sinon, il se rendort pendant une seconde. Le ramasse-meittes a une priorité égale à 1, tout comme la thread de finalisation.
  - Thread de finalisation (Finalizer thread) Elle appelle la méthode `finalize()` sur les objets libérés par le ramasse-miettes. Elle tourne à priorité 1.
- Dans le **groupe Main**, on a 4 threads, dont **3 ne seront actives que si l'application utilise des composants graphiques** (paquetages du JFC comme `java.awt`, etc.) :
  - Thread Main (Main thread) La thread par défaut de ce groupe exécute la méthode `main()`.
  - Thread AWT-Input Ne tourne que si l'application utilise des composants graphiques. Gère les entrées du système de fenêtrage sous-jacent et effectue les appels nécessaires aux méthodes de gestion d'événements de l'AWT (Abstract Window Toolkit).
  - Thread AWT-Toolkit Gère les événements du système de fenêtrage sous-jacent et appelle les méthodes appropriées. Cette thread est, dans AWT 1.1, spécifique à la plateforme et se nomme AWT-Motif (sur UNIX), AWT-Windows, etc. Certaines plateformes utilisent deux threads au lieu d'une.
  - Rafraîchisseur d'écran (ScreenUpdater) Gère les appels à la méthode `repaint()`. Lorsqu'un tel appel se produit, cette thread est prévenue et elle appelle les méthodes `update()` des composants concernés.

4.2 Groupe de threads `ThreadGroup`

- La classe `ThreadGroup` permet de créer un groupe de threads.
- 2 constructeurs :
  - Avec un paramètre : `ThreadGroup(String groupName)`
  - Avec deux : `ThreadGroup(ThreadGroup parentObj, String groupName)`

  Dans les 2 formes, `groupName` spécifie le nom du groupe de threads. La première version crée un nouveau groupe qui a la thread courante comme parent. Dans le 2 $^{\text{ième}}$ forme, le parent est spécifié par `parentObj`.
- Méthodes de `ThreadGroup` :

`methode()`	But

`static int activeCount()`	Renvoie le nombre de threads dans le groupe, plus le nombre de groupe dont cette thread est le parent.
`static int activeCountGroup()`	Renvoie le nombre de groupe dont cette thread est le parent.
`void checkAccess()`	Provoque la vérification par le gestionnaire de sécurité du fait que la thread appelante peut eccéder et/ou changer le groupe à partir duquel `checkAccess()` est appelé.
`void destroy()`	Détruit le groupe de threads (et tous les enfants) à partir duquel l'appel est effectué.
`static int enumerate(Thread group[])`	Les threads constituant le groupe appelant sont stockées dans le tableau `group`.
`static int enumerate(Thread group[], boolean all)`	Les threads constituant le groupe appelant sont stockées dans le tableau `group`. Si `all` est à `true`, les threads de tous les sous groupes de la thread appelante sont également stockés dans `group`.
`static int enumerate(ThreadGroup group[])`	Les threads constituant le groupe appelant sont stockées dans le tableau `group`.
`static int enumerate(ThreadGroup group[], boolean all)`	Les threads constituant le groupe appelant sont stockées dans le tableau `group`. Si `all` est à `true`, les sous groupes de tous les sous groupes (et ainsi de suite) de la thread appelante sont également stockés dans `group`.
`final int getMaxPriority()`	Renvoie la plus haute priorité d'un membre du groupe.
`final String getName()`	Renvoie le nom du groupe.
`final ThreadGroup getParent()`	Renvoie `null` si le `ThreadGroup` appelant n'a pas de parent. Sinon, renvoie le parent de l'objet invoqué.
`final boolean isDaemon()`	Renvoie `true` si le groupe est un groupe de démons. Sinon, renvoie `false`.
`void list()`	Affiche des informations sur le groupe.
`final boolean parentOf(ThreadGroup group)`	Renvoie `true` si la thread appelante est le parent de `group` (ou `group` lui-même). Sinon, renvoie `false`.
`final void resume()`	Fait reprendre l'exécution à toutes les threads du groupe appelant.
`final void setDaemon(boolean isDaemon)`	Si `isDaemon` est égal à `true`, le groupe appelant est étiqueté comme un groupe de démons.

`final void setMaxPriority(int priority)`	Fixe la priorité maximale du groupe appelant à `priority`.
`final void stop()`	Termine toutes les threads du groupe appelant.
`final void suspend()`	Suspend toutes les threads du groupe appelant.
`String toString()`	Renvoie un équivalent `String` du groupe appelant.
`void uncaughtException(Thread thread, Throwable e)`	Cette méthode est appelée lorsqu'une excpetion n'est pas levée.

4.3 Exemple de groupe de threads

- L'exemple suivant liste (méthode `listAllThreads()` les threads (et leurs groupes) s'exécutant.
- La méthode `listAllThread()` utilise `currentThread()` (de la classe `Thread`) pour obtenir la thread courante, et utilise `getThreadGroup()` pour trouver le groupe de cette thread.
- Elle utilise ensuite `getParent()` de `ThreadGroup` pour se déplacer dans la hiérarchie des groupes de threads jusqu'à trouver le groupe racine, qui contient tous les autres groupes.
- La méthode `listAllThreads()` appelle ensuite la méthode privée `list_group()` de `ThreadLister` pour afficher le contenu du groupe de threads racine, et pour afficher récursivement le contenu de tous les groupes qu'elle contient.
- La méthode `list_group()`, et la méthode `print_thread_info()` qu'elle appelle, utilisent diverses méthodes de `Thread` et de `ThreadGroup` pour obtenir des informations sur les threads et sur leurs groupes.
- Remarquez que la méthode `isDaemon()` détermine si une méthode est un démon ou pas. Une thread démon est censée s'exécuter en arrière-plan et ne pas se terminer. L'interpréteur Java termine lorsque toutes les threads non démon ont terminé.

```
import java.io.*;
import java.applet.*;
import java.awt.*;


class ThreadLister {
    // Afficher des informations a propos d'une activite
    private static void print_thread_info(PrintStream out, Thread t,
```

```
                      String indent) {
        if (t == null) return;
        out.println(indent + "Activite : " + t.getName() +
                "  Priorite : " + t.getPriority() +
                (t.isDaemon()?" Demon":"") +
                (t.isAlive()?"":" Non vivante"));
    }

    // Afficher des informations a propos d'un groupe d'activite, de
    //   ses activites et groupes
    private static void list_group(PrintStream out, ThreadGroup g,
                   String indent) {
        if (g == null) return;
        int num_threads = g.activeCount();
        int num_groups = g.activeGroupCount();
        Thread[] threads = new Thread[num_threads];
        ThreadGroup[] groups = new ThreadGroup[num_groups];

        g.enumerate(threads, false);
        g.enumerate(groups, false);

        out.println(indent + "Groupe d'activites : " + g.getName() +
                "  Priorite max : " + g.getMaxPriority() +
                (g.isDaemon()?" Demon":""));

        for(int i = 0; i < num_threads; i++)
            print_thread_info(out, threads[i], indent + "    ");
        for(int i = 0; i < num_groups; i++)
            list_group(out, groups[i], indent + "    ");
    }

    // Trouver la racine du groupe d'activites et lister recursivement
    public static void listAllThreads(PrintStream out) {
        ThreadGroup current_thread_group;
        ThreadGroup root_thread_group;
        ThreadGroup parent;

        // Obtenir le groupe d'activites courant
        current_thread_group = Thread.currentThread().getThreadGroup();

        // Aller chercher la racine des groupes d'activites
        root_thread_group = current_thread_group;
        parent = root_thread_group.getParent();
```

```
        while(parent != null) {
            root_thread_group = parent;
            parent = parent.getParent();
        }

        // Et le lister, recursivement
        list_group(out, root_thread_group, "");
    }


    public static void main(String[] args) {
        ThreadLister.listAllThreads(System.out);
    }

}// class ThreadLister


public class AppletThreadLister extends Applet {
    TextArea textarea;

    // Creer une zone de texte pour y mettre les informations
    public void init() {
        textarea = new TextArea(20, 60);
        this.add(textarea);
        Dimension prefsize = textarea.preferredSize();
        this.resize(prefsize.width, prefsize.height);
    }

    // Effectuer l'affichage. ByteArrayOutputStream est bien utile
    public void start() {
        ByteArrayOutputStream os = new ByteArrayOutputStream();
        PrintStream ps = new PrintStream(os);
        ThreadLister.listAllThreads(ps);
        textarea.setText(os.toString());
    }
}
```

I.5 États d'une thread et ordonnancement

5.1 Descriptif des états d'une thread

Une thread de Java peut se trouver dans l'un des 7 états qui suivent.

- **nouveau** (new) Lorsqu'une nouvelle thread est créée, par exemple avec
  `Thread th = new Thread(a);`
- **prêt** (runnable) ou prête à s'exécuter. Lorsque la méthode `start()` d'une thread est appelée, cette dernière passe dans l'état prêt. Toutes les threads prêtes d'un programme Java sont organisées par la machine virtuelle en une structure de données nommée la file d'attente prête. Une thread entrant dans l'état prêt est mise dans cette file. Le code de la méthode `run()` sera appelé à l'exécution de la thread.
- **en cours d'exécution** (running) La thread se voit allouée des cycles CPU par l'ordonnanceur. Si une thread dans létat en cours d'exécution appelle sa méthode `yield()`, ou si elle est supplantée par une thread de priorité plus élevée, elle laisse la CPU et est remise dans la file d'attente prête, entrant dans l'état prêt.
- **suspendu** (suspended) Une thread prête ou en cours d'exécution entre dans l'état suspendu lorsque sa méthode `suspend()` est appelée. Une thread peut se suspendre elle-même ou être par une autre. De l'état suspendu, une thread ré-entre dans l'état prêt lorsque sa méthode `resume()` est appelée par une autre thread.
- **bloqué** (blocked) Une thread entre dans l'état bloqué lorsque
  - elle appelle sa méthode `sleep()`,
  - elle appelle `wait()` dans une méthode synchronisée d'un objet,
  - elle appelle `join()` sur un autre objet dont la thread ne s'est pas encore terminée,
  - elle effectue une opération d'entrée/sortie bloquante (comme lire à partir du clavier).

  À partir de l'état bloqué, une thread ré-entre dans l'état prêt.
- **suspendu-bloqué** (suspended-blocked) Losqu'une thread est bloquée est suspendue par une autre thread. Si l'opération bloquante se termine, la thread entre dans l'état suspendu. Si la thread est réintégrée (resumed) avant que l'opération bloquante ne se termine, elle entre dans l'état bloqué.
- **mort** (dead) Une thread se termine et entre dans l'état mort lorsque sa méthode `run()` se termine ou lorsque sa méthode `stop()` est appelée.

5.2 Récapitulatif des états d'une thread

Le tableau suivant résume les différents états et les événements qui engendrent une transition d'un état à un autre

	Nouvel état					
État actuel	prêt	en cours	suspendu	bloqué	suspendu-bloqué	mort
nouveau	`start()`					
prêt		élu[1]	`suspend()`			`stop()`
en cours[2]	quantum épuisé, `yield()`		`suspend()`	sur E/S, `sleep()`, `wait()`, `join()`		`stop()` `run()` terminé
suspendu	`resume()`					`stop()`
bloqué	E/S terminée, `sleep()` expire, `notify()`, `notifyAll()`, `join()` terminé				`suspend()`	`stop()`
suspendu-bloqué			E/S terminée	`resume()`		`stop()`

5.3 Ordonnancement

- L'ordonnaceur s'assure que la thread de plus haute priorité (ou les s'il y en a plusieurs d'égale priorité) est exécutée par la CPU.
- Si une thread de plus haute priorité entre dans la file des threads prêtes, l'ordonnanceur de threads Java met la thread en cours dans la file d'attente, pour que la prioritaire s'exécute.
- La thread en cours est dite supplantée *par préemption* par la thread de plus haute priorité.
- Si la thread en cours d'exécution passe la main (*via* `yield()`, est suspendue ou bloquée, l'ordonnanceur choisit dans la file d'attente prête la thread de priorité la plus élevée (ou l'une d'entre elles si elles sont plusieurs) pour être exécutée.
- Un autre aspect est le partage du temps (time slicing). Un ordre de grandeur courant pour un quantum de temps est 100ms. En temps partagé, si une thread en cours d'exécution n'a pas été stoppée, bloquée, suspendue, ne s'est pas terminée, ou n'a pas passé la main, et s'il y a d'autres threads de priorité égale dans la file d'attente prête, la CPU est réallouée à l'une d'entre elles.
- ☛ Sous Solaris 2.x, la version 1.1 du JDK n'effectue pas de partage de temps entre les threads : **une thread s'exécute jusqu'à ce qu'elle se termine,**

[1](par l'ordonnanceur)
[2](d'exécution)

soit stoppée, suspendue, bloquée, passe la main ou qu'une autre thread de plus haute priorité devienne prête.
- Le JDK 1.1 pour Windows 95/NT partage le temps entre les threads.

5.4 Résumé pour la création d'une thread

- Les constructeurs d'une `Thread` acceptent comme arguments :
  - Un objet `Runnable`, auquel cas un `start()` ultérieur appelle `run()` de l'objet `Runnable` fourni. Si aucun `Runnable` n'est fourni, l'implantation par défaut de `run()` retourne immédiatement.
  - Une `String` servant d'identificateur de la thread. N'est utile que pour le traçage ou le debugging.
  - Le `ThreadGroup` dans laquelle la nouvelle `Thread` doit être placée. Si l'accès au `ThreadGroup` n'est pas autorisé, une `SecurityException` est levée.

II – Génie logiciel appliqué

Références bibliographiques

- *Object-Oriented Analysis and Design with Applications*, G. Booch [Boo94],
- *Flexible Java*, B. Venners [Ven99],
- *Design Patterns*, E. Gamma, R. Helm, R. Johnson et J. Vlissides [GHJV95],
- *Data Structures*, M.A. Weiss [Wei98].

II.1 Implantation d'une pile en Java

1.1 Implantation Java d'une pile : interface

Nous avons besoin (par souci de commodité) d'une exception de type `Underflow` que l'on définit per exemple comme suit :

```
/**
 * Classe Exception pour l'acces dans des conteneurs vides
 * tels des piles, listes et files d'attente avec priorite
 */
public class Underflow extends Exception
{
    /**
     * Construit cet objet exception
     * @param message le message d'erreur.
     */
    public Underflow( String message )
    {
        super( message );
    }
}
```

Voici l'interface d'une pile

```
package DataStructures;

import Exceptions.*;

// Stack interface
//
// ****************** OPERATIONS PUBLIQUES *************************
// void push( x )          --> Inserer x
// void pop( )             --> Enlever l'elt insere le + recemment
// Object top( )           --> Renvoyer l'elt insere le + recemment
// Object topAndPop( )     --> Renvoyer et enlever l'elt le + receent
// boolean isEmpty( )      --> Renvoyer true si vide; false sinon
// void makeEmpty( )       --> Enlever tous les elements
// ****************** ERREURS ************************************
// top, pop, ou topAndPop sur une pile vide

/**
 * Protocole pour piles

 */
public interface Stack
{
    /**
     * Tester si la pile est logiquement vide
     * @return true si vide, false sinon.
     */
    boolean isEmpty( );

    /**
     * Prendre l'element insere le plus recemment sur la pile.
     * n'altere pas la pile.
     * @return l'element le plus recemment insere dans la pile
     * @exception Underflow si la pile est vide
     */
    Object  top( )       throws Underflow;

    /**
     * Enleve l'element insere le plus recemment sur la pile
     * @exception Underflow si la pile est vide
     */
    void    pop( )       throws Underflow;

    /**
```

```
     * Renvoie et enleve l'element le plus recemment insere
     * de la pile.
     * @return l'element le plus recemment insere dans la pile
     * @exception Underflow si la pile est vide
     */
    Object  topAndPop( ) throws Underflow;

    /**
     * Insere un nouvel element sur la pile
     * @param x l'element a inserer.
     */
    void    push( Object x );

    /**
     * Rendre la pile logiquement vide
     */
    void    makeEmpty( );
}
```

1.2 Implantation Java d'une pile par tableau

Voici maintenant l'implantation par tableau d'une pile. Notez que le tableau croît de manière dynamique.

```
package DataStructures;

import Exceptions.*;


// StackAr class
//
// CONSTRUCTION : sans constructeur
//

// ****************** OPERATIONS PUBLIQUES *************************
// void push( x )         --> Inserer x
// void pop( )            --> Enlever l'elt insere le + recemment
// Object top( )          --> Renvoyer l'elt insere le + recemment
// Object topAndPop( )    --> Renvoyer et enlever l'elt le + receent
// boolean isEmpty( )     --> Renvoyer true si vide; false sinon
// void makeEmpty( )      --> Enlever tous les elements
// ****************** ERREURS ***********************************
// top, pop, ou topAndPop sur une pile vide
```

```
/**
 * Implantation d'une pile a partir d'un tableau
 */
public class StackAr implements Stack
{
    /**
     * Construire la pile
     */
    public StackAr( )
    {
        theArray = new Object[ DEFAULT_CAPACITY ];
        topOfStack = -1;
    }


    public boolean isEmpty( )
    {
        return topOfStack == -1;
    }


    public void makeEmpty( )
    {
        topOfStack = -1;
    }


    public Object top( ) throws Underflow
    {
        if( isEmpty( ) )
            throw new Underflow( "Stack top" );
        return theArray[ topOfStack ];
    }


    public void pop( ) throws Underflow
    {
        if( isEmpty( ) )
            throw new Underflow( "Stack pop" );
        topOfStack--;
    }
```

```
    public Object topAndPop( ) throws Underflow
    {
        if( isEmpty( ) )
            throw new Underflow( "Stack topAndPop" );
        return theArray[ topOfStack-- ];
    }

    public void push( Object x )
    {
        if( topOfStack + 1 == theArray.length )
            doubleArray( );
        theArray[ ++topOfStack ] = x;
    }

    /**
     * Methode interne pour etendre theArray.
     */
    private void doubleArray( )
    {
        Object [ ] newArray;

        newArray = new Object[ theArray.length * 2 ];
        for( int i = 0; i < theArray.length; i++ )
            newArray[ i ] = theArray[ i ];
        theArray = newArray;
    }

    private Object [ ] theArray;
    private int        topOfStack;

    static final int DEFAULT_CAPACITY = 10;
}
```

1.3 Implantation Java d'une pile par liste

Nous avons d'abord besoin d'éléments d'une liste

```
// Moeud de base stocke dans une liste chainee

class ListNode
{
        // Constructeurs
    ListNode( Object theElement )
    {
```

```
        this( theElement, null );
    }

    ListNode( Object theElement, ListNode n )
    {
        element = theElement;
        next    = n;
    }

        // Donnees accessibles par d'autres classes du meme paquetage
    Object   element;
    ListNode next;
}
```

Voici maintenant l'implantation par liste d'une pile

```
package DataStructures;

import Exceptions.*;

// StackLi class
//
//
// CONSTRUCTION : sans initialiseur
//

// ****************** OPERATIONS PUBLIQUES *************************
// void push( x )         --> Inserer x
// void pop( )            --> Enlever l'elt insere le + recemment
// Object top( )          --> Renvoyer l'elt insere le + recemment
// Object topAndPop( )    --> Renvoyer et enlever l'elt le + receent
// boolean isEmpty( )     --> Renvoyer true si vide; false sinon
// void makeEmpty( )      --> Enlever tous les elements
// ****************** ERREURS ***************************************
// top, pop, ou topAndPop sur une pile vide


/**
 * Implantation d'une pile sur une liste
 */
public class StackLi implements Stack
{
    /**
     * Construire la pile
```

```
     */
    public StackLi( )
    {
        topOfStack = null;
    }


    public boolean isEmpty( )
    {
        return topOfStack == null;
    }

    public void makeEmpty( )
    {
        topOfStack = null;
    }


    public Object top( ) throws Underflow
    {
        if( isEmpty( ) )
            throw new Underflow( "Stack top" );
        return topOfStack.element;
    }


    public void pop( ) throws Underflow
    {
        if( isEmpty( ) )
            throw new Underflow( "Stack pop" );
        topOfStack = topOfStack.next;
    }


    public Object topAndPop( ) throws Underflow
    {
        if( isEmpty( ) )
            throw new Underflow( "Stack topAndPop" );

        Object topItem = topOfStack.element;
        topOfStack = topOfStack.next;
        return topItem;
    }
```

```
    public void push( Object x )
    {
        topOfStack = new ListNode( x, topOfStack );
    }

    private ListNode topOfStack;
}
```

1.4 Génie logiciel pratique et POO

Nous allons maintenant détailler diverses techniques

- Minimiser le couplage entre méthodes
- Maximiser la cohésion

Une règle générale qui ne pourra qu'augmenter la fiabilité : "**Ne faites pas aux autres programmeurs (qui vont devoir maintenir votre code) ce que vous n'aimeriez pas que l'on vous fasse (si vous deviez maintenir leur code)**"

II.2 Minimisation de couplage entre méthodes

2.1 Types de méthodes

Pour réaliser le but visé, détaillons trois types de méthodes en java :

- Les méthodes utilitaires
- Les méthodes de consultation d'état
- Les méthodes de modification d'état

L'état d'une classe sera représenté par ses variables d'instance

2.2 Méthodes utilitaires

Ce type de méthode n'utilise ni ne modifie l'état de l'objet. Par exemples :

- Dans la classe `Integer`, `public static int toString(int i)` : renvoie un nouvel objet `String` représentant l'entier spécifié en base 10.
- Dans la classe `Math`, `public static native double cos(double a)` : envoie le cosinus d'un angle.

2.3 Méthode de consultation d'état

Ce type de méthode renvoie une vue de l'état de l'objet sans le modifier. Par exemple :

- Dans la classe `Object`, `public String toString()` : renvoie une représentation sous forme de chaîne de caractères d'un objet.
- Dans la classe `Integer`, `public byte byteValue()` : renvoie la valeur d'un `Integer` sous forme d'un octet.
- Dans la classe `String`, `public int indexOf(int ch)` : renvoie l'indice dans la chaîne de la première occurence du caractère spécifié.

2.4 Méthode de modification d'état

Ce type de méthode est susceptible de modifier l'état de l'objet Exemples :

- Dans la classe `StringBuffer`, `public StringBuffer append(int i)` : ajoute la représentation sous forme de chaîne de l' `int` au `StringBuffer`.
- Dans la classe `Hashtable`, `public synchronized void clear()` : vide la table de hachage, de façon qu'elle ne contienne plus de clés.
- Dans la classe `Vector`, `public final synchronized void addElement(Object obj)` : ajoute le composant spécifié à la fin du `Vector`, augmentant sa taille d'une unité.

2.5 Minimisation de couplage entre méthodes

- Le *couplage entre méthodes* est le degré d'interdépendance entre une méthode et son environnement (d'autres méthodes, objets et classes).
- Plus le degré de couplage est faible plus une méthode est indépendante et plus il est facile de la modifier (plus elle est **flexible**)
- Une technique simple pour estimer le degré de couplage est de voir une méthode comme un **transformateur de données** avec des données en entrée et des données en sortie (une vue systémique)

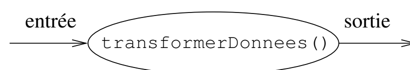


FIG. II.1: Une méthode comme transformateur de données

2.6 Entrées et sorties

Une méthode peut prendre ses entrées de diverses sources :

- De paramètres qui lui sont passés

- De données provenant de variables de classe (de sa propre classe ou d'une autre)
- S'il s'agit d'une méthode d'instance, de données provenant de variables d'instance de l'objet appelant

Une méthode peut avoir diverses sorties

- Elle peut renvoyer une valeur, soit un type primitif soit une référence à un objet
- Elle peut modifier des objets dont on a passé une référence en paramètre
- Elle peut modifier des variables de classe (de sa propre classe ou d'une autre)
- S'il s'agit d'une méthode d'instance, elle peut altérer les variables d'instance de l'objet appelant
- Elle peut lever une exception

2.7 Méthode utilitaire minimalement couplée

Une méthode la moins couplée possible en Java : méthodes utilitaires qui

- Ne prend des entrées que de ses paramètres
- N'effectue des sorties qu'au travers ses paramètres ou de sa valeur de retour (ou en levant une exception)
- N'accepte en entrée que des données réellement nécessaires à la méthode
- Ne renvoie comme sortie que des données effectivement produites par la méthode

2.8 Exemple de bonne méthode utilitaire

Voici par exemple la méthode `convertOzToMl()` qui accepte un `int` comme seule entrée et ne renvoie comme seule sortie qu'un `int` :

```
class Liquid {

     private static final double FL_OUNCES_PER_ML = 12.0/355.0;
     private static final double ML_PER_FL_OUNCE = 355.0/12.0;

     /**
     * Convertit des fluid ounces en millilitres
     */
     public static int convertOzToMl(int ounces) {

         double d = ounces * ML_PER_FL_OUNCE;
         d += 0.5;       // On doit ajouter .5 car (int) tronque
         return (int) d; // Resultat tronque si la fraction est >= .5
     }
 }
```

En écrivant une méthode utilitaire, on doit essayer de ne prendre des entrées que des paramètres et n'effectuer de sorties qu'au travers de paramètres, de valeurs de retour ou d'exceptions.

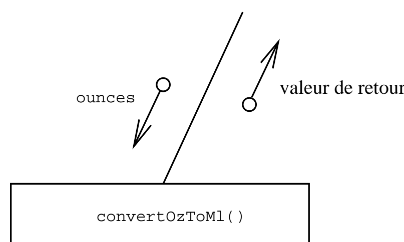


Fig. II.2: Une bonne méthode utilitaire (minimalement couplée)

2.9 Exemple de mauvaise méthode utilitaire

Un mauvais exemple qui accroît le couplage et d'avoir des données en entrée ou sortie qui ne sont pas vitales à la méthode.

Par exemple, supposons qu'en écrivant `convertOzToMl()`, vous pensez toujours placer sa sortie dans un objet `CoffeeCup` :

```
class Example2 {

    public static void main(String[] args) {

        CoffeeCup cup = new CoffeeCup();
        int amount = Liquid.convertOzToMl(16);
        cup.add(amount);
        //...
    }
}
```

Si c'est le cas, vous pourriez être tenté d'écrire `convertOzToMl()` comme suit :

```
class Liquid {

    private static final double FL_OUNCES_PER_ML = 12.0/355.0;
    private static final double ML_PER_FL_OUNCE = 355.0/12.0;


    public static void convertOzToMl(int ounces, CoffeeCup cup) {

        double d = ounces * ML_PER_FL_OUNCE;
```

```
        d += 0.5;
        cup.add((int) d);
    }
}
```

Et l'utilisation que vous en feriez serait alors :

```
class Example3 {

    public static void main(String[] args) {

        CoffeeCup cup = new CoffeeCup();
        Liquid.convertOzToMl(16, cup);
        //...
    }
}
```

Mais, maintenant, `convertOzToMl()` est couplée à la classe `CoffeCup`. Si quelqu'un veut plus tard convertir des onces en millilitres sans se servir d'une tasse de café, il lui faudra réécrire la méthode, écrire une méthode différente ou créer un objet `CoffeCup` juste pour contenir la sortie.

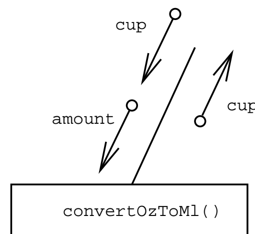


FIG. II.3: Une mauvaise méthode utilitaire

2.10 Exemple horrible

Un exemple horrible qui utilise en entrée comme en sortie des variables `public static` (l'équivalent de variables globales du C ++)

```
class Liquid {

    private static final double FL_OUNCES_PER_ML = 12.0/355.0;
    private static final double ML_PER_FL_OUNCE = 355.0/12.0;


    public static void convertOzToMl() {
```

```
        double d = PurpleZebra.k * ML_PER_FL_OUNCE;
        d += 0.5;
        FlyingSaucer.q = (int) d;
    }
}

class FlyingSaucer {
        public static int q;
        //...
}

class PurpleZebra {
    public static int k;
    //...
}
```

Pour utiliser cette version il faut :

```
class Example4 {

    public static void main(String[] args) {

        PurpleZebra.k = 16;
        Liquid.convertOzToMl();
        int mlFor16Oz = FlyingSaucer.q;

        System.out.println("Ml for 16 oz is: " + mlFor16Oz);
    }
}
```

- L'utilisation de cette version nécessite de connaître pas mal de détails internes d'implantation de la méthode.
- Par contraste, dans une bonne méthode utilitaire, le programmeur qui s'en sert peut comprendre comment l'utiliser **simplement en regardant la signature de la méthode et son type de retour** (ce qui est spécifié dans une interface).
- De plus, la variable `q`, si elle est utilisée ailleurs, peut provoquer des **effets de bords** très génants.

2.11 Méthodes de consultation d'état minimalement couplées

Une méthode de consultation de l'état minimalement couplée

- Ne prend en entrée que ses paramètres, les variables de classe (de la classe contenant la méthode), plus (s'il s'agit d'une méthode d'instance) les variables d'instance de l'objet appelant
- N'effectue ses sorties qu'au travers de sa valeur de retour, de ses paramètres, ou en levant une exception
- Ceci peut être reformulé en disant : l'entrée d'une méthode de consultation d'état minimalement couplée peut venir de n'importe où, excepté directement de variables de classe non constantes déclarées dans d'autres classes.

2.12 Méthodes de modification d'état minimalement couplées

- Ne prend en entrée que ses paramètres, les variables de classe (de la classe contenant la méthode) et (s'il s'agit d'une méthode d'instance) les variables d'instance de l'objet appelant
- N'effectue ses sorties qu'au travers de sa valeur de retour, de ses paramètres, en levant une exception, ou (s'il s'agit d'une méthode d'instance) au travers des variables d'instance de l'objet appelant
- Ceci peut être reformulé en disant : l'entrée d'une méthode de modification d'état minimalement couplée peut venir de n'importe où, excepté directement de variables de classe non constantes déclarées dans d'autres classes.
- La sortie d'une méthode de modification d'état minimalement couplée peut s'exprimer de n'importe quelle façon, excepté en modifiant directement des données déclarées ou référencées à partir de variables de classes déclarées dans d'autres classes.
- La méthode `add()` de `CoffeCup` en est un exemple :

```
class CoffeeCup {

     private int innerCoffee;

     public void add(int amount) {
         innerCoffee += amount;
     }
     //...
}
```

2.13 Remarque sur la référence cachée `this`

- Les méthodes d'instance peuvent accéder aux variables d'instance, car elles reçoivent une référence cachée `this` comme premier paramètre

- Le compilateur Java insère une référence `this` au début de la liste des paramètres de toute méthode d'instance qu'il compile
- Cette référence cachée `this` n'est pas passée aux méthodes de classe (déclarées `static`) ; c'est pourquoi l'on n'a pas besoin d'instance pour les invoquer et pourquoi elles ne peuvent accéder aux variables d'instances

2.14 Directives de couplage minimal

- **Directives pour les champs** Ne pas assigner de signification spéciale à des valeurs spécifiques d'un champ en représentant plusieurs attributs au sein du même champ. Utiliser au lieu de cela un champ séparé pour représenter chaque attribut d'une classe ou d'un objet.
- **Le corollaire constant** Préférer des constantes (champs `static final`) à des littéraux codés en dur (`1.5`, `"Coucou !"`), particulièrement si la même constante doit être utilisée en plusieurs endroits.
- **Le mantra de la minimisation de couplage** Toujours essayer de minimiser le couplage des méthodes : ne prendre comme entrées que les données nécessaires à la méthode ; n'effectuer comme sortie que les données produites par la méthode.

II.3 Maximisation de cohésion des méthodes

3.1 Notion de cohésion

- Les méthodes effectuent des opérations.
  - À bas niveau, ce sont des opérations comme accepter des données en entrée, opérer sur les données et déliver des données en sortie
  - À haut niveau, cela sera "cloner cet objet", "ajouter cet élément à la fin de ce vecteur", "voir tel enregistrement de la base de données des clients"
- Minimiser le couplage nécessite une vision bas niveau des méthodes
- Le couplage mesure comment les entrées et les sorties d'une méthode sont connectées à d'autres parties du programme
- Par contraste, maximiser la cohésion nécessite de regarder les méthodes à un haut niveau
- La *cohésion* mesure le degré avec lequel une méthode effectue tâche conceptuelle donnée.
- Plus une méthode est concentrée sur l'accomplissement d'une seule tâche conceptuelle, plus elle est cohésive

3.2 Pourquoi maximiser la cohésion ?

- Plus vos méthodes seront cohésives, plus votre code sera flexible (facile à comprendre et à modifier). Elles seront alors l'analogue de briques de base en LEGO.
- Les méthodes cohésives aident à la flexibilité sous deux angles :
  1/ Si votre code est plus focalisé sur une seule tâche conceptuelle, il est plus facile de lui trouver un nom qui indique clairement ce que la méthode fait.
  - Par exemple, une méthode nommée `convertOzToMl(int ounces)` est plus facile à comprendre que
    `convert(int fromUnits, int toUnits, int fromAmount)`

  2/ Les méthodes cohésives rendent votre code plus facile à changer lorsque chaque méthode effectue une seule tâche bien définie. On dispose de briques de base élémentaires à l'aide desquelles on peut contruire telle ou telle architecture logicielle. De plus, les changements de comportement que l'on peut être amené à faire au sein d'une méthode seront plus isolés si ce comportement est emballé dans sa propre méthode

3.3 Exemple de méthode peu cohésive

Voici un exemple de méthode peu cohésive sur les tasses à café

```
public class CoffeeCup {

     public final static int ADD = 0;
     public final static int RELEASE_SIP = 1;
     public final static int SPILL = 2;

     private int innerCoffee;

     public int modify(int action, int amount) {

         int returnValue = 0;

         switch (action) {
         case ADD:
             // ajouter des doses de cafe
             innerCoffee += amount;

             // renvoyer zero, meme si c'est sans signification
             break;

         case RELEASE_SIP:
```

```
                // enlever les doses de caffe passees avec amount
                int sip = amount;
                if (innerCoffee < amount) {
                    sip = innerCoffee;
                }
                innerCoffee -= sip;

                // renvoyer ce qui a ete enleve
                returnValue = sip;
                break;

            case SPILL:
                // mettre innerCoffee a 0
                // ignorer le parametre amount
                int all = innerCoffee;
                innerCoffee = 0;

                // renvoyer tout le cafe
                returnValue = all;

            default:
                // Il faut ici lever une exception, car une
                // commande invalide a ete passee
                break;
        }

        return returnValue;
    }
}
```

- La méthode `modify()` est peu cohésive parce qu'elle inclus du code pour effectuer des actions qui sont conceptuellement assez différentes
- Elle est difficile à comprendre en partie parce que son nom, `modify()`, n'est pas très spécialisé
- Une autre raison pour laquelle elle est difficile à comprendre est que certaines données qui lui sont passées ou qu'elle renvoie ne sont utilisées que dans certains cas.
  - Par exemple, si `action` est égal à `CoffeCup.ADD`, la valeur de retour de la méthode est sans signification
  - De même, si si `action` est égal à `CoffeCup.SPILL`, le paramètre d'entrée `amount` est inutilisé.

  Si l'on ne regarde qu'à la signature de la méthode et à sa valeur de retour, il n'est pas clair de deviner ce que la méthode va faire.

– Ce type de méthode est décrit par la figure II.4

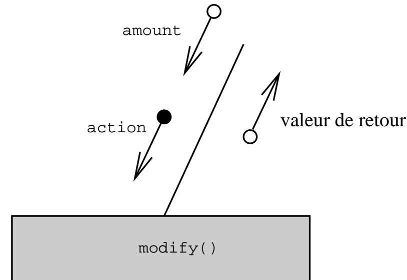


Fig. II.4: Passage de contrôle à `modify()`

– Le disque du paramètre `action` est noir, ce qui indique que ce paramètre contient des données utilisées pour du contrôle
– Les méthodes traitent des données fournies en entrée et génèrent des données en sortie. Lorsqu'une méthode utilise des données d'entrée non pour le traitement, mais pour décider comment traiter, ces données sont utilisées pour du contrôle.
– Afin de maximiser la cohésion, évitez de passer des données de contrôle à des méthodes ; au lieu de cela, essayez de diviser les fonctionnalités de la méthode en plusieurs méthodes qui n'ont pas besoin de données de contrôle.
– Il est par contre tout à fait indiqué de renvoyer des données de contôle d'une méthode. Lever une exception en est un bon exemple. Les données de contrôle devraient être passées d'une méthode à celle qui l'a appelée.

3.4 Exemple de méthode moyennement cohésive

– Afin de rendre `modify()` plus cohésive, scindons-la en deux : `add()` et `remove()`

```
public class CoffeeCup {

     private int innerCoffee;

     public void add(int amount) {
         innerCoffee += amount;
     }

     public int remove(boolean all, int amount) {

         int returnValue = 0;
```

```
            if (all) {
                int allCoffee = innerCoffee;
                innerCoffee = 0;
                returnValue = allCoffee;
            }
            else {
                // remove the amount of coffee passed as amount
                int sip = amount;
                if (innerCoffee < amount) {
                    sip = innerCoffee;
                }
                innerCoffee -= sip;
                returnValue = sip;
            }
           return returnValue;
        }
}
```

– Cette solution est meilleure, mais pas encore optimale. La méthode `add()` n'a pas besoin de paramètres de contrôle, mais `remove()` en a besoin.

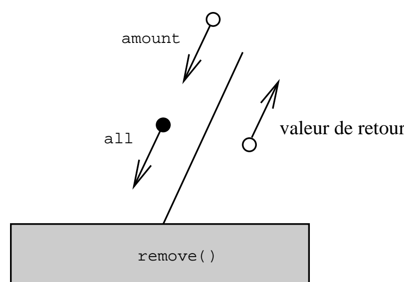


FIG. II.5: Passage de contrôle à `modify()`

3.5 Exemple de méthode hautement cohésive

– Afin d'obtenir un meilleure conception, divisons `remove()` en deux méthodes supplémentaires, aucune des deux n'acceptant des données de contrôle ou ayant des paramètres qui sont utilisés seulement une partie du temps. On divise donc en `releaseOneSip()` et `spillEntireContents()`

```
class CoffeeCup {

    private int innerCoffee;
```

```
    public void add(int amount) {
        innerCoffee += amount;
    }

    public int releaseOneSip(int sipSize) {
        int sip = sipSize;
        if (innerCoffee < sipSize) {
            sip = innerCoffee;
        }
        innerCoffee -= sip;
        return sip;
    }

    public int spillEntireContents() {
        int all = innerCoffee;
        innerCoffee = 0;
        return all;
    }
}
```

– Donc, au lieu d'exprimer ses souhaits au travers d'un paramètre supplémentaire, on appelle une méthode supplémentaire. Par exemple, au lieu de faire

```
class Example1 {

    public static void main(String[] args) {

        CoffeeCup cup = new CoffeeCup();
        // ignorer la valeur de retour erronnee de modify()
        //    dans le case ADD
        cup.modify(CoffeeCup.ADD, 355);
        int mlCoffee = cup.modify(CoffeeCup.RELEASE_SIP, 25);
        // 2ieme parametre inutilise dans le case SPILL
        mlCoffee += cup.modify(CoffeeCup.SPILL, 0);

        System.out.println("Ml of coffee: " + mlCoffee);
    }
}
```

on fera

```
class Example3 {

    public static void main(String[] args) {
```

```
        CoffeeCup cup = new CoffeeCup();
        cup.add(355);
        int mlCoffee = cup.releaseOneSip(25);
        mlCoffee += cup.spillEntireContents();

        System.out.println("Ml of coffee: " + mlCoffee);
    }
}
```

– Ce code est plus facile à comprendre, chaque méthode étant responsable d'une tâche bien déterminée.
– Il est de plus assez flexible, parce qu'il est plus aisé de changer une fonctionnalité sans affecter les autres

3.6 Suppositions réduites

– Les méthodes fonctionnellement cohésives augmentent la flexibilité du code également parce qu'elle font moins de suppositions sur l'ordre dans lequel les actions particulières sont effectuées.
– Voici un exemple fonctionnellement peu cohésif, parce qu'il fait trop de suppositions

```
class CoffeeCup {

    private int innerCoffee;
    private int innerCream;
    private int innerSugar;

    private static final int CREAM_FRACTION = 30;
    private static final int SUGAR_FRACTION = 30;

    public void add(int amountOfCoffee) {

        innerCoffee += amountOfCoffee;
        innerCream += amountOfCoffee/CREAM_FRACTION;
        innerSugar += amountOfCoffee/SUGAR_FRACTION;
    }
    //...
}
```

– La méthode `add()` augmente bien la quantité de café `innerCoffee`, mais suppose également que l'utilisateur voudra du sucre et de la crème, en quantité relative à la quantité de café.
– Une conception plus flexible est

```
class CoffeeCup {

     private int innerCoffee;
     private int innerCream;
     private int innerSugar;

     public void addCoffee(int amount) {
         innerCoffee += amount;
     }

     public void addCream(int amount) {
         innerCream += amount;
     }

     public void addSugar(int amount) {
         innerSugar += amount;
     }
     //...
}
```

3.7 La cohésion est haut niveau

- La cohésion représente le fait d'effectuer une opération à haut niveau, bien qu'à bas niveau, il puisse y avoir pas mal d'opérations effectuées.
- Par exemple, supposons avoir un consommateur régulier, Joe, qui veut toujours son café avec 30 doses de café, une dose de crème et une dose de sucre. Ceci peut s'implanter comme suit

```
class VirtualCafe {

     private static final int JOES_CREAM_FRACTION = 30;
     private static final int JOES_SUGAR_FRACTION = 30;

     public static void prepareCupForJoe(CoffeeCup cup, int amount) {

         cup.addCoffee(amount);
         cup.addCream(amount/JOES_CREAM_FRACTION);
         cup.addSugar(amount/JOES_SUGAR_FRACTION);
     }
}
```

- La méthode `prepareCupForJoe()` est fonctionnellement cohésive, bien quà bas niveau, elle effectue les mêmes actions que la précédente méthode `add()`, qui n'était pas cohésive.

– La raison est que, conceptuellement, `prepareCupForJoe()` prépare du café pour Joe, et pas pour tout le monde, comme le faisit `add()`. Elle n'empêche donc pas d'autres méthodes de se servir de `addCoffe()`, `addCream()`, `addSugar()`.

3.8 Explosion de méthodes

– Lorsque l'on divise les tâches entre diverses méthodes, il faut également prendre garde à ne pas générer un nombre trop important de méthodes
– Il y a donc un compromis à respecter entre la maximisation de la cohésion des méthodes et le fait de limiter les méthodes à un nombre raisonnable
– Il faut en tous les cas garder présent à l'esprit que lorsque vous programmerez en Java, vous ne dialoguerez pas (seulement) avec une machine virtuelle, vous communiquerez votre code à d'autres programmeurs.

3.9 Directives de maximisation de cohésion

– **Le mantra de la maximisation de cohésion** Toujours essayer de maximiser la cohésion des méthodes. Focaliser chaque méthode sur une tâche conceptuelle et lui donner un nom qui indique clairement la nature de cette tâche.
– **La directive d'observation-de-ce-qui-est-transmis** Éviter de passer des données de contrôle (pour décider comment réaliser le traitement) aux méthodes
– **Le principe d'aversion de l'explosion de méthode** Équilibrer la maximisation de cohésion de méthode (qui augmente le nombre de méthodes dans une classe) avec le maintien du nombre de méthodes à une valeur raisonnable
– **La règle d'or** Lors de la conception de logiciel, ne pas faire aux autres programmeurs (qui vont maintenir le logiciel) ce que l'on n'aimerait se voir faire (si l'on devait maintenir leur logiciel).

III – Programmation orientée objet en Java

Références bibliographiques

- *Understanding Object Oriented Programming with Java*, T. Budd [Bud98],
- *Design Patterns*, E. Gamma, R. Helm, R. Johnson et J. Vlissides [GHJV95].

0.10 Notion de paradigmes

- *Paradigme* : pour un étudiant du moyen-âge, **phrase exemple**. Peut être utilisée comme modèle ou comme aide dans l'apprentissage d'une langue.
- Langages vus comme une forme de littérature ...
- *Classe* :
  - **En-tête** (`public class nomClasse { ...}`)
  - Corps :
    - **champs de données** ou
    - **méthodes**

III.1 Exemple 1 : balles

1.1 `BallWorld`

```
import java.awt.*;
import java.awt.event.*;

public class BallWorld extends Frame {

        public static void main (String [ ] args)
        {
                BallWorld world = new BallWorld (Color.red);
                world.show ();
        }
```

```
    private static final int FrameWidth = 600;              // 7
    private static final int FrameHeight = 400;             // 8
    private Ball aBall;                                     // 9
    private int counter = 0;                                // 10

    private BallWorld (Color ballColor) {                   // 11
            // constructeur pour le nouveau monde           // 12
                    // on change la taille du cadre         // 13
            setSize (FrameWidth, FrameHeight);              // 14
            setTitle ("Ball World");                        // 15

            // initialiser les champs de donnees            // 16
            aBall = new Ball (10, 15, 5);                   // 17
            aBall.setColor (ballColor);                     // 18
            aBall.setMotion (3.0, 6.0);                     // 19
    }

    public void paint (Graphics g) {
                    // d'abord, afficher la balle
            aBall.paint (g);
                    // puis la bouger legeremenet
            aBall.move();
                    // et traiter les cas de debordement
            if ((aBall.x() < 0) || (aBall.x() > FrameWidth))
                    aBall.setMotion (-aBall.xMotion(), aBall.yMotion());
            if ((aBall.y() < 0) || (aBall.y() > FrameHeight))
                    aBall.setMotion (aBall.xMotion(), -aBall.yMotion());
                     // enfin, reafficher le cadre
            counter = counter + 1;
            if (counter < 2000) repaint();
            else System.exit(0);
    }
}
```

- Déclarations en lignes 7–10 : constantes (`public static final`) ne peuvent être modifiées
- Utilités d'un constructeur ; **évite les 2 erreurs** suivantes :
  - Un objet est créś, mais **utilisé avant d'être initialisé**. Ex. en C

```
struct enreg {
    int  no;
    char nom[255];
};
main(int argc, char *argv[]) {
```

```
        struct enreg  moi, *ptrtoi;
        printf("Moi: no+10 : %d    nom : %s\n", moi.no+10, moi,nom);
        printf("Toi: no : %d    nom : %s\n", ptrtoi->no, ptrtoi->moi,nom);
    }
```

  – Un objet est créé et **initialisé plusieurs fois** avant d'être utilisé.
- Constructeur en lignes 11–19
- La classe `BallWorld` est une sous classe de `Frame` : un cadre (fenêtre) graphique
- AWT (Abstract Window Toolkit) `show()` et `paint()`

1.2 Classe `Frame`

- Classe `Frame` : *fenêtre principale d'application*, pouvant changer de taille
- Peut avoir un titre, une barre de menu, une icône, un curseur
- **Méthodes** :

`methode()`	But
`setCursor(int cursorType)`	choisit le type de curseur
`setIconImage(Image image)`	choisit l'icône
`setMenuBar(MenuBar mb)`	choisit le menu
`setResizable(boolean resizable)`	peut changer de taille ou pas
`setTitle(String title)`	choisit le titre
`dispose()`	redéfinition de `Window.dispose()`

- Méthodes analogues à `setXXX()` : `getXXX()`
- **Sous classe de** `Window` elle-même **sous classe de** `Component`

1.3 Modèle graphique en Java

- **Java AWT** (Abstract Window Toolkit) : exemple de *cadre logiciel*
- Cadre logiciel : fournit **la structure** d'un programme, mais **pas les détails** de l'application
- Le contrôle général, le flux d'exécution, est fourni par le cadre, qui n'a **pas besoin d'être réécrit**
- Exemple : la méthode `show()` ; la fenêtre est crée et l'image de la fenêtre doit être dessinée
- Pour cela `show()` appelle la méthode `paint()`, passant en argument un **objet graphique**

- Le programmeur définit l'**apparence** de la fenêtre en **implantant la méthode** `paint()`
- L'objet graphique en argument permet de dessiner des lignes, des polygônes, ...et d'afficher du texte
- Cas de la classe `BallWorld`

```
public void paint(Graphics g) {
    // D'abord dessiner la balle
    aBall.paint(g);
    // ensuite la bouger legerement
    aBall.show();
    if ( (aBall.x() < 0) || (aBall.x() > FrameWidth) )
        aBall.setMotion(-aBall.xMotion(), aBall.yMotion());
    if ( (aBall.y() < 0) || (aBall.y() > FrameWidth) )
        aBall.setMotion(-aBall.xMotion(), -aBall.yMotion());
    // finalement, redessiner le cadre (frame)
    counter = counter + 1;
    if (counter < 2000)
        repaint();
    else
        System.exit();
```

  - La méthode `paint()` produit l'image de la balle
  - Elle permet également de réactualiser la position de la balle (on la bouge un peu ; si elle sort du cadre, on inverse la direction)
- La méthode `repaint()` (aussi héritée de `Frame`) indique à AWT que la fenêtre doit être redessinée
- Le programmeur appelle `repaint()` qui, elle, appellera éventuellement `paint()`. Il **n'appelle pas directement** `paint()`

1.4 La classe `Ball`

```
//      abstraction de balle rebondissante generale et reutilisable
//

import java.awt.*;

public class Ball {
        protected Rectangle location;
        protected double dx;
        protected double dy;
        protected Color color;

        public Ball (int x, int y, int r)
```

```
{
        location = new Rectangle(x-r, y-r, 2*r, 2*r);
        dx = 0;
        dy = 0;
        color = Color.blue;
}

// fonctions fixant les attributs
//
public void setColor (Color newColor)
                           { color = newColor;   }

public void setMotion (double ndx, double ndy)
                           { dx = ndx; dy = ndy; }

// fonctions d'access aux attributs de la balle
//   (Methodes de consultation de l'etat de l'objet)
//
public int radius()        {  return location.width / 2; }

public int x()             {  return location.x + radius(); }

public int y()             { return location.y + radius(); }

public double xMotion()    { return dx; }

public double yMotion()    { return dy; }

public Rectangle region()  { return location; }

// fonctions de modification d'attributs de la  balle
//   (Methodes de changement d'etat de l'objet)
//
public void moveTo(int x, int y)
                           { location.setLocation(x, y); }

public void move()  { location.translate ((int) dx, (int) dy); }

public void paint (Graphics g)
{
    g.setColor (color);
    g.fillOval (location.x, location.y,
                location.width, location.height);
```

```
        }
}
```

- Les 4 champs de données sont déclarés `protected` :
  - Le champ `location`, un `Rectangle` dans lequel est *dessinée* la balle ,
  - Les champs `dx, dy`, les *directions* horizontale et verticale de la balle,
  - Le champ `color`, la *couleur* de la balle.

  ⇒ les **fils** éventuels **peuvent y accéder**, **pas les autres**.
- BONNE PRATIQUE : **déclarer** les données `protected`, **plutôt que** `private`
- 6 méthodes permettent d'accéder aux attributs de la balle : `radius()`, `x()`, `y()`, `xMotion()`, `yMotion()`, `region()`. Ce sont des *fonctions d'accès.*
- PRATIQUE FORTEMENT ENCOURAGÉE, plutôt que d'avoir des champs de données publics.
- Permet de **contrôler strictement la modification** des données
- Certaines de ces fonctions utilisent des opérations fournies par la classe `rectangle`
- La méthode `paint()` utilise 2 fonctions de la classe `Graphics` : `setColor()` et `fillOval()`

1.5 Multiples objets de la même classe

On a ici un tableau de balles `ballArray` instancié dans le constructeur de la classe `MultiBallWorld()`.

```
import java.awt.*;
import java.awt.event.*;

public class MultiBallWorld extends Frame {

    public static void main (String [ ] args)
    {
        MultiBallWorld world = new MultiBallWorld (Color.red);
        world.show ();
    }

    private static final int  FrameWidth    = 600;
    private static final int  FrameHeight   = 400;
    private Ball []           ballArray;
    private static final int  BallArraySize = 6;
    private int               counter       = 0;

    private MultiBallWorld (Color ballColor) {
            // on redimensionne le cadre
```

```
        setSize (FrameWidth, FrameHeight);
        setTitle ("Ball World");

            // initialisation des champs de donnees
        ballArray = new Ball [ BallArraySize ];
        for (int i = 0; i < BallArraySize; i++) {
            ballArray[i] = new Ball(10, 15, 5);
            ballArray[i].setColor (ballColor);
            ballArray[i].setMotion (3.0+i, 6.0-i);
            }
    }

    public void paint (Graphics g) {
        for (int i = 0; i < BallArraySize; i++) {
            ballArray[i].paint (g);
                // la deplacer legerement
            ballArray[i].move();
            if ((ballArray[i].x() < 0) || (ballArray[i].x() > FrameWidth))
                ballArray[i].setMotion(-ballArray[i].xMotion(),
                                       ballArray[i].yMotion());
            if ((ballArray[i].y() < 0) || (ballArray[i].y() > FrameHeight))
                ballArray[i].setMotion(ballArray[i].xMotion(),
                                       -ballArray[i].yMotion());
            }
                // enfin, redessiner le cadre
        counter = counter + 1;
        if (counter < 2000) repaint();
        else System.exit(0);
    }
}
```

– Un tableau de balles est **créé en 3 temps** :
  1) `private Ball[] ballArray ; // declaration de ballArray`
  2) `ballArray = new Ball[BallArraySize] ;    //allocation memoire pour les references BallArray[i]`
  3)
  ```
  for(int i = 0; i < BallArraySize; i++) {
      ballArray[i] = new Ball(10, 15, 5);
  } // allocation memoire pour chacune des instances de Ball
  ```

– Chacune des balles s'anime **indépendamment des autres**

III.2 Exemple 2 : Flipper

2.1 La classe PinBallGame, 1^{iere} version

```
import java.awt.*;
import java.awt.event.*;
import java.util.Vector;

public class PinBallGame extends Frame {

    public static void main (String [ ] args) {
        world = new PinBallGame();
        world.show();
    }

    public static final int FrameWidth = 400;
    public static final int FrameHeight = 400;
    public static PinBallGame world;
    private Vector balls;

    public PinBallGame () {
        setTitle ("Pin Ball Construction Kit");
        setSize (FrameWidth, FrameHeight);

        balls = new Vector();
        addMouseListener (new MouseKeeper());
    }

    private class PinBallThread extends Thread { ... }

    private class MouseKeeper extends MouseAdapter { ... }

    public void paint (Graphics g) {
        g.setColor (Color.white);
        g.fillRect (FrameWidth-40, FrameHeight-40, 30, 30);
        g.setColor (Color.red);
        g.fillOval (FrameWidth-40, FrameHeight-40, 30, 30);
            // dessiner les boules
        for (int i = 0; i < balls.size(); i++) {
            Ball aBall = (Ball) balls.elementAt(i);
            aBall.paint(g);
            }
        }
```

```
}
```

- Pulieurs balles, nombre inconnu à l'avance ⇒ utilisation de `Vector` : **tableau à croissance dynamique**
- **Pas besoin de spécifier le type** des éléments d'un vecteur
- Un vecteur contient des `Object` ⇒ les types primitifs (`int`, ...) doivent être convertis en leurs objets associés (`Integer`, ...) ⇒ **Accès** à un élément doit être **suivi d'un cast** : `Ball aBall = (Ball) balls.elementAt(i) ;`

2.2 Écouteurs à souris

- Modèle à événements de Java centré autour des *écouteurs*, objets qui écoutent les événements qui arrivent
- Les écouteurs sont spécifiés *via* une interface. Pour les événements de souris :

```
public interface MouseListener {
     public void mouseClicked(MouseEvent e);
     ...
     public void mouseReleased(MouseEvent e);
     ....
}
```

- *Interface* java : description de **comportement**. Pas d'implantation de méthode. Seuls champs de données permis déclarés `static`
- *Héritage* : un enfant hérite de la **structure** de ses parents **et** de la possibilité d'imiter le comportement des parents.
- Interface : pas de corps de méthode dans une "interface parent" Une classe "enfant" "hérite" seulement de la **spécification des méthodes** et non de la structure (champs de données, corps de méthodes)
- Une interface peut être **utilisée comme type de données**. La variable doit être une instance de classe implantant l'interface
- Une classe implantant une interface doit fournir le corps de **toutes** les méthodes ⇒ *adaptateurs* : classes implantant l'interface avec toutes les **méthodes à corps vide** : `{}`
- Exemple d'adaptateur pour les écouteurs à souris (interface `MouseListener`) : classe `MouseAdapter`
- L'utilisateur définit une classe héritant de `MouseAdapter` avec une **redéfinition des seules méthodes nécessaires**.
- Ici, seule `mousePressed()` est redéfinie, dans la classe `MouseKeeper`.

```
private class MouseKeeper extends MouseAdapter {

    public void mousePressed (MouseEvent e) {
```

```
            int x = e.getX();
            int y = e.getY();
            if ((x > FrameWidth-40) && (y > FrameHeight -40)) {
                PinBall newBall = new PinBall(e.getX(), e.getY());
                balls.addElement (newBall);
                Thread newThread = new PinBallThread (newBall);
                newThread.start();
            }
        }
```

Une instance est crée, puis passée à : `addMouseListener(new MouseKeeper())` ;

– L'argument des méthodes d'un `MouseListener` est un `MouseEvent` : il contient des informations sur l'événement survenu. Ici on s'intéresse aux coordonnées de la souris

2.3 Plusieurs activités (threads)

– Une activité (`thread`) par balle pour son mouvement

```
public class PinBallGame extends Frame {
...

    private class PinBallThread extends Thread {
        private Ball theBall;

        public PinBallThread (Ball aBall) {
            theBall = aBall;
        }

        public void run () {
            while (theBall.y() < FrameHeight) {
                theBall.move ();
                // D'autres actions viendront ici
                repaint();
                // on prend un peu de repos
                try {
                    sleep(10);
                } catch (InterruptedException e) { System.exit(0); }
            }
        }
    }
    ...
}
```

- Notez : on doit toujours appeler `start()` et implanter `run()`. On ne **doit pas appeler directement** `run()`
- Traitement de l'exception d'interruption (touche `^C`)

2.4 Ajout de cibles : héritage et interfaces

- On veut traiter toutes les cibles **de manière uniforme** pour certaines opérations.
- Chaque cible étant toutefois représentée de manière interne par une **structure** de données **différente**

⇒ Pas d'utilisation de l'héritage pour les cibles. Plus précisément :
  - `PinBall` (boule de flipper) est un **type particulier** de `Ball` (boule) **avec des caractéristiques en plus** (peut s'exécuter comme une tâche séparée) ⇒ **héritage**.
  - Peu de choses en commun entre un `Peg` (piquet : cible qui, lorsqu'elle est touchée, compte un nombre de points et renvoie la balle dans une autre direction) et un `Wall` (mur : qui réfléchit le mouvement de la balle) ⇒ un **partage de certains comportements** *via* une **interface**
- L'interface `PinBallTarget` s'écrit

```
interface PinBallTarget {
        public boolean intersects (Ball aBall);
        public void moveTo (int x, int y);
        public void paint (Graphics g);
        public void hitBy (Ball aBall);
}
```

- Plus précisément, chaque cible
  - peut savoir si une boule entre en contact avec elle, *via* `intersects()` : s'il y a une intersection entre elle et la région couverte par la boule ;
  - peut être déplacée en un point de la surface de jeu *via* `moveTo()` ;
  - peut se dessiner *via* `paint()` ;
  - peut réagir lorsqu'elle est touchée par une boule *via* `hitBy()`.

  Chaque cible aura une implantation différente de ces opérations

2.5 Cible `Spring`

- Cas de `Spring` (ressort) :

```
class Spring implements PinBallTarget {
        private Rectangle pad;
        private int state;

        public Spring (int x, int y) {
```

```
                pad = new Rectangle(x, y, 30, 3);
                state = 1;
        }

        public void moveTo (int x, int y)
                { pad.setLocation(x, y); }

        public void paint (Graphics g) {
                int x = pad.x;
                int y = pad.y;
                g.setColor(Color.black);
                if (state == 1) {
                        g.fillRect(x, y, pad.width, pad.height);
                        g.drawLine(x, y+3, x+30, y+5);
                        g.drawLine(x+30, y+5, x, y+7);
                        g.drawLine(x, y+7, x+30, y+9);
                        g.drawLine(x+30, y+9, x, y+11);
                        }
                else {
                        g.fillRect(x, y-8, pad.width, pad.height);
                        g.drawLine(x, y+5, x+30, y-1);
                        g.drawLine(x+30, y-1, x, y+3);
                        g.drawLine(x, y+3, x+30, y+7);
                        g.drawLine(x+30, y+7, x, y+11);
                        state = 1;
                }
        }

        public boolean intersects (Ball aBall)
                { return pad.intersects(aBall.location); }

        public void hitBy (Ball aBall) {
                        // on s'assure que l'on monte
                if (aBall.dy > 0)
                        aBall.dy = - aBall.dy;
                        // on booste un peu la boule
                aBall.dy = aBall.dy - 0.5;
                state = 2;
        }
}
```

– Lorsqu'une boule qui descend entre en contact avec lui, le `Spring` fait rebondir la balle.
– Il est représenté par un rectangle (aplati) et une série de segments en

zigzag

– Il y a intersection ressort-boule si le rectangle aplati du ressort et celui circonscrit à la boule “s’intersectent”
– 2 représentations graphiques du ressort, contrôlé par `state` :
  – état comprimé (`state == 1`) : état de départ
  – état détendu (`state == 2`) : déclenché par le choc d’une boule (dans `hitBy()`)

2.6 Cible `Wall`

– Cas de `Wall` (mur) :

```
class Wall implements PinBallTarget {
        public Rectangle location;

        public Wall (int x, int y, int width, int height)
                { location = new Rectangle(x, y, width, height); }

        public void moveTo (int x, int y)
                { location.setLocation (x, y); }

        public void paint (Graphics g) {
                g.setColor(Color.black);
                g.fillRect(location.x, location.y,
                           location.width, location.height);
        }

        public boolean intersects (Ball aBall)
                { return location.intersects(aBall.location); }

        public void hitBy (Ball aBall) {
                if ( (aBall.y() < location.y) ||
                     (aBall.y() > (location.y + location.height)) )
                        aBall.dy = - aBall.dy;
                else
                        aBall.dx = - aBall.dx;
        }
}
```

  – Représenté par un rectangle (aplati).
  – Une balle entre en contact avec un mur si leurs rectangles s’intersectent.
  – La balle rebondit alors sur le mur dans la direction opposée de l’incidente.
– Une variable de type `PinBallTarget` peut être aussi bien une référence à une instance de `Spring` que de `Wall` : un des aspects du **polymorphisme**

2.7 Cible `Hole`

- 3^{ieme} type de cible : `Hole` (trou) agit comme un puits à boules :
  - il est représenté par un cercle coloré, comme une `Ball` (boule)
  - il a une position sur l'aire de jeu, comme une `Ball` (quoique ayant une position fixe)
- ⇒ un `Hole` est structurellement similaire à une `Ball`. On utilise donc l'héritage
- De plus `Hole` est une cible et implante à ce titre l'interface `PinBallTarget`. Donc, `Hole` :
  - hérite en particulier du comportement de `Ball`, et donc des méthodes `paint()` et `moveTo()` ;
  - doit implanter toutes les méthodes de `PinBallTarget` ⇒ implantation de `intersects()` et `hitBy()` ;
  - voit la méthode `setMotion()` redéfinie pour ne rien faire. `Hole` est une `Ball` statique

2.8 Cible `ScorePad`

- La cible `ScorePad` (zone à bonus) est une région circulaire fixe, comme un trou. Lorsqu'une boule passe dessus, le carré à bonus ajoute un certain montant au score du joueur.
- ⇒ `ScorePad` hérite de `Hole`. Une classe héritant d'un parent qui implante une interface doit implanter elle-même cette interface. `ScorePad` redéfinit `paint()` et `hitBy()`
- On ajoute un label (zone de texte en lecture seulement) à l'interface : score du joueur
- Notez : la méthode `addScore()` est déclarée `synchronized`. Deux boules peuvent très bien passer sur le même `ScorePad` en même temps ⇒ un score peut être indéfini. Si `addScore()` est `synchronized`, 2 threads (activités) ne peuvent l'exécuter en même temps

2.9 Cible `Peg`

Un `Peg` (piquet ou champignon) est similaire à un `ScorePad` :
- aspect circulaire, de petite tour ;
- augmentation de score quand une boule entre en contact avec ;
- la boule est réfléchie quand elle butte dessus ;
- loi de réflexion simpliste (normalement loi de Descartes) et attente que la boule n'ait plus d'intersection avec la cible ;

- 2 représentations graphiques : état de base, comprimé et état venant d'être touché, détendu.

2.10 Autres versions du jeu – palettes

- 2e version du jeu : avec un vecteur de cibles
- 3e version du jeu : le joueur peut composer lui-même l'aire de jeu en plaçant les diverses cibles
- Les cibles sont représentées dans une palette tout au long de la bordure gauche. On clique sur la cible désirée et on la fait glisser où l'on veut (click and drag of the desired target)
- Réalisé par redéfinition de `mousePressed()` et `mouseReleased()` :
  - `mousePressed()` et `mouseReleased()` communiquent *via* la variable `element`.
  - `mousePressed()` créé une cible potentielle.
  - `mouseReleased()` vérifie si les coordonnées de la souris au relâchement sont à l'intérieur de l'aire de jeu ; si oui, ET si une cible était sélectionnée (`element != null`), la cible est ajoutée au vecteur de cibles.

III.3 Formes d'héritage

3.1 Héritages

- En Java, l'héritage *stricto sensu* (mot clé `extends`) est la sous classification.
- Toutefois, beaucoup de **points communs** entre l'**héritage de spécification** (mot clé `implements`) et la sous classification, ou **héritage de code**.
- *Héritage* :

  le comportement et les données des classes enfants sont une **extension** des propriétés des classes parents (mot clé `extends`).

  Mais également :

  une classe enfant est une version plus spécialisée (ou restreinte) d'une classe parent ; c'est donc également une **contraction** d'une classe parent.
- Exemple : un `Frame` est tout type de fenêtre, mais un `PinBallGame` est un type bien de fenêtre particulier.
- Les Extension/Contraction donnent plusieurs usages de l'héritage.

3.2 Classe `Object`

- Classe `Object` : **dont toutes les classes Java dérivent**.
- **Fonctionnalités minimales** dans `Object` incluent :

methode()	But
`equals(Object obj)`	teste l'égalité de l'objet argument et de l'appelant. Souvent redéfinie.
`getClass()`	Renvoie une chaîne contenant le nom de la classe.
`hashCode()`	Renvoie la valeur de hachage pour cet objet. Doit être définie lorsque `equals()` l'est.
`toString()`	Renvoie une chaîne associée à l'objet pour affichage. Souvent redéfinie.

3.3 Substituabilité

- *Substituabilité* : le **type** donné lors de la **déclaration** d'une variable peut **différer** de celui de **l'instance référencée** par la variable.
- Exemple : dans le jeu de flipper, la variable `target` est déclarée comme `PinBallTarget`, mais correspond à beuacoup de cibles différentes (du vecteur `targets`) `PinBallTarget target = (PinBallTarget) target.elementAt(j);`
- Une variable déclarée de type `Object` peut contenir (référencer une instance de) tout type non primitif. Un vecteur contient des éléments de type `Object` ; on peut donc tout y stocker.
- Une justification possible de la *substituabilité* par héritage est le

principe de substitution	
(i)	les instances de la sous classe doivent **posséder tous les champs de données** associés à la classe parent ;
(ii)	les instances de la sous classe doivent **implanter**, au moins par héritage, **toutes les fonctionnalités** définies pour la classe parent ;
(iii)	donc une instance d'une classe enfant peut imiter le comportement d'une classe parent et être **indistinguable** d'une instance de la classe parent lorsque **substituée** dans une situation similaire.

- Ceci **ne couvre pas toutes les utilisations** de l'héritage.

3.4 Sous type – Sous classe

- *Sous type* : relation entre 2 types reconnaissant explicitement le **principe de substitution**.
- C'est-à-dire, un type `B` est un sous type de `A` si :

  - une instance de `B` peut être légalement assignée à une variable déclarée de type `A` ;
  - cette valeur peut alors être utilisée par la variable sans changement visible de comportement.
- *Sous classe* : mécanisme de construction d'une **nouvelle classe** utilisant **l'héritage**. Se reconnaît facilement à la présence du mot clé `extends`.
- La relation de sous type est plus abstraite et se trouve peu documentée directement dans le source.
- Dans la plupart des cas (mais pas toujours), une sous classe est égalament un sous type.
- Des **sous types** peuvent être formés en **utilisant des interfaces**, reliant des types n'ayant **pas de lien d'héritage**.

3.5 Héritage par spécialisation

- Nouvelle classe : variété spécialisée de la classe parent, mais satisfaisant les spécifications du parent pour tous les aspects importants.
- Cette forme **crée toujours un sous type** et le **principe de substitution** est **parfaitement respecté**.
- **Forme d'héritage idéale**, à rechercher.
- Exemple du jeu de flipper : `public class PinBallGame extends Frame ...`
  - pour exécuter l'application, des méthodes comme `setSize()`, `setTitle()` et `show()`, héritées de `Frame`, sont appelées.
  - Ces méthodes ne réalisent pas qu'elles manipulent une instance de la classe `PinBallGame`.
  - Elles agissent comme si elles opéraient sur une instance de `Frame`.
  - Lorsque nécessaire, une méthode redéfinie pour l'application (par ex. `paint()`) est appelée.

3.6 Héritage par spécification

- But : garantir que les classes aient une **interface commune**, c'est-à-dire implantent les mêmes méthodes.
- La classe parent peut être une **combinaison d'opérations implantées** et **d'opérations déléguées** aux classes enfants.
- Forme d'héritage en fait cas particulier de l'héritage par spécialisation, sauf que les sous classes ne sont pas des raffinements d'un type existant mais des réalisations de spécifications abstraites.
- Classe parent parfois nommée *classe de spécification abstraite.*

- **Forme d'héritage idéale**, à rechercher, **respectant le principe de substitution**.
- **Deux mécanismes**, en Java, pour l'héritage par spécification :
  - les **interfaces** (par ex. des cibles du jeu de flipper)
  - les **classes abstraites**
- Exemple de classe abstraite : `Number` de la bibliothèque Java (classe parent de `Integer`, `Long`, `Double`, ...). Description de classe :

```
public abstract class Number {
    public abstract int intValue();
    public abstract int longValue();
    public abstract int floatValue();
    public abstract int doubleValue();
    public abstract int byteValue() {
        return (byte) intValue();
    }
    public abstract int shortValue() {
        return (short) intValue();
    }
```

- Les méthodes déclarées `abstract` doivent être redéfinies dans une sous-classe instanciable.
- Héritage par spécification : la classe parent n'implante pas le comportement mais définit celui qui doit être implanté dans les classes enfant.

3.7 Héritage par construction

- Une classe peut **hériter des fonctionnalités désirées** d'un parent, juste en changeant le nom des méthodes, ou en modifiant des arguemnts. Peut être vrai **même s'il n'y a pas de relation conceptuelle** entre les classes parent et enfant.
- Par exemple, dans le jeu de flipper, `Hole` est déclaré comme sous classe de `Ball`. Pas de relation logique entre les 2 concepts.
- Mais d'un **Point de vue pratique**, beaucoup du comportement de `Hole` correspond à celui de `Ball`. ⇒ Gain de temps de développement en utilisant l'héritage.

```
class Hole extends Ball implements PinBallTarget {

        public Hole (int x, int y) {
                super (x, y, 12);
                setColor (Color.black);
        }
```

```
        public boolean intersects (Ball aBall)
                { return location.intersects(aBall.location); }

        public void hitBy (Ball aBall) {
                // mettre la boule hors champ
                aBall.moveTo (0, PinBallGame.FrameHeight + 30);
                // arreter le mouvement
                aBall.setMotion(0, 0);
        }
}
```

– Autre exemple, de la bibliothèque Java : `Stack`, construite par héritage à partir de `Vector`.

```
class Stack extends Vector {

    public Object push(Object item) {
        addElement(item);
        return(item);
   }

   public boolean empty() {
       return isEmpty();
   }

   public synchronized Object pop() {
       Object obj = peek();
       removeElementAt(size() - 1);
       return obj;
   }

   public synchronized Object peek() {
       return elementAt(size() - 1);
   }
```

– Forme d'héritage parfois boudée, parce que **violant directement le principe de substitution**.
– Forme toutefois pratique et assez largement utilisée.

3.8 Héritage par extension

– Une classe enfant **ajoute de nouveaux comportements et n'altère pas ceux hérités**.
– Exemple de la bibliothèque Java : `Properties` héritant de `Hashtable`.

- `HashTable` : structure de dictionnaire avec un ensemble de paires clés/valeurs.
- `Properties` : utilisé notamment pour les informations relatives à l'environnement d'exécution. Par ex. le nom de l'utilisateur, al version de l'interpréteur, le nom du système d'exploitation.
- `Properties` utilise les méthodes de `HashTable` pour stocker et retirer des paires nom/valeur.
- Elle définit également d'autres méthodes :
  - `load()`, pour lire du disque,
  - `save()`, pour stocker sur disque,
  - `getProperty()`,
  - `propertyNames()`,
  - `list()`.
- Les fonctionnalités du parent restant intactes, cette forme d'héritage **respecte le principe de substitution** et les sous classes sont des sous types.

3.9 Héritage par limitation

- Le comportement de la sous-classe est plus restreint que celui du parent.
- Forme caractérisée par la présence de **méthodes qui rendent une opération permise pour le parent illégale**.
- Cette forme d'héritage **contredit explicitement le principe de substitution**. ⇒ l'éviter autant que possible.

3.10 Héritage par combinaison

- Une classe **héritant de 2 parents ou plus**, c.à.d. de l'*héritage multiple*.
- Une sous classe Java ne peut être formée par héritage à partir de plus d'une classe parent. Mais plusieurs formes très analogues existent.
- D'abord, il est commun pour une nouvelle classe d'**étendre une classe existante et d'implanter une interface**. Par ex., dans le jeu de flipper : `class Hole extends Ball implements PinBallTarget ...`
- Ensuite, une **classe peut implanter plus d'une interface**. Par exemple, dans la bibliothèque d'E/S Java, un `RandomAcessFile` implante à la fois les protocoles `DataInput` et `DataOutput`.

3.11 Résumé des formes d'héritage

Dans ce qui suit, le sigle **(PS)** désigne une forme d'héritage qui respecte le principe de sunstitution.

- *Spécialisation*^(PS) : la classe enfant est un cas particulier de la classe parent, c.à.d. un **sous type**.
- *Spécification*^(PS) : la classe parent définit un **comportement** implanté **dans la classe enfant** et **non dans la classe parent**.
- *Construction* : la classe enfant **utilise le comportement** fourni par le **parent** mais n'en est **pas un sous type**.
- *Extension*^(PS) : la classe enfant **ajoute de nouvelles fonctionnalités** à la classe parent, mais ne modifie aucun comportement hérité.
- *Limitation* : la classe enfant **restreint l'utilisation de certains comportements** hérités de la classe parent.
- *Combinaison*^(PS) : la classe enfant **hérite de caractéristiques de plus d'une classe parent**. Bien que l'héritage multiple ne soit pas intégré à Java, des formes analogues fournissent les mêmes bénéfices ; les classes peuvent utiliser l'héritage et l'implantation d'interface et une classe peut implanter plusieurs interfaces.
- ☛ : le langage Java suppose implicitement que les sous classes sont également des sous types. ⇒ une instance de sous classe peut être assignée à une variable déclarée du type parent. Cependant, cette supposition que les **sous classes sont des sous types n'est pas toujours valide**, plus précisément dans les cas d'héritage par construction et par limitation. Ceci peut être une **source directe d'erreurs**.

3.12 Modificateurs

- Plusieurs modificateurs peuvent altérer divers aspects du processus d'héritage.
- Modificateurs de *visibilité* (ou de contrôle d'accès) : `public`, `protected`, `private`
- Modificateurs d'*accessibilité* :
  - `static` : on ne peut redéfinir une méthode `static`
  - `abstract` : classes non instanciables ; ne peut être utilisé que comme type parent.
  - `final` : l'inverse du précédent. On ne peut créer de sous classe d'une classe `final`.

3.13 Bénéfices de l'héritage

- Un programmeur peut soit agir comme *développeur*, soit comme *utilisateur* de composants réutilisables.

- La construction de logiciels devient alors une **entreprise à grande échelle**, où chacun créé certains composants et en utilise d'autres, disponibles sur le marché.
- Les mécanismes de **production** et d'**utilisation** de ces composants ont été codifiés : les composants ainsi créés sont les *beans* Java (ou grains, c.à.d. de la matière première pour la production de café, déja sélectionnée et traitée).
- **Réutilisabilité logicielle** Un code **hérité** n'a **pas besoin d'être réécrit**.
- **Robustesse renforcée** Du code **fréquemment utilisé** contient **moins d'erreurs** que du code peu utilisé. Les mêmes composants réutilisés dans plusieurs applications verront leurs erreurs plsu rapidement découvertes que ceux utilisés une fois. Le coût de maintenance logicielle peut également être partagé entre plusieurs projets.
- **Partage de code** Intervient à plusieurs niveaux :
  - Plusieurs utilisateurs ou projets partagent les mêmes classes.
  - Deux ou plusieurs classes développées par un membre d'un projet héritent d'une classe parente.
- **Cohérence de l'interface** Lorsque 2 classes ou plus héritent de la même super classe, on est sûr que le comportement hérité sera identique dans tous les cas.
- **Composants logiciels** L'héritage permet la onstruction de composants réutilisables. ceci réduit d'autant le codage.
- **Prototypage rapide** En utilisant des composants réutilisables, une première version peut être rapidement élaborée puis raffinée par itérations successives avec les utilisateurs du produit.
- **Polymorphisme et cadres logiciels** Les logiciels sont souvent écrits de manière ascendante (ou bottom up) et sont conçus de manière descendante (top down). La portabilité décroît lorsqu'on gravit les niveaux d'abstraction. Le **polymorphisme permet** de générer des **composants réutilisables à un haut niveau d'abstraction** ; il suffit pour les réutiliser de changer quelques composants de plus bas niveau. La bibliothèque AWT est un exemple de cadre logiciel qui s'appuie sur l'héritage et la substitution.
- **Protection de l'information** L'utilisateur d'un composant n'a besoin de comprendre **que la nature d'un composant et son interface**. $\Rightarrow$ le **couplage** entre systèmes logiciels est **réduit**. Le couplage est l'une des principales causes de complexité des logiciels.

3.14 Contreparties : coût de l'héritage

- **Vitesse d'exécution** La vitesse d'exécution d'un assemblage de composants réutilisables est souvent moindre que celle de code spécialisé. Mais :
  - La différence en vitesse d'exécution est souvent peu élevée.

  - Elle est compensée par une plus **grande vitesse de développement logiciel** et un **coût de maintenance moindre**.
  - Il ne faut surtout pas être obnubilé par la vitesse d'exécution, mais **développer d'abord sainement**. Puis déterminer les goulots d'étranglement et optimiser les seules parties qui en ont besoin. Rappel de la remarque de Bill Wulf[1] "Plus de pêchés informatiques sont commis au nom de l'efficacité (sans nécessairement l'atteindre) que pour toute autre raison – y compris le crétinisme le plus viscéral."
- **Taille des programmes** L'utilisation des bibliothèques tend à faire croître la taille du code final. Point peu important avec la chute de prix des mémoires.
- **Surcharge du passage de message** En général peu important. Contrebalancé par les bénéfices de l'héritage.
- **Complexité des programmes** Il se peut qu'un **graphe d'héritage complexe** nécessite plusieurs aller-retours du haut en bas de ce graphe pour bien le comprendre. C'est le *problème du yo–yo*

III.4 Exemple 3 : jeu de solitaire

4.1 Classe Card

```
import java.awt.*;

public class Card {
        // Constantes publiques pour les cartes et couleurs
        final public static int width   = 50;
        final public static int height  = 70;
        final public static int heart   = 0;
        final public static int spade   = 1;
        final public static int diamond = 2;
        final public static int club    = 3;
        // Champs de donnees privees pour le rang et la couleur
        private boolean         faceup;
        private int             r;
        private int             s;

        Card (int sv, int rv)
                { s = sv; r = rv; faceup = false; }

                // access attributes of card
```

[1] W.A. Wulf "A case Against the GOTO", Proc. of the 25th National ACM Conf., 1972.

```
        // Acces aux attributs de la carte
        //   (methodes de visualisation d'etat)
        //
        public int rank () { return r; }

        public int suit() { return s; }

        public boolean faceUp() { return faceup; }

        public void flip() { faceup = ! faceup; }

        public Color color() {
                if (faceUp())
                        if (suit() == heart || suit() == diamond)
                                return Color.red;
                        else
                                return Color.black;
                return Color.yellow;
                }

        public void draw (Graphics g, int x, int y) { ... }
}
```

- Pour chaque carte :
  - une valeur de couleur (pique, coeur, carreau, trèfle),
  - une valeur de rang (2-10, V, D, R, A).
- Variables d'instance déclarées `private` et **accessibles *via* des "fonctions d'accès"**.
- Ce type d'accès assure entre autres que la rang et la couleur **ne peuvent être modifiés arbitrairement**.
- Les hauteur, largeur et couleurs sont déclarées comme `static final` (c.à.d. des constantes).
- La plupart des méthodes sont déclarées `final` :
  - sert comme documentation au lectuer indiquant que ces méthodes ne peuvent être redéfinies ;
  - elles peuvent éventuellement être optimisées.
- Une carte peut :
  - initialiser son état (*via* le constructeur),
  - renvoyer son état (`rank()`, `suit()`, `faceUp()`, `color()`),
  - se retourner (`flip()`),
  - se dessiner (`draw()`).

```
public class Card {
```

```
...
        public void draw (Graphics g, int x, int y) {
                String names[] = {"A", "2", "3", "4", "5", "6",
                        "7", "8", "9", "10", "J", "Q", "K"};
                // effacer le rectangle et dessiner le contour
                g.clearRect(x, y, width, height);
                g.setColor(Color.blue);
                g.drawRect(x, y, width, height);
                // dessiner le corps de la carte
                g.setColor(color());
                if (faceUp()) {
                        g.drawString(names[rank()], x+3, y+15);
                        if (suit() == heart) {
                                g.drawLine(x+25, y+30, x+35, y+20);
                                g.drawLine(x+35, y+20, x+45, y+30);
                                g.drawLine(x+45, y+30, x+25, y+60);
                                g.drawLine(x+25, y+60, x+5, y+30);
                                g.drawLine(x+5, y+30, x+15, y+20);
                                g.drawLine(x+15, y+20, x+25, y+30);
                                }
                        else if (suit() == spade) {
                                g.drawLine(x+25, y+20, x+40, y+50);
                                g.drawLine(x+40, y+50, x+10, y+50);
                                g.drawLine(x+10, y+50, x+25, y+20);
                                g.drawLine(x+23, y+45, x+20, y+60);
                                g.drawLine(x+20, y+60, x+30, y+60);
                                g.drawLine(x+30, y+60, x+27, y+45);
                                }
                        else if (suit() == diamond) {
                                g.drawLine(x+25, y+20, x+40, y+40);
                                g.drawLine(x+40, y+40, x+25, y+60);
                                g.drawLine(x+25, y+60, x+10, y+40);
                                g.drawLine(x+10, y+40, x+25, y+20);
                                }
                        else if (suit() == club) {
                                g.drawOval(x+20, y+25, 10, 10);
                                g.drawOval(x+25, y+35, 10, 10);
                                g.drawOval(x+15, y+35, 10, 10);
                                g.drawLine(x+23, y+45, x+20, y+55);
                                g.drawLine(x+20, y+55, x+30, y+55);
                                g.drawLine(x+30, y+55, x+27, y+45);
                                }
                        }
```

```
                else { // face cachee
                        g.drawLine(x+15, y+5, x+15, y+65);
                        g.drawLine(x+35, y+5, x+35, y+65);
                        g.drawLine(x+5, y+20, x+45, y+20);
                        g.drawLine(x+5, y+35, x+45, y+35);
                        g.drawLine(x+5, y+50, x+45, y+50);
                        }
                }
}
```

- Une caret connît sa valeur et sait se dessiner. De cette manière, l'information est **encapsulée et isolée** de l'application utilisant la carte à jouer.
- ⇒ Pour porter l'application sur une plateforme avec des routines graphiques spécifiques, seule la méthode `draw()` doit être réécrite.

4.2 Règles du jeu

- Version *Klondie* du solitaire[2]
- On utilise un paquet de 52 cartes.
- Le *tableau* (table de jeu) consiste en 28 cartes réparties en 7 piles. La 1 $^{\text{ière}}$ pile a 1 carte, la 2 $^{\text{ième}}$ en a 2, etc. jusqu'à 7.
- La carte du haut de chaque pile est initialement face visible (c.à.d. couleur visible). Les autres sont face cachée.
- Les piles de couleur (ou fondements) se construisent des as aux rois selon la couleur. Elles sont remplies avec les cartes disponibles.
- **But du jeu** : placer les 52 cartes dans les piles de couleur.
- Les cartes ne faisant pas partie du tableau sont intialement dans la *pioche*.
- Elles y sont face cachée et sont prises une par une, puis placées, face visible, sur la pile de défausse (ou de mise à l'écart).
- De là, elles peuvent être placées sur une pile tableau ou sur une pile de couleur.
- On tire des cartes jusqu'à ce que la pioche soit vide. À ce moment là, le jeu est terminé si aucun autre mouvement n'est possible.
- Les cartes sont placées sur une pile tableau **seulement sur une carte d'un rang au dessus et d'une couleur opposée**.
- Les cartes sont placées sur une pile de couleur **seulement si elles sont de même couleur et d'un rang au dessus ou si la pile est vide et que la carte est un as**.
- Les places vides dans le tableau qui surviennent pendant le jeu ne peuvent être remplies que par des rois.

[2]A.H. Morehead et G. Mott-Smith, "The Complete Book of Solitaire and Patience Games", Grosset & Dunlap, New York, 1949.

- La carte du dessus de chaque pile tableau et la carte du dessus de la pile défausse sont toujours disponibles.
- La seule fois où plus d'une carte peut être déplacée est lorsqu'un ensemble de cartes faces visibles d'un tableau, nommé une *construction*, est déplacé vers une autre pile tableau.
- Ceci n'est possible que si la carte la plus en dessous de la construction peut être jouée légalement sur la carte la plus au dessus de la destination.
- Transfert de constructions non supproté ici.
- La carte la plus au dessus d'un tableau est toujours face visible.
- Si une carte d'un tableau déplacée laisse sur le dessus de la pile une carte face cachée, cette dernière peut être retournée.

4.3 Héritage des piles de cartes

- Variables d'instance `protected` :
  - coordonnées en haut à gauche de la pile,
  - une `Stack`, l'ensemble des cartes.

```
import java.awt.*;
import java.util.Stack;
import java.util.EmptyStackException;

public class CardPile {
                // coordonnees de la pile de cartes
        protected int x;
        protected int y;
        protected Stack thePile;

        CardPile (int xl, int yl)
                { x = xl; y = yl; thePile = new Stack(); }

        // L'acces aux cartes n'est pas redefini
        //
        public final Card top() { return (Card) thePile.peek(); }

        public final boolean isEmpty() { return thePile.empty(); }

        public final Card pop() {
                try {
                        return (Card) thePile.pop();
                        } catch (EmptyStackException e) { return null; }
                }
```

```
        // Les suivantes sont certaines fois redefinies
        //
        public boolean includes (int tx, int ty) {
                return x <= tx && tx <= x + Card.width &&
                        y <= ty && ty <= y + Card.height;
                }

        public void select (int tx, int ty) {
                // ne rien faire
                }

        public void addCard (Card aCard)
                { thePile.push(aCard); }

        public void display (Graphics g) {
                g.setColor(Color.blue);
                if (isEmpty())
                        g.drawRect(x, y, Card.width, Card.height);
                else
                        top().draw(g, x, y);
                }

        public boolean canTake (Card aCard) {
                return false;
                }

}
```

- 3 méthodes déclarées `final`, communes à toutes les piles de cartes :
  - `top()`, permet de voir quelle est la carte du dessus d'une pile.
  - `isEmpty()`, teste si une pile est vide.
  - `pop()`, retire la carte du dessus d'une pile.
- Traitement de `Stack` vide comme une exception dans `pop()`.
- 5 méthodes peuvent être redéfinies :

  `include()` détermine si le point de coordonnées fournies en arguments est à l'intérieur de la frontière (bord) de la pile. Est redéfinie pour les piles tableau afin de tester toutes les cartes.

  `canTake()` Dit si une pile peut prendre une carte donnée. Seules les piles tableau et couleur peuvent prendre des cartes ; aussi l'action par défaut est-elle non (`false`). Redéfinie pour les 2 piles mentionnées.

  `addCard()` Ajoute une carte à la pile. Redéfini dans la pile de défausse pour être sûr que la carte soit face visible.

  `display()` Affiche la pile de cartes. La méthode par défaut affiche la carte

du dessus de la pile. Redéfinie dans la classe `Tableau` pour afficher une colonne de cartes.

`select()` Effectue une action en réponse à un clic de souris. Est appelée lorsque l'utilisateur clique en un point à l'intérieur de la pile. L'action par défaut est de ne rien faire. Redéfinie pour pour les piles tableau, de pioche et de défausse afin de jouer les cartes du dessus, si possible.

Bénéfice de l'héritage : **au lieu de 25 méthodes à implanter, 13 le sont effectivement** :

	CardPile	SuitPile	DeckPile	DiscardPile	TablePile
includes()	X				X
canTake()	X	X			X
addCard()	X			X	
display()	X				X
select()	X		X	X	X

4.4 Les piles couleur

```
class SuitPile extends CardPile {

        SuitPile (int x, int y) { super(x, y); }

        public boolean canTake (Card aCard) {
                if (isEmpty())
                        return aCard.rank() == 0;
                Card topCard = top();
                return (aCard.suit() == topCard.suit()) &&
                        (aCard.rank() == 1 + topCard.rank());
                }
}
```

- Dans le constructeur `super()` est appelé. Le constructeur est redéfini, comme dans toutes les autres piles, sinon `super()` est automatiquement appelée sans arguments.
- `canTake()` est redéfinie. Une carte peut être placée sur le dessus de la pile si soit :
  - c'est un as (elle a un rang `== 0`),
  - la carte est de la même couleur que la carte du dessus de la pile et d'un rang supérieur (un 3 de trèfle peut être joué sur un 2 de trèfle).

4.5 La pioche

```
class DeckPile extends CardPile {

        DeckPile (int x, int y) {
                // Initialiser le parent
                super(x, y);
                // Puis creer la pioche
                // que l'on place d'abord dans une pile locale
                for (int i = 0; i < 4; i++)
                        for (int j = 0; j <= 12; j++)
                                addCard(new Card(i, j));

                // Ensuite, on bat les cartes
                Random generator = new Random();
                for (int i = 0; i < 52; i++) {
                        int j = Math.abs(generator.nextInt() % 52);
                        // Echanger 2 valeurs de cartes
                        Object temp = thePile.elementAt(i);
                        thePile.setElementAt(thePile.elementAt(j), i);
                        thePile.setElementAt(temp, j);
                        }
                }

        public void select(int tx, int ty) {
                if (isEmpty())
                        return;
                Solitare.discardPile.addCard(pop());
                }
}
```

- Lorsque construite, la pioche crée le jeu complet des 52 cartes. Une fois créé, le jeu est mélangé. Un générateur de nombres pseudo-aléatoires est d'abord créé, puis chaque carte est échangée avec une, prise au hasard.
- On réalise un **accès alátoire dans une pile**. Logiquement, on veut restreindre l'accès au seul élément du dessus. En Java, une `Stack` est construite par héritage à partir d'un `Vector`. Donc la méthode `elementAt()` peut être appliquée à une `Stack`. C'est un exemple **d'héritage par construction**.
- `select()` ne fait rien si la pioche est vide. Sinon, la carte du dessus est enlevée et ajoutée à la pile de défausse.
- Une variable `static` par pile est utilisée. Ici, on se sert de `Solitaire.discardPile`.

4.6 La pile de défausse

```
class DiscardPile extends CardPile {

        DiscardPile (int x, int y) { super (x, y); }

        public void addCard (Card aCard) {
                if (! aCard.faceUp())
                        aCard.flip();
                super.addCard(aCard);
                }

        public void select (int tx, int ty) {
                if (isEmpty())
                        return;
                Card topCard = pop();
                for (int i = 0; i < 4; i++)
                        if (Solitare.suitPile[i].canTake(topCard)) {
                                Solitare.suitPile[i].addCard(topCard);
                                return;
                                }
                for (int i = 0; i < 7; i++)
                        if (Solitare.tableau[i].canTake(topCard)) {
                                Solitare.tableau[i].addCard(topCard);
                                return;
                                }
                // personne ne peut l'utiliser, la remettre dans la liste
                addCard(topCard);
                }
}
```

- **Deux formes d'héritage différentes** sont utilisées. Dans la 1 ière, la méthode `select()` redéfinit ou **remplace** le comportement par défaut. Lorsque l'on clique sur la souris au dessus de la pile, `select()` regarde si la carte du dessus peut être jouée sur une pile couleur ou sur une pile tableau. Sinon la carte est gardée dans la pile de défausse.
- La 2 ième forme d'héritage est un **raffinement** du comportement parental. Dans `addCard()`, le comportement du parent est exécuté (*via* `super()`) et on s'assur que la carte est toujours face visible.
- Un autre type de raffinement apparaît dans les constructeurs.

4.7 Les piles tableau

```
class TablePile extends CardPile {

        TablePile (int x, int y, int c) {
                // initialisation de la classe parent
                super(x, y);
                // initialisation de la pile de cartes
                for (int i = 0; i < c; i++) {
                        addCard(Solitare.deckPile.pop());
                        }
                // retourner la carte du dessus face visible
                top().flip();
                }

        public boolean canTake (Card aCard) {
                if (isEmpty())
                        return aCard.rank() == 12;
                Card topCard = top();
                return (aCard.color() != topCard.color()) &&
                        (aCard.rank() == topCard.rank() - 1);
                }

        public boolean includes (int tx, int ty) {
                // ne pas tester la carte du dessous
                return x <= tx && tx <= x + Card.width &&
                        y <= ty;
                }

        public void select (int tx, int ty) {
                if (isEmpty())
                        return;

                // si elle est face cachee, la retourner
                Card topCard = top();
                if (! topCard.faceUp()) {
                        topCard.flip();
                        return;
                        }

                // sinon, voir si une pile de couleur peut la prendre
                topCard = pop();
                for (int i = 0; i < 4; i++)
                        if (Solitare.suitPile[i].canTake(topCard)) {
```

```
                                Solitare.suitPile[i].addCard(topCard);
                                return;
                                }

                // sinon, voir si une autre pile peut la prendre
                for (int i = 0; i < 7; i++)
                        if (Solitare.tableau[i].canTake(topCard)) {
                                Solitare.tableau[i].addCard(topCard);
                                return;
                                }
                // sinon, la remettre dans notre pile
                addCard(topCard);
                }

        public void display (Graphics g) {
                int localy = y;
                for (Enumeration e = thePile.elements(); e.hasMoreElements(); ) {
                        Card aCard = (Card) e.nextElement();
                        aCard.draw (g, x, localy);
                        localy += 35;
                        }
                }
}
```

- Lorsqu'initialisée, la pile tableau prend un ceratin nombre de cartes de la pioche et les insère dans sa pile. Le nombre de ces cartes est donné en argument au constructeur.
- Une carte peut être ajoutée à la pile (`canTake()`) si la pile est vide et la carte est un roi ou si la carte est de couleur opposée (couleur au sens rouge/noir) à celle de la carte du dessus et d'un rang inférieur.
- Lors d'un test de position d'un clic de souris (`includes()`), seuls les bords gauche, droit et supérieur sont testés, la pile étant de longueur variable.
- Lorsque la pile est sélectionnée, la carte du dessus est retournée si elle est face cachée. Sinon, on essaie de la déplacer vers une pile couleur, puis vers une pile tableau. Sinon, an la laisse en place.
- Pour afficher la pile, on dessine chaque carte ; on utilise une `Enumeration` pour cela, disponible dans les classes dites de collection (`Vector`, `Stack`, `BitSet`, `Hashtable`, `Properties`).

4.8 Classe `Solitaire`

```
public class Solitare {
        static public DeckPile deckPile;
```

```
static public DiscardPile discardPile;
static public TablePile tableau [ ];
static public SuitPile suitPile [ ];
static public CardPile allPiles [ ];
private Frame window;

static public void main (String [ ] args) {
        Solitare world = new Solitare();
        }

public Solitare () {
        window = new SolitareFrame();
        init();
        window.show();
        }

public void init () {
        // D'abord allouer les tableaux
        allPiles = new CardPile[13];
        suitPile = new SuitPile[4];
        tableau = new TablePile[7];
        // ensuite les remplir
        allPiles[0] = deckPile = new DeckPile(335, 30);
        allPiles[1] = discardPile = new DiscardPile(268, 30);
        for (int i = 0; i < 4; i++)
                allPiles[2+i] = suitPile[i] =
                        new SuitPile(15 + (Card.width+10) * i, 30);
        for (int i = 0; i < 7; i++)
                allPiles[6+i] = tableau[i] =
                        new TablePile(15 + (Card.width+5) * i,
                                            Card.height + 35, i+1);
        }

private class SolitareFrame extends Frame {

        private class RestartButtonListener implements ActionListener {
                public void actionPerformed (ActionEvent e) {
                        init();
                        window.repaint();
                        }
                }

        private class MouseKeeper extends MouseAdapter {
```

```
                    public void mousePressed (MouseEvent e) {
                            int x = e.getX();
                            int y = e.getY();
                            for (int i = 0; i < 13; i++)
                                    if (allPiles[i].includes(x, y)) {
                                            allPiles[i].select(x, y);
                                            repaint();
                                            }
                            }
                    }


            public SolitareFrame() {
                    setSize(600, 500);
                    setTitle("Solitaire Game");
                    addMouseListener (new MouseKeeper());
                    Button restartButton = new Button("New Game");
                    restartButton.addActionListener(new RestartButtonListener());
                    add("South", restartButton);
                    }

            public void paint(Graphics g) {
                    for (int i = 0; i < 13; i++)
                            allPiles[i].display(g);
                    }
            }
}
```

- Constructeur :
  - création d'une fenêtre, `SolitaireFrame`,
  - appel de `init()`,
  - affichage de la fenêtre
- `init()` réalise l'initialisation des piles déclarées `static`.
- Notez : 3 opérations distinctes pour un tableau : déclaration, allocation mémoire et assignation.
- Exemple de `suitPile[]` :
  - **Déclaration** : `static public SuitPile suitPile[] ;`
  - **Allocation** : `suitPile = new SuitPile[4] ;` on réserve de la mémoire pour les références `suitPile[0]`, ..., `suitPile[3]` (c.à.d. n octets pour les 4 pointeurs, en interne). Noter que le constructeur `SuitPile()` n'est pas encore appelé.
  - **Assignation** :

```
        for(int i = 0; i < 4; i++)
            suitPile[i] = new SuitPile(15 + (Card.width+10) * i, 30);
```

Ici le constructeur est appelé et les références `suitPile[0]`, . . . , `suitPile[i]` sont assignées.

- Classe interne `SolitaireFrame` sert à gérer la fenêtre de l'application.
- Des écouteurs (listeners) sont créés pour un bouton et l'événement de clic de souris.
- Pressant le bouton, on réinitialise le jeu et on redessine la fenêtre.
- Cliquant sur la souris, les piles sont examinées et la pile appropriée est dessinée.
- On utilise ici le tableau `allPiles[]` ainsi que **le principe de substitution** : `allPiles[]` est déclaré comme un tableau de `CardPile` mais contient en fait plusieurs variétés de piles de cartes.
- Ce tableau est utilisé pour les **comportements génériques** d'une pile de cartes. Par ex. dans `paint()`, chaque `allPiles[i].display()` pourra appeler une méthode différente. C'est de la **répartition de méthode dynamique**, un aspect du **polymorphisme**. Cela correspond aux méthodes "virtual" du C ++.

IV – Boîtes à outils awt et Swing

Références bibliographiques

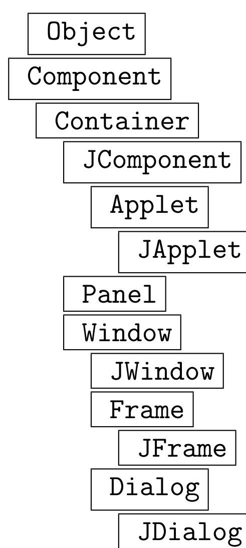
Understanding Object Oriented Programming with Java, T. Budd [Bud98].

IV.1 Aperçu des classes

1.1 Paquetages

- `java.awt` : bases communes et composants de "1ère génération"
- `java.awt.event` : gestion des événements
- `javax.swing` : alternative aux composants AWT
- Les composants Swing ont un nom débutant par `J`. Par exemple, `Frame` est `awt`, `JFrame` est Swing.

1.2 Hiérarchie des classes Swing



1.3 Classes `Component`, `Container`, `Window`

- Une classe de base est la classe `Frame`. Elle hérite en fait d' `Object`, de `Component`, `Container` et `Window`.
- `Component` : élément pouvant s'afficher sur un écran et avec lequel l'utilisateur interagit.
- Parmi les attributs : la taille, la position, des couleurs de fond et de premier plan, le fait d'être visible ou non et un ensemble d'écouteurs à événements.
- Méthodes de `Component` :

methode()	But
`setSize(int, int), getSize()`	Fixer et obtenir la taille du composant.
`enable(), disable()`	Rendre un composant actif/inactif.
`setLocation(int, int), getLocation()`	Fixer et obtenir la position du composant.
`setVisible(boolean)`	Montrer ou cacher un composant.
`setForeground(Color), getForeground()`	Fixer et obtenir la couleur d'arrière-plan.
`setBackground(Color), getBackground()`	Fixer et obtenir la couleur de premier plan.
`setFont(Font), getFont()`	Fixer et obtenir la police de caractères.
`addMouseListener(MouseListener)`	Ajouter un écouteur à événements de souris pour le compoasnt.
`addKeyListener(KeyListener)`	Ajouter un écouter à événement de touche de clavier pour le composant.
`repaint(Graphics)`	Planifier le ré-affichage d'un composant.
`paint(Graphics)`	Ré-afficher le composant.

- Un `Container` est un composant qui peut en inclure d'autres.
- Parmi les méthodes de `Container` :

methode()	But
`setLayout(LayoutManager)`	Préparer le gestionnaire de positionnement (layout manager) pour l'affichage.
`add(Component), remove(Component)`	Ajouter ou enlever un composant de l'affichage.

- Une `Window` (fenêtre) est une surface à 2 dimensions sur laquelle on peut dessiner et qui peut être affichée. Parmi les méthodes de `Window` :

methode()	But
show()	Rendre la fenêtre visible.
toFront()	Mettre la fenêtre au premier plan.
toBack()	Mettre la fenêtre en arrière plan.

1.4 Classe Frame

- Un Frame (chassis) est une fenêtre qui comporte une barre de titre, une barre de menu, une bordure, un curseur et d'autres propriétés.
- Parmi les méthodes de Frame :

methode()	But
setResizable()	Rendre la fenêtre à taille variable.
setTile(String), getTitle()	Fixer ou obtenir le titre.
setCursor(int)	Fixer le curseur.
setMenuBar(MenuBar)	Fixer la barre de menu de la fenêtre.

- Utilisation dans l'exemple du flipper de la section 2.1, p. 56 de

methode()	But
setTitle(String)	Héritée de Frame.
setSize(int, int)	Héritée de Component.
show()	Héritée de Window.
paint()	Héritée de Component, redéfini.

- La puissance d'AWT, comme celle de tout cadre logiciel, vient de l'utilisation de variables polymorphes.
- Lorsque la méthode show() de la classe Window est appelée :
  - elle appelle setVisible() de la classe Component ;
  - cette dernière appelle repaint(),
  - qui appelle paint().
- Le code de l'algorithme utilisé par setVisible() et repaint() réside dans la classe Component. Lorsqu'elle est exécutée, le cadre (AWT) "pense" qu'il ne s'agit que d'un instance de Component.
- En réalité, la méthode paint() qui est finalement exécutée est celle redéfinie dans la classe PinBallGame.

- Le code des classes parent (`Component`, `Container`, `Window` et `Frame`) est donc en un sens générique.
- Il suffit de redéfinir la méthode `paint()` pour avoir un comportement spécifique à l'application.
- La combinaison de l'héritage, de la redéfinition et du polymorphisme permet une réutilisation à grande échelle.

1.5 Classe `JFrame`

- Fenêtre générale
- Une classe héritée de `JFrame` contiendra le `main()` de l'application
- Contient une fille unique, de type `JRootPane`
- Cette fille contient 2 filles :
  - `layeredPane` de type `JLayeredPane`, responsable des composants pouvant être empilés les uns au dessus des autres
  - `glassPane` de type `JPanel`, responsable de la capture des événements souris
- `layeredPane` a 2 filles :
  - `MenuBar` de type `JMenuBar`, responsable des composants menus
  - `contentPane` de type `JPanel`, responsable des autres composants
- sous-classe de `Frame` de AWT
- Fonctionne de la même façon à part qu'elle contient un un `contentPane` de type `JPanel`
  - Tout composant (ou sous-Panel) `c` doit être ajouté à la contentPane par `getContentPane().add(c)`
  - Le gestionnaire de positionnement est celui de la contentPane (`BorderLayout` par défaut) Il se modifie par `getContentPane().setLayout()`
- Possibilité de définir un comportement par défaut quand on la ferme (exécuté APRES les `windowClosing()` des `WindowListener` enregistrés) par `setDefaultCloseOperation()`

1.6 Composants Usuels

- `JComponent` : ancêtre de tous les composants Swing (sauf `JApplet, JDialog, JFrame`)
- `JButton` : bouton usuel
- `JCheckBox` : case cochable ("indépendante")
- `JLabel` : texte affiché par le programme
- `JList` : liste "scrollable" de choix
- `JTextField, JTextArea` : pour entrer du texte
- `JPanel` : conteneur de base pour grouper des composants

- `JDialog` : fenêtre secondaire ("esclave")
- `JMenu, JMenuBar, JMenuItem, JPopupMenu` : pour les menus
- `JScrollBar` : barre de défilement
- `JScrollPane` : pour donner une vue d'un seul composant avec défilement
- `JWindow` : fenêtre sans barre de titre ni bordure

1.7 Conteneurs

classe abstraite `Container` (sous-classe de `Component`), avec sous-classes :
- `Panel` : conteneur de base, pour grouper des composants Admet un analogue Swing `JPanel` héritant de `JComponent`
- `ScrollPane` : conteneur avec barres de défilement, mais pour un seul composant Admet un analogue Swing `JScrollPane` héritant de `JComponent`
- `Window` : fenêtre (sans bordure, ni titre), pour création fenêtres personelles par dérivation Admet un analogue Swing `JWindow` héritant de `Window`
- `Frame` : Fenêtre "usuelle" avec bandeau en haut Admet un analogue Swing `JFrame` héritant de `Frame`
- `Dialog` : fenêtre secondaire, associée à une fenêtre maîtresse (notamment pour pop-up de dialogue) Admet un analogue Swing `JDialog` héritant de `Dialog`
- `FileDialog` : fenêtre de sélection de fichier

1.8 Composants spécifiquement Swing

- `JCheckBox` : boîte à cocher permettant divers choix mutuellement
- `JRadioButton` : sorte de `JCheckBox`, de forme circulaire, permettant des choix mutuellement exclusifs via un `ButtonGroup`
- `JComboBox` : liste déroulante
- `JPasswordField` : sorte de `JTextField` masquant les caractères tapés (par exemple pour saisie de mot de passe)
- `JTextPane` : zone de texte éditable avec police de caractères et style
- `JSlider` : curseur pour choisir graphiquement une valeur numérique
- `JToolTip` : bulle d'aide
- `JProgressBar` : barre d'avancement de tâche
- `JTable` : tableau (éditable) de données
- `JTree` : représentation de données arborescentes (façon explorateur Windows)
- `JToolbar` : barre d'outils
- `JColorChooser` : utilitaire pour choix de couleur
- `JFileChooser` : utilitaire pour choix de ficher avec exploration possible

1.9 Conteneurs spécifiquement Swing

- `JOptionPane` : boites de dialogue usuelles, que l'on peut créer et afficher par un simple appel de fonction :
  - `JOptionPane.showMessageDialog()` –> message + 1 bouton OK
  - `int r = JOptionPane.showConfirmDialog()` –> message + boutons style Oui / Non / Annuler
  - `int r = JOptionPane.showOptionDialog()` –> message + choix de boutons
  - `String s = JOptionPane.showInputDialog()` –> message + zone saisie texte + boutons OK / Annuler
- `JSplitPane` : pour afficher deux composants côte à côte (ou l'un au-dessus de l'autre), et avec ligne de séparation déplaçable par l'utilisateur
- `JTabbedPane` : regroupement de plusieurs composants accessibles via des onglets
- `JFileChooser` : fenêtre de sélection de fichier (? FileDialog de AWT mais en mieux)
- `JInternalFrame` : pour faire des sous-fenêtres dans une fenêtre ("bureau virtuel")

1.10 Classe `JPanel`

- Conteneur très général, dérive directement de `JComponent`
- Contient un `FlowLayout` par défaut
- Est opaque, ce qui importe pour les dessins

1.11 Classe `JLayeredPane`

- Conteneur général pour des composants en couches.
- On peut donner des valeurs de niveau aux composants indiquant qui est affiché au dessus.
- Utilise le null Layout, donc positionner ses enfants avec setBounds().
- Classe mère de `JDesktopPane`.

IV.2 Gestion des événements

2.1 Gestion des événements

- Modèle émetteur/récepteur, avec séparation claire entre
  - les éléments d'interface qui émettent les événements,

  - et des objets récepteurs qui "écoutent" les événements et agissent en conséquence.
- Classes dans le paquetage `java.awt.event`

2.2 Modèle d'événements

- Chaque composant peut générer des événements (classe abstraite `AWTEvent` et ses sous-classes `MouseEvent`, `ActionEvent`, ...)
- Tout objet `o` qui doit réagir quand un type d'événement se produit dans un certain composant `c` doit :
  - implanter l'interface adéquate (`MouseListener`, `ActionListener`, ...)
  - être enregistré dans la liste des objets intéressés par ce type d'événements issus de ce composant (`c.addMouseListener(o)`, `c.addActionListener(o)`, ...)
- quand un événement se produit sur le composant `c`, il est transmis à tous les récepteurs enregistrés chez lui pour ce type d'événément, ceci par appel de sa méthode correspondante, par ex., pour appui sur bouton souris, `o.mousePressed(evt)` pour tous les `o` concernés, et où `evt` est l'événement

2.3 Types d'événements

Nom	Description
WindowEvent	apparition, iconification, (dé-)masquage, fermeture, ...
ComponentEvent	redimensionnement, déplacement, ...
FocusEvent	début/fin de sélection comme destinataire des entrées (clavier et souris)
KeyEvent	clavier (touche appuyée, ...)
MouseEvent	souris (boutons, déplacement, ...)
ActionEvent	appui sur `JButton`, double-clic sur item de `JList`, Return dans un `JTextField`, choix d'un `JMenuItem`, ...
ItemEvent	(dé-)sélection d'une `JCheckbox`, d'un item de `JList`, passage sur un `JMenuItem`, ...
TextEvent	modification de texte
ContainerEvent	ajout/suppression de composant
AdjustementEvent	défilement de `JScrollbar`

2.4 Sources d'événements

Événement	Source

Tous les `Component`	`ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent`
Toutes les `Window` `MenuItem`	`WindowEvent`, `Button` `ActionEvent`
`List`	`ActionEvent`, `ItemEvent`
`CheckBox`, `CheckBoxMenuItem`, `Choice`	`ItemEvent`
`TextField`	`ActionEvent`, `TextEvent`
`TextArea` et autres, `TextComponent`	`TextEvent`
Tous les `Container`	`ContainerEvent`

2.5 Interfaces récepteurs

A chaque classe `XxxEvent` est associée une interface `XxxListener`, qui regroupe une ou plusieurs méthodes (lesquelles passent toutes l'événement en paramètre) :

Événement	Interface	Méthodes associées
KeyEvent	KeyListener	keyPressed(), keyReleased(), keyTyped()
MouseEvent	MouseListener	mouseClicked(), mouseEntered(), mouseExited(), mousePressed(), mouseReleased()
MouseEvent	MouseMotionListener	mouseDragged(), mouseMoved()
ComponentEvent	ComponentListener	componentHidden(), componentMoved(), componentResized(), componentShown()
FocusEvent	FocusListener	focusGained(), focusLost()
WindowEvent	WindowListener	windowActivated(), windowClosed(), windowClosing(), windowDeactivated(), windowDeiconified(), windowIconified(), windowOpened()

`ActionEvent`	`ActionListener`	`actionPerformed()`
`ItemEvent`	`ItemListener`	`itemStateChanged()`
`TextEvent`	`TextListener`	`textValueChanged()`
`AdjustmentEvent`	`AdjustmentListener`	`adjustmentValueChanged()`
`ContainerEvent`	`ContainerListener`	`componentAdded()`, `componentRemoved()`

IV.3 Gestionnaire de disposition

3.1 Schéma général

- Un gestionnaire de disposition est responsable de la disposition des composants qu'il contient.
- En général , l'utilisateur choisit un gestionnaire parmi ceux disponibles et ce dernier se débrouille pour la disposition.
- Chaque gestionnaire implante l'interface `LayoutManager`.
- On a en fait le schéma suivant :
  - L'application hérite de `Container`. Cela lui permet de bénéficier des méthodes de `Window` et d'en redéfinir (par ex. `paint()`).
  - `Container` contient comme l'un de ses champs le gestionnaire de disposition. En fait la variable correspondante est polymorphe et c'est par ex. un
  - `GridLayout` qui implante en fait l'interface `LayoutManager`, de manière transparente au `Container`.
- Il y a donc combinaison d'héritage, de composition et d'implantation d'interface.

3.2 Le gestionnaires `BorderLayout`

- Il y a 5 types standard de gestionnaires : `BorderLayout`, `GridLayout`, `CardLayout`, `FlowLayout` et `GridBagLayout`.
- `BorderLayout` : gestionnaire par défaut pour les classes dérivant de `Frame`. Ne peut contenir plus de 5 composant différents. Les 5 positions sont `North`, `South`, `East`, `West` et `Center`.
- Les composants présents occupent toute la place disponible.
- Pour ajouter un composant, on utilise `add()` avec pour 1 ier argument sa position. Par ex. :

```
add("North", new Button("quit")) ;
```

3.3 Le gestionnaires `GridLayout`

- Les composants sont organisés selon une grille rectangulaire, chaque composant ayant la même taille.
- On fournit au constructeur :
  - en 2 premiers paramètres les nombres de lignes et de colonnes de la grille ;
  - en 2 derniers paramètres l'espace hrizontal et vertical inséré entre les composants.
- Exemple :

```
Panel p = new Panel();
p.setLayout(new GridLayout(4, 4, 3, 3));
p.add(new ColorButton(Color.black, "black"));
```

3.4 Le gestionnaires `FlowLayout`

- Au sein d'un `FlowLayout` les composants sont disposés en rangées, de gauche à droite et de haut en bas.
- Contrairement au cas d'un `GridLayout`, les composants peuvent avoir des tailles différentes.
- Lorsqu'un composant ne peut pas tenir dans la ligne sans être tronqué, une nouvelle ligne est créée.
- Gestionnaire par défaut pour la classe `Panel`.

3.5 Le gestionnaires `CardLayout`

- Dans un `CardLayout`, les composants sont empilés verticalement (l'un au dessus de l'autre), un seul composant étant visible à la fois.
- Chaque composant est repéré à l'aide d'une chaîne de caractères lors de son ajout par `add()`.
- Cette chaîne est ensuite utilisée pour rendre un composant visible.
- Exemple :

```
CardLayout lm = new CardLayout();
Panel p = new Panel(lm);
p.add("Un", new Label("Numero Un"));
p.add("Deux", new Label("Numero Dexu"));
```

- Type le plus général de gestionnaire de disposition : `GridBagLayout`.
- Génération d'une grille non uniforme de carrés et placement des composants dans chacun des carrés.

3.6 Application avec IHM

- Ecrire une sous-classe de `JFrame` adaptée à l'application dans son constructeur, Créer les sous-conteneurs et composants élémentaires, et installer chacun à sa place dans son conteneur (méthode `add()`). Redéfinir éventuellement la méthode `paint(Graphics)` pour y faire tout ce qui est dessin dans le `main()` :
  - Créer une instance `f` de la sous-classe de Frame en question
  - dimensionner : `f.setSize(w,h)`
  - positionner : `f.setLocation(x,y)`
  - afficher la fenêtre : `f.show()`

IV.4 Composants d'interface utilisateur spécifiquement Awt

4.1 Étiquette

- Tous les composants d'interface utilisateur, sauf les menus, sont des sous classes de `Component`.
- Une étiquette ou `Label` contient du texte qui est affiché.
- Exemple :

```
Label lab = new Label("score : 0 a 0");
add("South", lab);
```

- Une étiquette ne répond à aucun événement (clic de souris ou touche de clavier).
- Le texte d'une étiquette peut être changé *via* `setText(String)`. Il peut être obtenu par `getText()`.

4.2 Bouton

- Un bouton est un composant souvent représenté par une forme ronde et répondant aux interactions de l'utilisateur.
- On obtient une intercaction avec un bouton en attachant un objet `ActionListener` (écouteur à actions) au bouton.
- L'objet `ActionListener` est averti lorsque le bouton est enfoncé.

```
Button b = new Button("Fais le");
b.addActionListener(new doIt());

private class doIt implements ActionListener {
    public void actionPerformed(ActionEvent e) {
```

```
    ...
    }
}
```

- Une technique utile est de combiner le bouton et l'écouteur en une nouvelle et même classe. Par ex.

```
private class ColorButton extends Button implements ActionListener
{
    private Color ourColor;

    public ColorButton(Color c, String name)
    {
        super(name);  // creation du bouton
        ourColor = c; // sauvegarder la valeur de couleur
        addActionListener(this); // on s'ajoute comme ecouteur
    }

    public void actionPerformed(EctionEvent e)
    {
        setFromColor(ourColor);  // fixer la couleur du panneau central
    }
```

- L'objet s'enregistre lui-même comme un écouteur à actions sur boutons.
- Lorsqu'enfoncé, le bouton appelle la méthode `ActionPerformed()`.
- On peut même définir une classe abstraite :

```
abstract class ButtonAdapter extends Button implements ActionListener
{
    public ButtonAdapter(String name)
    {
        super(name);
        addActionListener(this);
    }

    public void actionPerformed(ActionEvent e)
    {
        pressed();
    }

   public abstract void presses();
}
```

qui est à la fois un bouton et un écouteur.
- Pour créer un bouton en utilisant cette abstraction, le programmeur doit créer une sous-classe et redéfinir la mt́hode `pressed()`. Par exemple :

```
Panel p = new Panel();

p.add(new ButtonAdapter("Quit"){
    public void pressed() { System.exit(0); } });
```

4.3 Canevas

- Un canevas ou `Canvas` est un composant ayant seulement une taille et la possibilité d'être une cible pour des opérations de dessin.
- Le canevas est souvent étendu (*via* jcextends) pour obtenir d'autres types de composants. Un exemple est la classe `ScrollPane`.

4.4 Ascenseur

- Un ascenseur ou `ScrollBar` (en général un carré que l'on peut faire glisser) est utilisé pour spécifier des valeurs entières dans un intervalle.
- Il est affiché dans une direction horizontale ou verticale.
- On peut spécifier :
  - les bornes de l'intervalle ;
  - l'incrément de ligne (la distance dont l'ascenseur se déplace lorsque l'on clique sur l'une de ses extrémités, souvent des petits triangles) ;
  - l'incrément de page (la distance dont l'ascenseur se déplace lorsque l'on clique sur une partie d'arrière plan entre le carré glissant et l'une de ses extrémités).
- Comme un bouton, une interaction avec l'utilisateur est fournie en définissant un écouteur qui est averti lorsque l'ascenseur est modifié.
- Exemple d'une classe `ColorBar` qui représente un ascenseur pour régler l'intensité d'une couleur. Le constructeur de la classe créé un ascenseur vertical avec une valeur initiale de 40 et un intervalle égal à [0,255].

```
private class ColorBar extends ScrollBar implements AdjustmentListener
{
    public ColorBar(Color c)
    {
        super(ScrollBar.VERTICAL, 40, 0, 0, 255);
        setBackground(c);
        addAdjustmentListener(this);
    }

    public void adjustmentValueChanged(AdjustmentEvent e)
    {
        setFromBar();  // prendre la valeur de l'ascenseur en
```

```
                    //   utilisant getValue()
  }

}
```

La couleur de fond est fournie en argument. Enfin l'objet est lui-même un écouteur à événements d'ascenseur. Lorsque l'ascenseur est modifié, la méthode `adjustmentValueChanged()` est exécutée.
Pour les 3 couleurs primaires, on prendra 3 ascenseurs et leur valeur sera obtenue *via* la méthode `setFromBar()`.

4.5 Composants texte

- Un tel composant est utilisé pour afficher du texte éditable.
- 2 formes :
  - `TextField` : bloc de taille fixe.
  - `TextArea` : utilise des ascenseurs pour afficher un bloc de texte plus gros dont tout n'est pas forcément visible en même temps

  Le texte peut être fixé ou obtenu *via* `setText(String)` et `getText()`. Du texte supplémentaire peut être rajouté *via* `append(String)`.
- Un `TextListener` peut être attaché à un composant de texte. L'écouteur doit implanter l'interface `TextListener`

```
interface TextListener extends EventListener {
    public void textValueChanged(TextEvent e);
}
```

4.6 Boîtes de contrôle

- Une boîtes de contrôle ou `Checkbox` est un composant qui maintient et affiche un état binaire. Cet état peut être soit "on", soit "off" et une étiquette aficahble y est associée.
- L'étiquette et l'état de la boîte peuvent être fixés par le programmeur en utilisant `getLabel()`, `setLabel()`, `getState()` et `setState()`.
- Changer l'état d'un `Checkbox` créé un événement `ItemEvent`, enregistré par un écouteur à événements `ItemListener`.

```
class ChackTest extends Frame
{
    private Checkbox cb = new Checkbox("Le checkbox est off');

    public static void main(String args[])
    {
```

```
        Frame world = new CheckTest();
        world.show();
    }

    public CheckTest()
    {
        setTitle("Exemple de checkbox");
        setSize(300, 70);
        cb.addItemListener(new CheckListener());
        add("Center", cb);
    }

    private class CheckListener implements ItemListener
    {
        public void ItemStateChanged(ItemEvent e)
        {
            if(cb.getState())
                cb.setLabel("Le checkbox est on");
            else
                cb.setLabel("Le checkbox est off");
        }
    }
}
```

4.7 Groupes de checkbox, choix et listes

- 3 types de composants pour choisir entre plusieurs possibilités :
  - Un groupe de checkbox connectées, une seule pouvant être sélectionnée à la fois. Ceci se nomme parfois un groupe de boutons radio (fonctionnement rappelant celui des boutons de certains autoradois).
  - Un choix ou `Choice` qui n'affiche qu'une sélection à la fois, mais lorsque l'utilisateur clique sur la zone de sélection, un menu instantané (pop-up menu) apparît et permet d'effectuer un choix différent.
  - Une liste ou `List`, similaire au choix, mais plusieurs possibilités parmi celles disponibles peuvent être affichées simultanément et un ascenseur permet d'aller voir les autres possibilités.

```
class ChoiceTest extends Frame
{
    public static void main(String args[])
    {
        Window world = new ChoiceTest();
        world.show();
```

```
    }

    private String[] choices = { "Un", "Deux", "Trois", "Quatre",
        "Cinq", "Six", "Sept", "Huit", "Neuf", "Dix" };
    private Label display = new Label();
    private Choice theChoice = new Choice();
    private List theList = new List();
    private CheckboxGroup theGroup = new CheckboxGroup();
    private ItemListener theListener = new ChoiceListener();

    public ChoiceTest()
    {
        setTitle("Example de selections");
        setSize(300, 300);
        for(int i = -; i < 10; i++)
        {
            theChoice.addItem(choices[i]);
            theList.addItem(choices[i]);
        }
        theChoice.addItemListener(theListener);
        theList.addItemListener(theListener);
        add("West", makeCheckBoxes());
        add("North", theChoice);
        add("East", theList);
        add("South", display);
    }

    private class ChoiceListener implements ItemListener
    {
        public void itemStateChanged(ItemEvent e)
        {
            display.setText( theGroup.getSelectedCheckboxGroup().getLabel() +
                theList.getSelectedItem() +  theChoice.getSelectedItem() );
        }
    }

    private Panel makeCheckBoxes()
    {
        Panel p = new Panel(new GridLayout(5, 2));
        for(int i = 0; i < 10; i++)
        {
            Checkbox cb = new Checkbox(choices[i], theGroup, false);
            cb.addItemListener(theListener);
```

```
            p.add(cb);
        }
        return(p);
    }
}
```

- Un groupe de checkbox ne devrait être utilisé que lorsque le nombre d'alternatives est petit.
  Un choix ou une liste devrait être utilisé si le nombre d'alternatives dépasse 5.
- Pour créer un `Choice` ou une `List`, le programmeur spécifie chaque alternative en utilisant `addItem()`.
- Un `ItemListener` peut être attaché à l'objet. Lorsqu'une sélection est faite, l'écouteur en est informé *via* la méthode `itemStateChanged()`. Le texte de l'entrée sélectionnée peut être obtenu *via* `getSelectedItem()`.
- Pour structurer un groupe de checkbox, le programmeur créé d'abord un `CheckboxGroup`. Cette référence est ensuite passée comme argument à chacune des checkbox crée avec un [e] argument indiquant si le checkbox doit être initialement actif. Le dernier checkbox sélectionné peut être obtenu en utilisant `getSelectedCheckbox()`.
- Puisqu'un groupe de checkbox est construit avec plusieurs composants, il est presque toujours disposé sur un panneau (ou `Panel`). Dans l'exemple, une grille 5×2 est utilisée comme disposition pour les 10 checkbox.

4.8 Panneau

- Un panneau `Panel` est un `Container` qui agit comme un `Component`. Il repŕsente une région rectangulaire qui possède son proper gestionnaire de disposition (qui peut être différent de celui de l'application).
- Des composants peuvent être insérés dans un panneau. Le panneau est alors lui-même inséré comme une seule unité dans la fenêtre de l'application.
- Utilisation d'un panneau dans l'exemple précédent. La méthode `makeChekcboxes()` crée un panneau pour contenir 10 checkbox. Il est structuré en utilisant un `GridLayout` 5×2. Puis ce groupe de 10 éléments est traité comme un unique élément et est placé à gauche de la fenêtre de l'application.

4.9 Carreau défilant

- Un carreau défilant ou `ScrollPane` est similaire à un panneau, mais il ne peut contenir qu'un élément. Et si cet élément est trop grand pour être visible d'un seul coup, un ou 2 ascenseurs sont automatiquement générés pour pouvoir se déplacer.

– Exemple d'une fenêtre de 300×300 pixels avec un `ScrollPane` contenant un `Canvas` de 1000×1000 pixels.

```
class BigCanvas extends Frame
{
    public static void main(String args[])
    {
        Frame world = new BigCanvas();
        world.show();
    }

    private Polygon poly = new Polygon();
    private Canvas cv = new Canvas();

    public BigCanvas()
    {
        setSize(300, 300);
        setTitle("Test de ScrollPane");
        cv.setSize(1000, 1000);
        cv.addMouseListener(new MouseKeeper());
        ScrollPane sp = new ScrollPane();
        sp.add(cv);
        add("Center", sp);
    }

    public void paint(Graphics g)
    {
        Graphics gr = cv.getGraphics();
        gr.drawPolygon(poly);
    }

    private class MouseKeeper extends MouseAdapter
    {
        public void mousePressed(MouseEvent e)
        {
            poly.addPoint(e.getX(), e.getY());
            repaint();
        }
    }
}
```

IV.5 Un exemple

5.1 Affichage de couleurs

- La classe `ColorTest` crée une fenêtre pour afficher des couleurs et leur valeur. La fenêtre est divisée en 4 régions. Les régions sont gérées par le gestionnaire par défaut, de type `BorderLayout`.
- En haut (région nord) se trouve une zone de texte, un composant de type `TextField` qui décrit la couleur courante.
- À gauche (région est) se trouve un trio d'ascenseurs qui peuvent être utilisés pour fixer les valeurs des canaux rouge, vert et bleu.
- ‘A droite (région ouest), il y a un banc 4×4 de 16 boutons. Ils sont construits sur un `Panel` qui est géré par un gestionnaire `GridLayout`. 13 boutons représentent des couleurs prédéfinies. 2 autres servent à jouer sur la luminance. Le dernier sort de l'application.
- Au milieu se trouve un panneau dans lequel on affiche la couleur.
- La classe `ColorTest` contient 6 champs :
  - la couleur courante du panneau du milieu,
  - les 3 ascenseurs de réglage des couleurs primaires,
  - le champ de texte en haut,
  - le panneau (coloré) du milieu.

```
class ColorTest extends Frame
{
    public static void main(String args[])
    {
        Frame window = new ColorTest();
        window.show();
    }

    private TextField colorDescription = new TextField();
    private Panel ColorField = new Panel();
    private Color current = Color.black;
    private ScrollBar redBar = new ColorBar(Color.red);
    private ScrollBar greenBar = new ColorBar(Color.green);
    private ScrollBar blueBar = new ColorBar(Color.blue);

    public ColorTest()
    {
        setTitle("Test de couleurs");
        setSize(400, 600);
        add("North", colorDescription);
        add("East", makeColorButtons);
```

```
        add("Center", colorField);
        add("West", makeScrollbars());
        setFromColor(current);
    }

    private void setFromColor(Color c)
    {
        current = c;
        colorField.setBackground(current);
        redBar.setValue(c.getRed());
        greenBar.setValue(c.getGreen());
        blueBar.setValue(c.getBlue());
        colorDescription.setText(c.toString());
    }

    private void setFromBar()
    {
        int r = redBar.getValue();
        int g = greenBar.getValue();
        int b = blueBar.getValue();
        setFromColor(new Color(r, g, b));
    }

    private Panel makeColorButtons() { ... }
    private Panel makeScrollBars() { ... }

    private class BrightenButton extends Button implements ActionListener ...
    private class ColorButton extends Button implements ActionListener ...
    private class ColorBar extends ScrollBar implements AdjustmentListener ...
}
```

- Les 3 ascenseurs utilisent la classe `colorBar` décrite précédemment. On fournit au constructeur les couleurs du carré glissant et du fonc. Lors de l'appel de l'écouteur, on invoque `adjustmentValueChanged()` qui appelle `setFromBar()`.
- `makeScrollBars()` crée le panneau qui contient les 3 ascenseurs.
- Chaque instance de `ColorButton`, décrit précédemment, hérite de la classe `Button` et implante l'interface `ActionListener`. Lorsque le bouton est enfoncé, la méthode `setFromColor()` est appelée pour mettre à jour la couleur du panneau central.
- La classe `BrightenButton` utilise un indice (`index`) qui indique si le bouton augmente ou diminue la clarté (éclairer ou assombrir).

```
private class BrightenButton extends Button implements ActionListener
```

```
{
    private int index;
    public BrightenButton(int i)
    {
        super(i == 0 ? "eclaire" : "assombrir");
        index = i;
        addEctionListener(this);
    }

    public void ActionPerformed(ActionEvent e)
    {
        if(index == 0)
            setFromColor(current.brighter());
        else
            setFromColor(current.brighter());
    }
}
```

Les 16 boutons sont définis par :

```
private Panel makeColorButtons()
{
    Panel p = new Panel();

    p.setLayout(new GridLayout(4,4,3,3));
    p.add(new ColorButton(Color.black, "black"));
    p.add(new ColorButton(Color.blue, "blue"));
    p.add(new ColorButton(Color.cyan, "cyan"));
    p.add(new ColorButton(Color.darkGray, "darkGray"));
    p.add(new ColorButton(Color.gray, "gray"));
    p.add(new ColorButton(Color.green, "green"));
    p.add(new ColorButton(Color.lightGray, "lightGray"));
    p.add(new ColorButton(Color.magenta, "magenta"));
    p.add(new ColorButton(Color.orange, "orange"));
    p.add(new ColorButton(Color.pink, "pink"));
    p.add(new ColorButton(Color.red, "red"));
    p.add(new ColorButton(Color.white, "white"));
    p.add(new ColorButton(Color.yellow, "yellow"));
    p.add(new BrightenButton(0));
    p.add(new BrightenButton(1));

}
```

IV.6 Boîtes de dialogue

6.1 Descrition d'un dialogue

- Un `Dialog` est un type de fenêtre spécial affichée pendant une période de temps limitée. Les dialogues sont souvent utilisés pour prévenir l'utilisateur de certains événements ou pour poser des questions simples.
- Les dialogues peuvent être modaux ou non modaux. Un dialogue modal exige une réponse de l'utilisateur et empêche ce dernier d'effectuer une autre action jusqu'à ce que le dialogue soit terminé.
- Un dialogue non modal peut être ignoré par l'utilisateur. Les actions d'un dialogue non modal sont souvent placées dans une action (thread) séparée ; de cette manière, les actions produites par le dialogue ne perturbent pas le reste de l'application.
- Les 2 arguments d'un constructeur de `Dialog` sont le `Frame` de l'application et une valeur booléenne qui est vraie si le dialogue est modal.
  `Dialog dial = new Dialog(this, false) ;`
- Un `Dialog` étant de type `Window`, des composants graphiques peuvent y être placés, comme dans `Frame` ou dans un `Panel`. Le gestionnaire de positionnement par défaut est un `BorderLayout`.
- La plupart des fonctions utilisées par un `Dialog` sont héritées des classes parentes. Parmi celles-ci

methode()	But
setSize(int, int)	Fixer la taille de la fenêtre
show()	Afficher la fenêtre
setVisible(false)	Faire disparaître la fenêtre
setTitle(String), getTitle()	Fixer ou obtenir le titre de la fenêtre.

Pour un dialogue modal, la méthode `show()` ne se termine que lorsque le dialogue disparaît. Il faut donc appeler à un moment `setVisible(false)`.

6.2 Exemple de dialogue

- Exemple

```
class Dialog extends Frame
{
    public static void main(String args[])
    {
        Frame world = new BigCanvas();
```

```
        world.show();
    }

    private TextArea display = new TestArea();
    private Chaeckbox cb = new Checkbox("Dialogue modal");

    public DialogTest()
    {
        setTitle("Programme de test de dialogue");
        setSize(300, 220);
        add("West", cb);
        add("East", new Makebutton());
        add("South", display);
    }

    private class Makebutton extends ButtonAdapter
    {
        public Makebutton() { super("Creer le dialogue"); }
        public void pressed() { makeDialog(cb.getState()); }
    }

    private void makeDialog(boolean modalFlag)
    {
        final Dialog dlg = new Dialog(this, modalFlag);
        dlg.setSize(100, 100);
        dlg.add("North", new CountButton(1));
        dlg.add("West", new CountButton(2));
        dlg.add("East", new CountButton(3));
        dlg.add("South", new ButtonAdapter("Hide") {
            public void pressed() { dlg.setVisible(false); }});
        dlg.show();
    }

    private class CountButton extends ButtonAdapter
    {
        public CountButton(int val)
        {
            super("" + val);
        }
        public void pressed()
        {
            display.append("Bouton " + getLabel() + "presse\n");
        }
```

```
    }

}
```

Création d'une fenêtre avec un checkbox, un bouton et une zone de texte. Le checkbox permet à l'utilisateur de spécifier si le dialogue à créer sera modal ou non. Le bouton permet de créer le dialogue, et la zone de texte affiche quels sont les boutons enfoncés dans le dialogue.

- La procédure `makeDialog()` crée la boîte de dialogue. La taille de la boîte est fixée à 100×100 pixels et 4 boutons sont placés dans la boîte. 3 des boutons affichent simplement une ligne de texte dans la fenêtre précédente. Le dernier cache le dialogue. Dans le cas d'un dialogue modal, acher le dialogue équivaut à le faire diparaître, et le contrôle est renvoyé à la procédure qui a créé le dialogue.

IV.7 Menus

7.1 Barre de menus

- Une barre de menus n'est pas une sous classe de `Component`, puisque les plate formes diffèrent dans la manière de gérer les barres de menus.
- Une barre de menus contient des menus qui contiennent une suite d'entrées. Une instance de barre de menus peut être attachée à un `Frame` *via* `setMenuBar()` :

```
MenuBar bar = new MenuBar();
setMenuBar(bar);
```

- Les divers menus sont nommés et ajoutés à la barre de menus *via* `add()` :

```
Menu helpMenu = new Menu("Aide");
bar.add(helpMenu);
```

- Des entrées de menus sont crées en utilisant la classe `MenuItem`. Chaque entrée de menu maintient une liste d'objets `ActionListener`. Ces écouteurs seront avertis lorsque l'entrée de menu est sélectionnée

```
MenuItem quitItem = new MenuItem("Quitter");
quitItem.addActionListener(new QuitListener());
helpMenu.add(quitItem);
```

- On peut également créer des menus de type particulier.

7.2 Un utilitaire de menu “quitter”

- La classe `QuitItem` crée un écouteur qui va arrêter l'application lorsque l'entrée de menu correspondante sera sélectionnée.
- En surchargeant le constructeur, on peut ajouter trivialement cet utilitaire à n'importe quelle application

```
class QuitItem implements ActionListener
{
    public QuitItem(Frame application)
    {
        MenuBar mbar = new MenuBar();
        application.setMenuBar(mbar);
        Menu menu = new Menu("Quitter");
        mbar.add(menu);
        MenuItem mitem = new MenuItem("Quitter");
        mitem.addActionListener(this);
        menu.add(mitem);
    }

    public QuitItem(MenuBar mbar)
    {
        Menu menu = new Menu("Quitter");
        mbar.add(menu);
        MenuItem mitem = new MenuItem("Quitter");
        mitem.addActionListener(this);
        menu.add(mitem);
    }

    public QuitItem(Menu menu)
    {
        MenuItem mitem = new MenuItem("Quitter");
        mitem.addActionListener(this);
        menu.add(mitem);
    }

    public QuitItem(MenuItem mitem)
    {
        mitem.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e)
    {
        System.exit(0);
```

```
    }
}
```

- Le constructeur de `QuitItem` peut avoir comme argument :
  - un `MenuItem`
  - un `Menu`, auquel cas une entrée étiquetée `"Quitter"` est crée.
  - un `MenuBar` ; un menu est alors créé, étiqueté `"Quitter"`, ainsi qu'une entrée de menu.
  - un `Frame` (l'application entière), auquel cas, une barre de menus, un menu et une entrée de menu sont créés.

V – Classe `Url`

Références bibliographiques

– *Java et Internet – Concepts et programmation*, G. Roussel, E. Duris, N. Bedon et R. Forax [RDBF02],
– *Java Network Programming*, E.R. Harold [Har97].

V.1 Classe `URL`

1.1 Généralités

– Un URL ("Uniform Resource Locator" ou localisateur de ressource uniforme) est un schéma de nommage et d'accès à des objets sur le réseau. L' objet est de fournir une interface simple afin d'extraire des objets du réseau. Un URL consiste en 6 champs : le *protocole*, le *nom de d'hôte* (ou de machine), le *numéro de port*, le `chemin`, le *nom de fichier* et la *section*. Exemple : `http ://sunsite.unc.edu :8080/pub/sound/monkey.au`. On a ici :
  – le protocole : `http`
  – le nom d'hôte : `sunsite.unc.edu`
  – le numéro de port : `8080`
  – le chemin de fichier : `/pub/sound/`
  – le nom de fichier : `monkey.au`.
– Syntaxe générale :
  `protocole ://nom-machine :no-port/chemin/nom-fichier#section`
– Informations sur les URLs à :
  `http ://www.ncsa.uiuc.edu/demoweb/url-primer.html`
– La classe `URL` permet d'obtenir des fichiers HTTP, des fichiers URLs et certains types multimédias.
– Il y a en fait six classes et interfaces :
  – La classe `URL`, pour rapatrier des URLs du réseau.
  – La classe `URLConnection`

1.2 Constructeurs

- La classe `URL` représente un URL, comme utilisé sur le Web.
- `URL(String url) throws MalformedURLException`
  Créé un objet de type `URL` à partir d'une spécification textuelle, typiquement de la forme `http ://www.nas.gov/index.html`
- `URL(String protocol, String host, String file) throws MalformedURLException`
  Ce constructeur suppose un numéro de port par défaut de l'URL. Ce numéro dépend du protocole et doit être géré par un gestionnaire de protocole.
- `URL(String protocol, String host, int port, String file) throws MalformedURLException`
  Ce constructeur crée un URL de toutes pièces. Le protocole est typiquement `http` ou `ftp`.
- `URL(String protocol, String host, int port, String file, URLStreamHandler handler) throws MalformedURLException`
  Ce constructeur crée un URL de toutes pièces avec un gestionnaire spécifique de flux d'URL, de type `URLStreamHandler`. C'est le mécanisme recommandé pour que des applets du JDK 1.2 utilisent des gestionnaires de protocole spécifiques.
- `URL(URL context, String relative)`
  Crée un URL à partir d'un URL existant `context` et d'un URL textuel relatif `relative`. Par exemple, si `context` est `http ://serveur/` et `relative` est `document.html`, l'URL crée correspondra à `http ://serveur/document.html`. Si `relative` est un URL absolu (commençant par le champ de protocole, par ex. `http :`), le résultat est juste un URL correspondant à `relative`. Les applets se serviront souvent de ce constructeur, utilisant le résultat de `getCodeBase()` comme contexte.
- `URL(URL context, String relative, URLStreamHandler handler) throws MalformedURLException`
  Crée un URL à partir d'un URL existant `context` et d'un URL textuel relatif `relative` avec un gestionnaire spécifique de flux d'URL, de type `URLStreamHandler`. Par défaut, le nouvel URL hérite d'un `URLStreamHandler` de `context`. C'est le mécanisme recommandé pour que des applets du JDK 1.2 utilisent des gestionnaires de protocole spécifiques.

1.3 Méthodes et exception

Les méthodes suivantes parmettent de dissecter un URL et d'en obtenir les différents champs.

- `String getProtocol()`
  Renvoie le champ protocole de l'URL. Ce sera par ex. `http`, `ftp`, `mailto`, etc.

- `String getHost()`
  Renvoie le champ nom d'hôte de l'URL.
- `int getPort()`
  Renvoie le champ numéro de port de l'URL.
- `String getFile()`
  Renvoie les champ chemin et nom de fichier de l'URL.
- `String getRef()`
  Renvoie le partie référence (ou section) de l'URL, si elle est présente (partie suivant le signe `#`).
- `boolean sameFile(URL other)`
  Renvoie `true` si l'URL est égale à `other` en ignorant le champ référence (ou section) si elle est présente.
- `String toExternalForm()`
  Renvoie une représentation sous forme de `String` de l'URL, par ex. `http ://x.y.z :80/fi`
- `URLConnection openConnection() throws IOException`
  ouvre un socket pour l'URL spécifié et renvoie un objet de type `OpenConnection`. Ce dernier représente une connexion ouverte à une ressource réseau. Cette méthode est utilisée lorsque l'on veut communiquer directement avec le serveur. L'objet de type `URLConnection` donne accès à tout ce qui est envoyé par le serveur ; en plus du document lui-même, sous forme brute (HTML, texte ou binaire), l'on voit tous les en-têtes requis par le protocole en cours d'utilisation. Par exemple, si l'on rapatrie un document HTML, `GetConnection()` donnera accès au en-têtes HTTP suivi du flux HTML brut.
- `InputStream openStream() throws IOException`
  Cette méthode ouvre une connexion pour l'URL correspondant et renvoie un objet de type `InputStream` afin de lire son contenu. Les données obtenues sont de type brut (données ASCII pour un fichier texte, code HTML pour un fichier HTML, etc.) et ne contiennent pas les en-tête de protocoles ; pour les obtenir, se servir de `openConnection()`. Dans le cas du protocole HTTP, un appel à `openStream()` envoie automatiquement une requête HTTP et traite les en-tête associés. Cet appel est équivalent à un premier appel à `openConnection()` suivi d'un appel à `getInputStream()` de la classe `URLConnection`.
- `Object getContent() throws IOException`
  Cette méthode récupère le contenu de l'URL et renvoie une référence à un type `Object`. Le “type réel” de l'objet dépend du contenu de l'URL. Si l'URL est une image et qu'un gestionnaire appriproprié est disponible, un objet de type `ImageProducer` est effectivement renvoyé ; si le fichier est textuel et qu'un gestionnaire est disponible, une objet de type `String` est renvoyé. Cet appel est un raccourci de l'appel `openConnection().getContent()`.
- `protected void set(String protocol, String host, int port, String file,`

`String ref)`
Permet à une sous-classe d'URL de changer les valeurs des différents champs : protocole, nom d'hôte, chemin et nom de fichier, référence.

– `static void setURLStreamHandlerFactory(URLStreamHandlerFactory factory)`
Cette méthode fixe le `URLStreamHandlerFactory` global pour l'application courante. Cette méthode est utilisée lorsque l'on cherche à développer un gestionnaire de protocole spécifique. Le gestionnaire `factory` doit être capable de renvoyer un gestionnaire approprié pour les types de protocoles URL envisagés. Cette méthode ne peut être appelée qu'une fois. Ceci dit, l'environnement Java inclut des gestionnaires de protocoles par défaut, contenant au moins HTTP. L'accès à cette méthode est restreint par le gestionnaire de sécurité en vigueur. Des applets non autorisées ne peuvent appeler cette méthode. Pour cette raison, l'extension (au sens de l'héritage) de la hiérarchie de `URL` n'est utile que pour des applications. Les applets doivent spécifier un gestionnaire, de type `URLStreamHandler`, pour chaque URL qu'elles construisent.
– Une *Exception* : `MalformedURLException`
Indique un URL ayant des champs incorrects. Sous-classe de `IOException`.
– Exemple de chargement d'une page HTML :

```
URL url = new URL("http://java.sun.com/index.html");
InputStream in = url.openStream();
Reader reader = new InputStreamReader(in, "latin1");
BufferedReader bufferedReader = new BufferedReader(reader);
PrintWriter console = new PrintWriter(System.out);
String line;
while ((line = bufferedReader.readLine()) != null)
    console.println(line);
console.flush();
bufferedReader.close();
```

VI – Paquetage java.net : Sockets

Références bibliographiques

- *Java et Internet – Concepts et programmation*, G. Roussel, E. Duris, N. Bedon et R. Forax [RDBF02],
- *Java Network Programming*, E.R. Harold [Har97].

VI.1 Sockets : Rappels TCP/IP

1.1 Socket logique ; numéro de port

- Une socket logique est une paire (@IP, no port). Un numéro de port est nécessairement compris entre 1 et 65 535. Un numéro de port sert è désigner deux choses :
  - Au sein de la machine du client et dans la socket logique contenant l'adresse du serveur (pour s'adresser à ce dernier), le numéro de port désigne le **numéro du service** auquel le client veut s'adresser.
  - Au sein du serveur et dans la socket logique contenant l'adresse du client (pour s'adresser à ce dernier), le numéro de de port sert à distinguer les différents clients (pour le cas où plusieurs clients de la même machine veulent s'adresser au même serveur).

  L'identification unique sur reseau se réalise au moyen d'une socket logique et d'un numéro de protocole.
- Ceci fournit la réponse aux questions :
  - Où ALLER (@ Internet),
  - Quoi FAIRE ou encore QUI contacter (numéro de port),
  - Comment y aller (numéro de protocole).

VI.2 Sockets : Clients

2.1 Classe `InetAdress`

- Abstraction d'une adresse IP ⟹ portage transparent sous IPv6 (de 32 à 128 bits)
- *Pas de constructeur*. Création par `getByName()`, `getLocalHost()`, `getAllByName()`.
- *Méthodes statiques* : `InetAddress getByName()`, `InetAddress getLocalHost(String host)`, `InetAddress[] getAllByName(String host)`.
  - `InetAddress getLocalHost() throws UnkownHostException`. Renvoie un `InetAddress` correspondant à la machine locale, éventuellement 127.0.0.1 (loopback, par ex. derrière un "firewall").
  - `InetAddress getByName(String host) throws UnkownHostException`. Renvoie un `InetAddress` correspondant à l'hôte spécifié en argument. Il peut être spécifié par nom (par ex. `proxy0.att.com` ou par adresse IP `127.0.0.1`. Pas d'autre moyen d'un client pour construire une `InetAddress` pour un hoôte distant.
  - `InetAddress[] getAllByName(String host) throws UnkownHostException`. Renvoie un taleau d' `InetAddress` correspondant à l'ensemble des adresses IP connues de l'hôte spécifié en argument. Ces machines diposent de plusieurs cartes (interfaces) ; c'est par exemple le cas de routeurs.
- *Méthodes d'instance* : `byte[] getAddress()`, `String getHostName()`, `String getHostAddress()`, `boolean isMulticastAddress()`
  - `byte[] getAddress()` Renvoie un tableau de `byte` contenant l'adresse IP en "network byte order" (octet de poids fort d'abord). Pour IPv4, 4 octets.
  - `String getHostName()` Renvoie le nom de l'hôte correspondant à l'objet `InetAddress` appelant. Si le nom n'est pas déja connu, une requête DNS est lancée. Peut échouer derrière des "firewalls".
  - `String getHostAddress()` Renvoie l'adresse IP de l'hôte correspondant à l'objet `InetAddress` appelant sous forme quadri-octet (par ex. `127.0.0.1`)
  - `boolean isMulticastAddress()` Teste si l'objet `InetAddress` correspondant représente une adresse multidiffusion ("multicast") ; c'est-à-dire si le premier octet de l'adresse est compris entre 224 et 239 (inclus).
- *Exceptions* : `UnknownHostException`, `SecurityException`
- `UnknownHostException` Sous-classe de `IOException` indiquant que l'hôte n'a pu être identifié.
- `SecurityException` Levée si le `SecurityManager` interdit une opération. Par ex., une applet générale ne peut construire une `InetAddress` du nom du serveur Web dont elle est originaire.
- *Exemple* : Attend en entrée des noms d'hôte et affiche leur adresse IP.

```
import java.net.*;
import java.io.*;

public class InetExample {
  public static void main (String args[]) {
    printLocalAddress ();
    Reader kbd = new FileReader (FileDescriptor.in);
    BufferedReader bufferedKbd = new BufferedReader (kbd);
    try {
      String name;
      do {
        System.out.print ("Enter a hostname or IP address: ");
        System.out.flush ();
        name = bufferedKbd.readLine ();
        if (name != null)
          printRemoteAddress (name);
      } while (name != null);
      System.out.println ("exit");
    } catch (IOException ex) {
      System.out.println ("Input error:");
      ex.printStackTrace ();
    }
  }

  static void printLocalAddress () {
    try {
      InetAddress myself = InetAddress.getLocalHost ();
      System.out.println ("My name : " + myself.getHostName ());
      System.out.println ("My IP : " + myself.getHostAddress ());
    } catch (UnknownHostException ex) {
      System.out.println ("Failed to find myself:");
      ex.printStackTrace ();
    }
  }


  static void printRemoteAddress (String name) {
    try {
      System.out.println ("Looking up " + name + "...");
      InetAddress machine = InetAddress.getByName (name);
      System.out.println ("Host name : " + machine.getHostName ());
      System.out.println ("Host IP : " + machine.getHostAddress ());
    } catch (UnknownHostException ex) {
```

```
        System.out.println ("Failed to lookup " + name);
      }
    }
  }
```

Notez qu'après un `System.out.print()`, on vide (`flush()`) `System.out`, car la méthode ne le fait pas automatiquement.

2.2 Sockets clientes : Classe `Socket`

- *Constructeurs*
  - Créer une socket établit automatiquement une connexion à l'hôte et au port spécifiés.
  - Il doit y avoir un **serveur en écoute** sur ce port.
  - Sinon, une `IOException` est levée (correspondant à une "connexion refused").
  - `Socket(String host, int port) throws IOException` Crée une socket et la connecte au port spécifié et à l'hôte spécifié. L'hôte est spécifié par son nom ou son adresse IP, le numéro de port doit être entre 1 et 65 535.
  - `Socket(InetAddress address, int port) throws IOException` Crée une socket et la connecte au port spécifié et à l'hôte spécifié par `address`. Le numéro de port doit être entre 1 et 65 535.
  - `Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException` Crée une socket, l'attache à l'adresse `localAddress` et au port `localPort`, puis la connecte au port `port` et à l'hôte `host`. Si `localAddress` est `0`, l'adresse locale par défaut est utilisée. Si `localPort` est `0`, un numéro de port aléatoire non attribué est utilisé.
    Chaque connexion TCP consiste en deux paires (adresse IP locale, numéro de port local) et (adresse IP distante, numéro de port distant). Lorsqu'une `Socket` est crée et connectée à un hôte distant, on lui alloue usuellement un numéro de port non attribué de manière aléatoire.
    Spécifier le numéro de port local peut être utile pour des serveurs réclamant une connexion à partir d'un numéro de port particulier ou bien en cas de présence d'un "firewall". Spécifier l'adresse locale peut être utile si l'hôte a plusieurs adresses (plusieurs interfaces) et que l'on souhaite en choisir une pour des raisons de performances par exemple.
  - `Socket(InetAddress address, int port, InetAddress localAddress, int localPort) throws IOException` Crée une socket, l'attache à l'adresse `localAddress` et au port `localPort`, puis la connecte au port `port` et à l'hôte d'adresse `address`. Si `localAddress` est `0`, l'adresse locale par défaut est utilisée. Si `localPort` est `0`, un numéro de port aléatoire non attribué est utilisé.

- *Méthodes*
  - Pour communiquer d'un client au travers d'une connexion TCP, il faut créer une `Socket`, puis utiliser `getInputStream()` et `getOutputStream()` pour communiquer avec le serveur distant.
  - `InputStream getInputStream() throws IOException` Cette méthode renvoie un `InputStream`, qui permet une communication par flux le long de la connexion TCP. Pour des raisons d'efficacité, il est louable de "bufferiser" les flux de `Sockets`.
  - `InputStream getOutputStream() throws IOException` Cette méthode renvoie un `InputStream`, qui permet une communication par flux le long de la connexion TCP. Pour des raisons d'efficacité, il est louable de "bufferiser" les flux de `Sockets`. Dans le cas contraire, des transmissions d'un seul octet seront fréquentes ...
  - `void close() thows IOException` Cette méthode ferme l'objet `Socket`, libérant toute ressource système et réseau utilisée. Toute donnée envoyée avant cet appel arrivera à destination, sauf en cas de panne ou congestion réseau. Si l' `OutputStream` de la `Socket` a été "bufferisé", l'objet `BufferedOutputStream` devrait être préalablement fermé, sous peine de perdre les données contenues dans le tampon. Fermer l'objet `Socket`, celui `InputStream` ou celui `OutputStream` fermera la connexion réseau. Pas de possibilité de fermeture partielle de connexion.
  - `InetAddress getInetAddress()` Renvoie l'adresse IP de l'hôte distant.
  - `int getPort()` Renvoie le numéro de port de l'hôte distant à laquelle l'objet `Socket` est connecté.
  - `InetAddress getLocalAddress()` Renvoie l'adresse locale à laquelle l'objet `Socket` est attaché ; c'est l'interface IP locale à travers laquelle les paquets sont envoyés.
  - `int getLocalPort()` Renvoie le numéro de port local auquel l'objet `Socket` est attaché. Cette valeur est assignée aléatoirement si elle n'est pas fournie au constructeur.
  - `void setSocketTimeout(int timeout) throws SocketException` Fixe un temps d'expiration, en milisecondes, au delà duquel une lecture bloquante sur cet objet `Socket` sera automatiquement arrétée. Une valeur égale à 0 rend à nouveau les lectures bloquantes. Dans le cas d'arrêt d'une lecture, une exception `InterruptedIOException` est levée.
  - `int getSocketTimeout(int timeout) throws SocketException` Renvoie le temps d'expiration d'une lecture. Une valeur égale à 0 indique que les lectures sont bloquantes.
  - `void setTcpNoDelay(boolean on) throws SocketException` Permet de déactiver l'utilisation de l'algorithme de Nagle sur une connexion socket. Cet algorithme est utilisé afin de rendre TCP plus efficace en retardant

l'écriture de petites quantités de données jusqu'à ce que soit suffisamment de données aient été "bufferisées" pour qu'un gros paquet puisse être envoyé, soit toutes les données prètes à être envoyées au serveur sont des accusés de réception. Pour des applications normales, l'algorithme de Nagle accroît significativement les parformances, sauf pour certaines applications bien particulières.

- `boolean getTcpNoDelay() throws SocketException` Teste si l'algorithme de Nagle est désactivé. Renvoie `false` par défaut.
- `void setSocketLinger(boolean on, int val) throws SocketException` Permet à un client de fixer un temps d'expiration maximum d'une socket TCP. Le protocole TCP garantit un acheminement sûr des données transmises à une socket. Ceci implique que lorsqu'une socket est fermée, la connexion réseau ne se termine pas immédiatement ; elle reste ouverte le temps que toutes les données soient acheminées avec accusé de réception. Fixer un temps d'expiration signifie que le système d'exploitation n'attendra que la période spécifiée après la fermeture d'une socket avant de fermer la connexion réseau. Si des données n'ont pas été transmises avec succès durant cette période, la connexion réseau est interrompue.
  Lorsqu'une connexion réseau est fermée naturellement, il y a habituellement une période de 4 minutes pendant laquelle une connexion identiqu ne peut être recrée (du même port client au même port serveur). Ceci sert à se protéger contre les duplications involontaires de paquets ; certains paquets peuvent être retardés dans certaines parties du réseau et retransmis par la source. Mettre une temporisation supprimera cette protection. Si une ancienne connexion est interrompue et une nouvelle connexion identique est établie avant que le retard de paquet soit écoulé, le paquet en retard peut arriver et s'insérer dans la conversation. C'est pourquoi, il est préférable, en dehors de cas très particuliers, d'éviter les temps d'expiration de connexion.
- `int getSocketLinger() throws SocketException` Renvoie le temps d'expiration d la connexion en cours ou -1 si l'option est désacivée.
- `void setSendBufferSize(int size) throws SocketException` Cette méthode demande au système d'exploitation de fixer le tampon d'envoi (`SO_SNDBUF`) à `size`. Le système d'exploitation peut ignorer la requête . . .
- `int getSendBufferSize() throws SocketException` Cette méthode renvoie la taille du tampon d'envoi.
- `void setRecieveBufferSize(int size) throws SocketException` Cette méthode demande au système d'exploitation de fixer le tampon de réception (`SO_RCVBUF`) à `size`. Le système d'exploitation peut ignorer la requête . . .
- `int getRecieveBufferSize() throws SocketException` Cette méthode

renvoie la taille du tampon de réception.
- *Exceptions.* Il y a 2 exceptions principales :
  - `SocketException`, fille de `IOException` et superclasse de diverses exceptions communes pour les sockets. Les sous-classes sont :
    - `BindException` Indique qu'un attachement au port ou à l'adresse locale n'a pas pu se faire. Ceci arrive typiquement lorsque le port en question est en cours d'utilisation ou est un port système, ou lorsque l'adresse locale spécifiée n'est pas celle d'une interface réseau valide.
    - `ConnectException` Indique que la connexion a été refusée à cause de l'abscence de serveur en écoute sur le port spécifié de la machine distante.
    - `NoRouteToHostException` Indique que la machine distante n'a pu être atteinte, typiquement à cause d'un problème réseau ou d'un "firewall".
  - `SecurityException`. Le gestionnaire de sécurité (`SecurityManager`) restreint la création des `Sockets`. Une applet non autorisée, par ex., ne peut ouvrir des sockets sur une autre machine que sur celle de laquelle elle a été lancée. Un autre exemple notable est qu'une applet derrière un "firewall" peut ne pas pouvoir faire de recherche (lookup) DNS ; il peut donc être nécessaire d'utiliser des adresses IP numériques.
- *Exemple de client `echo`*
  - Ce client crée un objet `Socket` pour établir la connexion avec le serveur.
  - Le client lit les lignes sur l'entrée standard, les transmet au serveur, lit la réponse du serveur et l'écrit sur la sortie standard. Le client s'arrête lorsqu'il détecte une fin de fichier sur l'entrée standard ou sur le flot d'entrée du serveur.
  - Afin de lire les lignes de texte provenant du serveur, il crée un objet `DataInputStream` à partir de l'objet `inputStream` fourni par l'objet `Socket`. Il crée de même un objet `DataOUtputStream`.
  - Ce client s'exécute avec le nom du serveur sur la ligne de commande et, de manière optionnelle, le numéro de port à utiliser.

```
import java.io.*;
import java.net.*;


public class EchoClient {

    // Variables de classe constantes : port et machine par defaut
    public static final int    DEFAULT_PORT = 6789;
    public static final String DEFAULT_HOST = "localhost";
    public static final int    DEFAULT_LINGER_TIME = 5;
```

```
// Variables d'instance (etat de EchoClient)
String host         = null;  // Machine du serveur
int    port;                 // No de port du serveur (no de service)
int    lingerTime;           // Tps d'expiration de socket apres fermeture
Socket s            = null;  // Reference a la socket client
DataInputStream sin = null;  // Flux en provenance du serveur
PrintStream sout    = null;  // Flux a destination du serveur
DataInputStream in  = null;  // Flux associe au clavier

//-------------------------------------------------------
// METHODE main() : POINT D'ENTREE DU PROGRAMME CLIENT
//-------------------------------------------------------
public static void main(String[] args) {
    // Le client
    EchoClient client = null;

    // Tests d'arguments et appel de constructeurs
    switch(args.length) {
        case 0 : client = new EchoClient();
                 break;
        case 1 : if (args[0].equals("?")) {
                     usage();
                     System.exit(0);
                 }
                 client = new EchoClient(args[0]);
                 break;
        case 2 : client = new EchoClient(args[0], args[1]);
                 break;
        case 3 : client = new EchoClient(args[0], args[1], args[2]);
                 break;
        default : usage();
                 throw new IllegalArgumentException();
    }// switch()

    // Creation d'objets utilitaires et traitement du service
    client.initSocketAndStreams();
    client.service();

}// main()



//---------------
```

```
    // CONSTRUCTEURS
    //---------------
    public EchoClient() {
        host       = new String(DEFAULT_HOST);
        port       = DEFAULT_PORT;
        lingerTime = DEFAULT_LINGER_TIME; // Temps d'expiration
    }

    public EchoClient(String theHost) {
        host = new String(theHost);
        try {
            // Test de validite de l'adresse
            InetAddress dummy = InetAddress.getByName(theHost);
        } catch (UnknownHostException e) {
            System.err.println("Machine " + theHost + " inconnue.");
            System.err.println("Essai avec " + DEFAULT_HOST);
            host = new String(DEFAULT_HOST);
        }
        port       = DEFAULT_PORT;
        lingerTime = DEFAULT_LINGER_TIME;
    }

    public EchoClient(String theHost, String thePortString) {
        this(theHost);
        try {
            port = Integer.parseInt (thePortString);
        } catch (NumberFormatException e) {
            System.err.println("Numero de port invalide. " +
                                DEFAULT_PORT + " Utilise.");
            port = DEFAULT_PORT;
        }
        lingerTime = DEFAULT_LINGER_TIME;
    }

    protected EchoClient(String theHost, String thePortString,
                           String theLingerTimeString) {
        this(theHost, thePortString);
        try {
            lingerTime = Integer.parseInt (theLingerTimeString);
        } catch (NumberFormatException e) {
            lingerTime = DEFAULT_LINGER_TIME;
        }
    }
```

```
//--------------------------------------------
// FONCTIONS D'UTILISATION ET D'INITIALISATION
//--------------------------------------------
static void usage() {
    System.out.println("Utilisation : " +
               "java EchoClient <hostname> [<port>] [<tempsExpir>]");
    System.exit(0);
}

public void initSocketAndStreams()
{
    try {
        // Creer une socket pour se connecter a la machine et au port
        s = new Socket(host, port);
        // Mise du temps d'expiration a la valeur specifiee
        // Voir la mise en garde dans le polycopie en cas reel
        if (lingerTime > 0)
            s.setSoLinger(true, lingerTime);
        else
            s.setSoLinger(false, 0);
        // Creer les flux pour lire et ecrire des lignes de texte
        // de et vers cette socket.
        sin  = new DataInputStream(s.getInputStream());
        sout = new PrintStream(s.getOutputStream());
        // Creer un flux pour lire des lignes de l'entree standard
        //   (par defaut le clavier)
        in   = new DataInputStream(System.in);
        // Signaler que l'on est connecte
        System.out.println("Connecte a " + s.getInetAddress()
                    + ":"+ s.getPort());
    } catch (IOException e) {
        System.err.println(e);
        try { if (s != null) s.close(); } catch (IOException e2) { ; }
    }
}


//---------------------
// TRAITEMENT DU SERVICE
```

```
    //----------------------
    public void service() {
        try {
            String line;
            while(true) {
                // afficher une invite
                System.out.print("EchoClient > ");
                System.out.flush();
                // lire une ligne de stdin; verifier l'EOF
                line = in.readLine();
                if ((line == null) || (line.equals("quit"))) break;
                // L'envoyer au serveur
                sout.println(line);
                // Lire une ligne du serveur
                line = sin.readLine();
                // Verifier si la connexion est fermee (par EOF)
                if (line == null) {
                    System.out.println("Connection fermee par le serveur.");
                    break;
                }
                // Puis ecrire la ligne sur stdout
                System.out.println(line);
            }// while(true)
        } catch (IOException e) { System.err.println(e); }
        // Etre sur de toujours fermer la socket
        finally {
            try { if (s != null) s.close(); } catch (IOException e2) { ; }
        }

    }// service()

}// class EchoClient
```

VI.3 Sockets serveurs

3.1 Classe `ServerSocket`

- Schéma simple (serveur itératif) :
  - Un serveur prend un numéro de port et se met en écoute de connexions ;
  - lorsqu'un client se connecte, le serveur reçoit une objet `Socket` afin de communiquer avec le client ;
  - l'échange peut commencer.

- Schémas plus complexes :
  - Serveurs concurrents : multi-activités (“multi-threaded”).
  - Communications inter-clients *via* le serveur.
  - Sécurisation, authentification.
- *Constructeurs*
  - Les objets `ServerSocket` sont construits en choisissant un numéro de port sur la machine locale (compris entre 1024 et 65 535). Si le numéro de port est 0, le système d'exploitation selectionnera un numéro arbitraire d'un port libre à chaque lancement de l'application. Le nu'méro doit alors être communiqué aux clients par un autre moyen. Un serveur doit explicitement accepter (par la méthode `accept()` une connexion d'un objet `ServerSocket` pour obtenir un objet `Socket` de connexion avec un client. Le système d'exploitation commencera à accepter les connexions dès la création de l'objet `ServerSocket`. Ces connexions seront placées dans une file d'attente et seront retirées une à une à chaque appel d' `accept()`. Un constructeur permet de spécifier combien de connexions il est possible de mettre dans la file d'attente. Si la file est pleine, toute autre connexion est refusée par le système d'exploitation.
  - `ServerSocket(int port) throws IOException` Construit un objet `ServerSocket` en écoute sur le port numéro `port` de la machine locale. La valeur par défaut de connexions pendantes (mises en file d'attente) est de 50. Le système d'exploitation acceptera donc au plus 50 clients dans la file des clients en attente. Toute tentative de connexion au-delà de cette limite sera refusée. Notez bien qu'il ne s'agit pas du nombre maximum de clients que le serveur est capable de traiter simultanément, mais du nombre de connexions en attente dans le cas où le serveur est lent en acceptation de nouvelles connexions.
  - `ServerSocket(int port, int backlog) throws IOException` Construit un objet `ServerSocket` en écoute sur le port numéro `port` de la machine locale. La valeur `backlog` spécifie le nombre de connexions pendantes que le système d'exploitation met en file d'attente.
  - `ServerSocket(int port, int backlog, InetAddress bindAddress) throws IOException` Construit un objet `ServerSocket` en écoute sur le port numéro `port` de la machine locale avec au maximum `backlog` connexions pendantes. Cet objet `ServerSocket` n'acceptera de connexions quà l'adresse locale `bindAddress` ; cci est utile sur une machine “multi-hôte” (à plusieurs interfaces réseaux) afin d'empêcher l'acceptation de connexions à d'autres adresses locales. Une adresse `bindAddress` égale à `null` accepte des connexions à n'importe quelle adresse locale.
- *Méthodes*
  - `Socket accept() throws IOException` Cette méthode bloque (attend)

jusqu'à ce qu'un client se connecte sur le port sur lequel l'objet de type `ServerSocket` est en écoute. Un objet `s` de type `Socket` est renvoyé, correspondant à une connexion TCP avec le client. S'il y a un gestionnaire de sécurité, la méthode `checkAccept()` de ce dernier est appelée avec `s.getInetAddress().getHostAddress()` et `s.getPort()` comme arguments afin de s'assurer que l'opération est permise. Si elle ne l'est pas, une exception `SecurityException` est levée.

- `void close() throws IOException` Cette méthode ferme le point de communication de type `ServerSocket`. Il ne ferme aucune des connexions déjà acceptées et non encore fermées, de sorte qu'un serveur peut fermer ce point de communication et maintenir les connexions existantes avec ses clients. Par contre, cet appel informe le système d'exploitation d'arrêter d'accepter de nouvelles connexions de clients.
- `InetAddress getInetAddress()` Renvoie l'adresse locale à laquelle le point de communication de type `ServerSocket` est attaché. Si aucune adresse locale n'a été spécifiée au constructeur, la valeur renvoyée correspondra à une adresse locale quelconque (typiquement `0.0.0.0`).
- `int getLocalPort()` Renvoie le numéro de port sur lequel l'objet de type `ServerSocket` est en écoute. Ceci est utile si le numéro 0 a été fourni et donc un numéro de port inutilisé a été attribué par le système d'exploitation.
- `void setSocketTimeout(int timeout) throws SocketException` Cette méthode fixe une valeur de temps d'expiration, en milisecondes, après laquelle un appel à `accept()` n'ayant pas reçu de réponse sera interrompu avec une exception de type `InterruptedIOException`. Une valeur de 0 inhibe le compteur d'expiration et un appel à `accept()` bloque indéfiniment.
- `int getSocketTimeout() throws IOException` Renvoie la valeur du temps d'expiration, ou 0 si aucune expiration n'est valide.
- `protected final void implAccept(Socket s) throws IOException` Les sous-classes de `ServerSocket` utilisent cette méthode afin de redéfinir `accept()` et renvoyer leur propre sous classe de `Socket`. Par exemple un objet de type `MaServerSocket` effectuera typiquement un appel du type :

```
MaSocket  ms;
try {
    implAccept(ms);
catch (IOException e) { ... }
```

où `MaSocket` sera une sous-classe de `Socket`. Au retour de `implAccept()`, l'objet de type `MaSocket` sera connecté à un client.

VI.4 Serveurs itératifs

4.1 Exemple de serveur echo

- L'exemple qui suit est un serveur `echo` itératif ("single-threaded"). Il accepte une connexion à la fois, sert le client et une fois que le client a terminé, accepte une autre connexion (d'où le terme itératif). Le service réalisé est le renvoi exact de ce qu'il a reçu.

```
import java.io.*;
import java.net.*;

public class SingleThreadEchoServer {

  public final static int DEFAULT_PORT = 6789;

  public static void main (String[] args) throws IOException {
    int             port;        // no de port du service
    String          line = null;
    ServerSocket    server;
    Socket          client;
    DataInputStream in;
    PrintStream     out;

    // Tests d'arguments
    switch(args.length) {
        case 0 : port = DEFAULT_PORT;
                 break;
        case 1 : try {
                     port = Integer.parseInt (args[0]);
                 } catch (NumberFormatException e) {
                     port = DEFAULT_PORT;
                 }
                 break;
        default :
            throw new IllegalArgumentException ("Syntaxe : STServer [<port>]");
    }

    // Creation de socket serveur
    System.out.println ("Demarrage sur le port " + port);
    server = new ServerSocket (port);
    System.out.println ("En ecoute ...");

    // Boucle generale et service
```

```
    while (true) {
        // Acceptation de connexion
        client = server.accept();
        System.out.println ("Connexion a " + client.getInetAddress() +
                                                          " acceptee");
        System.out.flush();
        // Capture des flux de reception et d'envoi
        in = new DataInputStream(client.getInputStream());
        out = new PrintStream(client.getOutputStream());

        //----------------------------
        // TRAITEMENT DU SERVICE D'ECHO
        //----------------------------
        while (true) {
                // lire une ligne
            line = in.readLine();
            if ((line == null) || (line.equals("exit")))
                break;
            // ecrire la ligne
            out.println(line);
        }
        if (line == null) {
            // Sortie du service : le client a termine
            System.out.println ("Fermeture de socket client");
            client.close ();
        } else if (line.equals("exit")) {  // Fermeture de socket serveur
            System.out.println ("Fermeture des socket client & serveur");
            client.close ();
            server.close ();
            break;
        }
    }// while(true)

  }/* main() */

}/* class STServer */
```

VI.5 Serveurs non bloquants

5.1 Exemple de serveur `echo`

- L'exemple qui suit est un serveur `echo` itératif ("single-threaded") non bloquant, en se servant de la méthode `InputStream.available()` (qui renvoie le nombre d'octets qui peuvent être lus lors d'une prochaine lecture).
- Plusieurs problèmes se posent avec cette approche. Le plus gênant est que, faisant n `accept()` pour recevoir n clients, on doit attendre que les n se soient effectivement connectés ...

```
import java.io.*;
import java.net.*;

public class NBServer {

  static InputStream  in0, in1;
  static OutputStream out0, out1;
  protected int       port = 0;

  public static void main (String[] args) throws IOException {
    // Tests d'arguments
    if (args.length != 1)
      throw new IllegalArgumentException ("Syntaxe : NBServer <port>");

    port = Integer.parseInt(args[0]);
    System.out.println ("Initialisation sur le port " + port);
    ServerSocket server = new ServerSocket (port);

    while(true)
      try {
        accept (Integer.parseInt (args[0]));
        int x0, x1;
        while (((x0 = readNB (in0)) != -1) &&
               ((x1 = readNB (in1)) != -1)) {
          if (x0 >= 0)
            out1.write (x0);
          if (x1 >= 0)
            out0.write (x1);
        }
      } finally {
        System.out.println ("Closing");
        close (out0);
        close (out1);
```

```
    }
  }// main()


  // fermeture de socket
  static void close (OutputStream out) {
    if (out != null) {
      try {
        out.close ();
      } catch (IOException ignored) { ; }
    }
  }// close()


  static void accept (int port) throws IOException {
    try {
      System.out.println ("En ecoute de client 1 ...");
      Socket client0 = server.accept ();
      System.out.println ("Client 1 connecte de " +
                            client0.getInetAddress ());
      in0 = client0.getInputStream ();
      out0 = client0.getOutputStream ();
      out0.write ("Veuillez patienter ...\r\n".getBytes ("latin1"));
      System.out.println ("En ecoute de client 2 ...");
      Socket client1 = server.accept ();
      System.out.println ("Client 2 connecte de " +
                            client1.getInetAddress ());
      in1 = client1.getInputStream ();
      out1 = client1.getOutputStream ();
      out1.write ("Bonjour.\r\n".getBytes ("latin1"));
      out0.write ("Veuillez continuer.\r\n".getBytes ("latin1"));
    } finally {
      server.close ();
    }
  }

  // Fonction de lecture non bloquante
  static int readNB (InputStream in) throws IOException {
    if (in.available () > 0)
      return in.read ();
    else
      return -2;
  }
```

```
}// class NBServer
```

VI.6 Serveurs concurrents

6.1 Exemple de serveur `echo` multi-activités

- Ce serveur lit une ligne de texte envoyée par le client et la lui renvoie sans changement.
- Le serveur crée un objet `ServerSocket` qui surveille le port de connexion.
- Dès qu'une connexion avec un client est acceptée, l'objet `ServerSocket` alloue un nouvel objet `Socket` (connecté à un nouveau port) pour se charger des communications ultérieures avec ce client. Le serveur peut reprendre sa surveillance avec le même objet `ServerSocket` en attente d'autres connexions clientes.
- Ce serveur est multi-activités (multithreaded). La classe `EchoServeur` est implantée par une activité qui a dans sa méthode `run()` une boucle infinie. Cette boucle attend la connexion de nouveaux clients grâce à l'objet `ServerSocket`.
- Pour chaque client, elle crée une nouvelle activité (ici un objet `Connexion` qui se charge des communications avec le client. Ces communications se font à partir de l'objet `Socket` renvoyé par lobjet `ServerSocket`.
- Le constructeur de la classe `Connexion` initialise des flux de communication à partir de l'objet `Socket` et active l'exécution de l'activité.
- La méthode `run()` de `Connexion` gère les communications avec le client ; c'est elle qui fournit effectivement le service, ici un pur et simple renvoi des lignes reçues.

```
import java.io.*;
import java.net.*;

public class EchoServer extends Thread {
    public final static int DEFAULT_PORT = 6789;
    protected int port;
    protected ServerSocket listen_socket;

    // Sortir avec un message d'erreur lorsqu'une exception survient
    public static void fail(Exception e, String msg) {
        System.err.println(msg + ": " +  e);
        System.exit(1);
    }
```

```
    // Creer un ServerSocket pour y ecouter les demandes de connexions;
    //   debuter l'execution de l'activite
    public EchoServer(int port) {
        if (port == 0) port = DEFAULT_PORT;
        this.port = port;
        try { listen_socket = new ServerSocket(port); }
        catch (IOException e) { fail(e,
                                "Exception en creation de socket serveur"); }
        System.out.println("Serveur : en ecoute sur le port no " + port);
        this.start();
    }


    // Corps de l'activite serveur. Effectue une boucle infinie, ecoutant
    // et acceptant les connexions de clients. Pour chaque connexion,
    // creer un objet Connection pour gerer les communications via la
    // nouvelle socket.
    public void run() {
        try {
            while(true) {
                Socket client_socket = listen_socket.accept();
                Connection c = new Connection(client_socket);
            }
        }
        catch (IOException e) {
            fail(e, "Exception en ecoute de connexions");
        }
    }

    // Lancer le serveur, ecoutant sur un port eventuellement specifie
    public static void main(String[] args) {
        int port = 0;
        if (args.length == 1) {
            try { port = Integer.parseInt(args[0]);  }
            catch (NumberFormatException e) { port = 0; }
        }
        new EchoServer(port);
    }
}

// Activite gerant toutes les communications avec un client
class Connection extends Thread {
    protected Socket client;
```

```
    protected DataInputStream in;
    protected PrintStream out;

    // Initialiser les flux et debuter l'activite
    public Connection(Socket client_socket) {
        client = client_socket;
        try {
            in = new DataInputStream(client.getInputStream());
            out = new PrintStream(client.getOutputStream());
        }
        catch (IOException e) {
            try { client.close(); } catch (IOException e2) { ; }
            System.err.println("Exception en obtention de flux de sockets : "
                               + e);
            return;
        }
        this.start();
    }

    // Fournit effectivement le service
    // Lire un ligne et la renvoyer inchangee
    public void run() {
        String line;
        int len;
        try {
            for(;;) {
                // lire une ligne
                line = in.readLine();
                if (line == null) break;
                // ecrire la ligne
                out.println(line);
            }
        }
        catch (IOException e) { ; }
        finally { try {client.close();} catch (IOException e2) {;} }
    }
}
```

VII – Invotaion de méthodes distantes : RMI

Références bibliographiques

Java RMI, W. Grosso [Gro02].

VII.1 Notion d'invocation de méthode distante

1.1 RMI

- Historiquement les réseaux se rencontrent dans 2 types d'applications :
  - le transfert de fichiers, au travers de FTP, SMTP (Simple Mail Transfer Protocol), HTTP (HyperText Transport Protocol), ...
  - Éxécution d'un programme sur une machine distante, au travers de `telnet`, `rlogin`, `RPC`, ...
- *RMI* (Remote Method Invocation) : mécanisme permettant à un programme Java d'*appeler certaines méthodes sur un serveur distant*.
- Peut être vu comme un moyen de définir des protocoles sur mesure et des serveurs avec un minimum d'efforts.

1.2 Transparence du mécanisme

- Chaque objet distant implante une *interface distante* (remote interface) qui **spécifie quelles méthodes peuvent être invoquées** (appelées) par les clients.
- Les clients peuvent appeler des méthodes des objets distants presque exactement comme ils appellent des méthodes locales.
- Les serveurs Web compatibles Java peuvent implanter des méthodes distantes qui permettent au clients d'obtenir un index complet des fichiers du site.

- Ceci peut réduire énormément le temps que prend un serveur à remplir les requêtes de moteurs de recherche (Web spiders) comme Lycos et Altavista. Excite utilise déja une version non Java de cette idée.
- Du point de vue du programmeur, les objets et les méthodes distants fonctionnent comme les locaux. Tous les détails d'implantation sont cachés.
- Les seules choses à réaliser sont :
  - importer un paquetage,
  - aller chercher l'objet dans un registre (une ligne de code),
  - gérer les `RemoteException` lorsque l'on appelle les méthodes.
- Un *objet distant* est un objet dont les méthodes peuvent être appelées par une machine virtuelle Java différence de celle où l'objet vit, généralement celle d'un autre ordinateur.

VII.2 Sécurité – Sérialisation

2.1 Sécurité

- Les activités d'un objet distant sont limitées d'une manière analogue à la limitation des activités d'une applet.
- Un objet `SecurityManager` vérifie qu'une opération est autorisée par le serveur.
- Des gestionnaires de sécurité spécifiques peuvent être définis.
- Une authentification par clé publique peut être utilisée pour vérifier l'identité d'un utilisateur et autoriser à différents utilisateurs différents niveaux d'accès à un objet distant.
- Par exemple, le grand public peut être autorisé à consulter une base de données, mais pas à la modifier. Le personnel interne d'une entreprise peut être autorisé à consulter et à mettre à jour la base.

2.2 Sérialisation

- Lorsqu'un objet est passé ou renvoyé d'une méthode Java, ce qui est réllement transféré est une référence à l'objet.
- Dans les implantations actuelles de Java, les références sont des pointeurs à double indirection ; ils pointent sur les zones mémoire de stockage des objets de la machine virtuelle.
- Passer des objets d'une machine virtuelle à une autre pose quelques problèmes.
- 2 solutions possibles :

  – on passe une référence distante (qui pointe sur des zones mémoires de la machine distante),
  – on passe une copie de l'objet.
- Lorsque la machine locale passe un objet distant à la machine distante, elle passe en fait une référence distante. L'objet n'a en réalité pas quitté la machine distante.
- Lorsque la machine locale passe l'un de ses propres objets à la machine distante, elle réalise une copie de l'objet qu'elle envoie.
- Pour copier un objet, il faut un moyen de convertir cet objet en un flux d'octets.
- La *sérialisation* est un schéma qui convertit les objets en un flux d'octets qui peut être envoyé à d'autres machines. Ces dernières reconstruisent l'objet original à partir des octets.
- Ces octets peuvent également être écrits sur disque et lus ultérieurement ; on peut donc sauvegarder l'état d'un objet individuel.
- Pour des raisons de sécurité, **seulement certains objets peuvent être sérialisés**.
- Tous les types primitifs ainsi que les objets distants peuvent être sérialisés, mais les objets locaux ne peuvent l'être que s'ils implantent l'interface `Serializable`.
- Liste des classes sérialisables :

Sérialisable	Non sérialisable
`java.lang.Character`	`Thread`
`java.lang.Boolean`	`InputStream et classes filles`
`java.lang.String`	`OutputStream et classes filles`
`java.lang.Throwable`	`JDBC ResultSet`
`java.lang.Number`	`...`
`java.lang.StringBuffer`	
`java.util.Hashtable`	
`java.util.Random`	
`java.util.Vector`	
`java.util.Date`	
`java.util.BitSet`	
`java.io.File`	
`java.net.InetAdress`	
`java.net.URL`	
`java.awt.BorderLayout`	
`java.awt.Color`	
`java.awt.Dimension`	
`java.awt.Event`	
`java.awt.Font`	

java.awt.Polygon java.awt.CardLayout java.awt.FontMetrics java.awt.Image java.awt.Window java.awt.FlowLayout java.awt.GridLayout java.awt.Point java.awt.Rectangle java.awt.MenuComponent java.awt.Insets java.awt.CheckboxGroup java.awt.MediaTracker java.awt.GridBagLayout java.awt.GridBagConstraints java.awt.Cursor java.rmi.server.RemoteObject	

VII.3 Brève description du fonctionnement des RMI

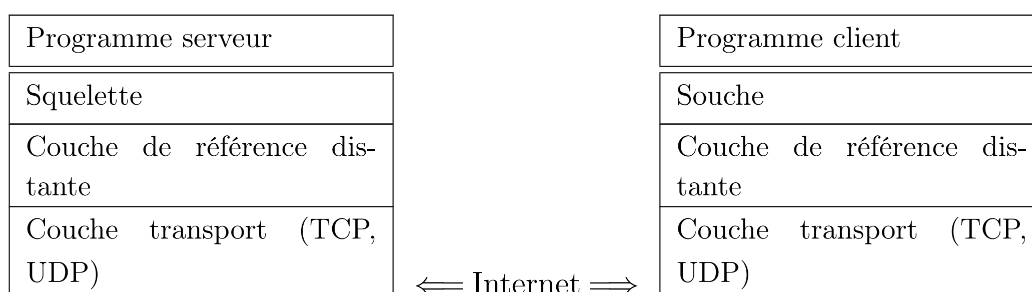
3.1 Passage d'arguments

- 3 mécanismes différents sont utilisés pour passer des arguments et en récupérer d'une méthode distante, selon le type de donnée passée :
  - Les **types primitifs** (`int`, `double`, `boolean`, ...) sont passés par valeur, de la même manière qu'en appel de méthode locale Java.
  - Les **références à des objets distants** (c.à.d. des objets implantant l'interface `Remote`) sont passés comme références distantes, qui permettent au récepteur d'appeler les méthodes de l'objet. Ceci est similaire à la manière dont les références d'objets locaux sont passés à des méthodes locales Java.
  - Les **objets qui n'implantent pas l'interface** `Remote` sont passés par valeur. C'est-à-dire que des copies complètes sont passées, en utilisant la sérialisation.
- Des objets qui ne sont pas sérialisables **ne peuvent pas être passés à des méthodes distantes**.

- Des méthodes d'objets distants s'exécutent sur le serveur mais peuvent être appelées par des objets au sein du client. Des méthodes d'objets non distants et sérialisables s'exécutent sur le client.

3.2 Communications

- Les communications se réalisent par une série de couches comme suit



- En fait le programme client **converse avec une souche** (stubs) et le programme serveur **converse avec un squelette** (skeleton).
- Un client trouve un serveur approprié en utilisant un registre, qui possède une méthode `lookup()`.
- Lorsqu'on appelle `lookup()`, cette dernière trouve la souche nécessaire pour l'objet désiré, et le charge. Ce squelette peut provenir de n'importe quelle plate-forme.
- Un client appelle une méthode distante en utilisant une souche.
- L'*souche* (stub) est un objet spécial qui implante les interfaces distantes de l'objet distant.
- ⇒ la souche possède des **méthodes ayant les mêmes signatures** que toutes les méthodes que l'objet distant exporte.
- De fait, le client pense qu'il appelle une méthode de l'objet distant, mais en réalité il appelle une méthode équivalente de la souche.
- Les souches sont utilisés dans la machine virtuelle du client à la place des objets réels et de leurs méthodes qui résident au sein du serveur. Lorsque la souche est appelée, elle passe l'appel à la couche de référence distante (Remote Reference Layer).
- La couche de référence distante contient un protocole de référence distante indépendant des souche client et squelette serveur. Cette couche de référence distante est responsable de l'interprétation des références distantes.
- Plus précisément, les références locales de la souche client sont converties en références distantes à l'objet sur le serveur. Puis, les données sont passées à la couche transport.

3.3 Registres

- Dans un mécanisme d'appel de procédures distantes (RPC) tel qu'on le trouve sous Unix, il y a un gestionnaire de numéros de ports (port mapper) qui maintient une table de paires (nom de service distant, numéro de port).
- Un processus client voulant effectuer un appel de procédure distant s'adresse au gestionnaire de ports, qui lui renvoie le numéro de port correspondant au service (à la procédure distante) demandé. Le processus client peut alors s'y connecter et la procédure distante s'effectue.
- Le client n'a besoin de connître que le numéro de port du gestionnaire de ports et le nom du service (de la procédure distante).
- Le mécanisme de RMI, de manière similaire aux RPC, utilise un service analogue au gestionnaire de port, nommé un (gestionnaire de) registre (registry).
- Un registre est généralement lancé sur la même machine que le serveur. **À un gestionnaire de regsitre correspond une unique paire numéro de de machine, numéro de port et réciproquement**.
- Chaque gestionnaire de registre implante l'interface `Registry` et fournit donc des services :
  - d'enregistrement d'objet (dont des méthodes seront disponibles de manière distante),
  - d'obtention de référemce distante à un objet,
  - de listage des objets enregistrés,
  - d'annulation d'enregistrement d'un objet.

VII.4 Implantation

4.1 Aperçu des paquetages

- 4 paquetages pour les rmi :
  - `java.rmi`,
  - `java.rmi.server`,
  - `java.rmi.registry`,
  - `java.rmi.dgc`
- `java.rmi` : classes, interfaces et exceptions coté client.
- `java.rmi.server` : classes, interfaces et exceptions coté serveur. Classes utilisées lorsqu'on écrit des objets distants utilisés par des clients.
- `java.rmi.registry` : classes, interfaces et exceptions utilisées pour localiser et nommer des objets distants.
- `java.rmi.dgc` : gestion du ramasse-miettes distribué (dgc, ou Distributed Grabage Collector).

- ☛ Dans la suite, et conformément à la documentation Sun, on utilisera l'adjectif *distant* en se référant implicitement à un serveur et l'adjectif *local* en se référant à un client.

4.2 Implantation du serveur

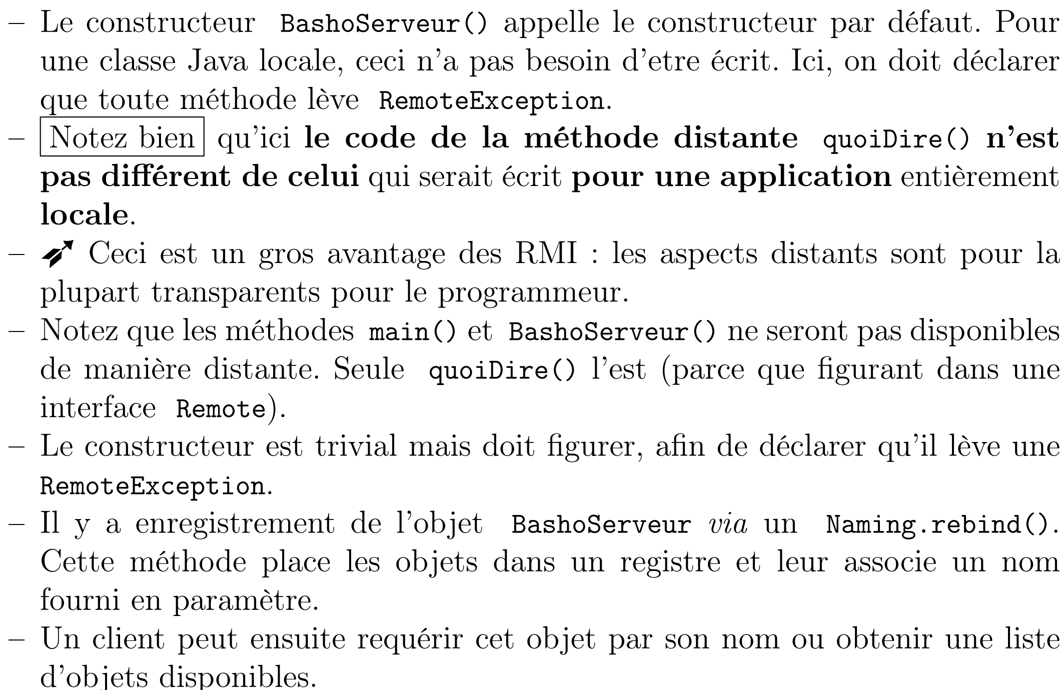
- Pour créer un nouvel objet distant, on **définit une interface qui étend** `java.rmi.Remote`.
- L'interface `Remote` n'a aucune méthode. Elle agit comme un marqueur d'objets distants.
- L'interface enfant de `Remote` que l'on définit détermine quelles méthodes de l'objet distant à créer peuvent être appelées par les clients.
- **Seules les méthodes** publiques **déclarées** dans une **interface distante** peuvent être **appelées de manière distante**.
- **Chaque méthode de l'interface enfant** de `Remote` définie par le programmeur **doit déclarer lever** l'exception `java.rmi.RemoteException`.
- Exemple d'une interface pour un "Hello World" distant. Cette interface a une seule méthode, `parler()`

```
import java.rmi.*;

public interface DoueDeParole extends Remote {
    public String parler() throws java.rmi.RemoteException;
}
```

- Après avoir défini une interface distante, il faut **définir une classe** qui **implante cette interface et étend** `java.rmi.UnicastRemoteObject`.
- La classe `UnicastRemoteObject` fournit un certain nombre de méthodes qui se chargent des détails de la communication distante.
- En particulier, il y a le découpage et le rassemblement.
- *Découpage* (marshaling) : conservion des arguemnts et des valeurs de retour en un flux d'octets qui peuvent être envoyés à travers le réseau.
- Exemple de `BashoServeur`, une classe qui implante l'interface distante `DoueDeParole` et qui étend `UnicastRemoteObject`

```
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;

public class BashoServeur extends UnicastRemoteObject
                                  implements DoueDeParole {

    public BashoServeur() throws RemoteException {
        super();
```

```
    }

    public String parle() throws RemoteException {
        return "La cloche du temple s'est tue      " + "\n"
               "Dans le soir, le parfum des fleurs " + "\n"
               "En prolonge le tintement.          " + "\n"
               "Matsuo Basho (1644-1694)" ;
    }

    public static void main(String args[]) {
        try {
            BashoServeur b = new BashoServeur();
            Naming.rebind("MatsuoBasho", b);
            System.out.println("Serveur BahoServeur pret.");
        } catch (RemoteException re) {
            System.out.println("Exception ds BashoServeur.main() : " + re);
        } catch (MalformedURLException re) {
            System.out.println("MalformedURLException ds BashoServeur.main() : "
                                    + re);
        }
    }// main()
}
```



- Le constructeur `BashoServeur()` appelle le constructeur par défaut. Pour une classe Java locale, ceci n'a pas besoin d'etre écrit. Ici, on doit déclarer que toute méthode lève `RemoteException`.
- Notez bien qu'ici **le code de la méthode distante** `quoiDire()` **n'est pas différent de celui** qui serait écrit **pour une application** entièrement **locale**.
- Ceci est un gros avantage des RMI : les aspects distants sont pour la plupart transparents pour le programmeur.
- Notez que les méthodes `main()` et `BashoServeur()` ne seront pas disponibles de manière distante. Seule `quoiDire()` l'est (parce que figurant dans une interface `Remote`).
- Le constructeur est trivial mais doit figurer, afin de déclarer qu'il lève une `RemoteException`.
- Il y a enregistrement de l'objet `BashoServeur` *via* un `Naming.rebind()`. Cette méthode place les objets dans un registre et leur associe un nom fourni en paramètre.
- Un client peut ensuite requérir cet objet par son nom ou obtenir une liste d'objets disponibles.

4.3 Génération des souche et squelette

- On génère la souche (stub) et le squelette (skeleton) *via* `rmic`, que l'on applique aux `.class` Par ex.

```
/home/mounier > rmic BashoServeur
/home/mounier > ls Doue* BashoServeur*
BashoServeur.class    BashoServeur_skel.class    DoueDeParole.class
BashoServeur.java     BashoServeur_stub.class    DoueDeParole.java
```

- Arguments : `rmic BashoServeur -d -classpath JavaProgs/Tests` L'option `-d` va mettre le .class dans le répertoire où le source est trouvé (par défaut, le `.class` est placé dans le répertoire courant).

4.4 Lancement du serveur

- Pour lancer le serveur :
  - On lance d'abord le serveur de registre, avec lequel le serveur dialogue : `rmiregistry &` (ou `start rmregistry` sous DOS). Il **écoute sur le port 1099 par défaut**. Pour qu'il écoute sur un autre port : `rmiregistry 2048 &`
  - On lance ensuite le serveur,

```
/home/mounier > java BashoServeur &
Serveur BashoServeur pret.
```

4.5 Implantation du client

- Pour qu'un client appelle une méthode distante, il doit récupérer une référence à un objet distant.
- Pour cela, il récupère un objet distant en demandant au serveur de rechercher dans un registre. La demande s'effectue au travers de la méthode `lookup(String name)` qu'appelle le client.
- Le schéma de nommage dépend du registre utilisé. La classe `Naming` du paquetage `java.rmi` fournit un schéma fondé sur des URL pour localiser des objets.
- L'obtention de référence distante se fera alors par :

```
Object o1 = Naming.lookup("rmi://sunsite.unc.edu/MatsuoBasho");
Object o2 = Naming.lookup("rmi://sunsite.unc.edu:2048/MatsuoBasho");
```

- Le champ de protocole est ici `rmi`, qui signifie que l'URL référence un objet distant.
- Le champ fichier (ici `MatsuoBasho`) spécifie le nom de l'objet distant.
- Les champs de nom de machine et de numéro de port optionnel sont inchangés par rapport au cas d'HTTP.

– Puisque la méthode `lookup()` renvoie un `Object`, il faut effectuer une conversion de type (cast) en le type d'interface distante que l'objet distant implante (et non la classe elle-même, cachée des clients) : `DoueDeParole b = (DoueDeParole) Naming.lookup("MatsuoBasho") ;`
– Exemple de client associé à `BashoServeur` :

```
import java.rmi.*;

public class BashoClient {
    public static void main(String args[]) {
        System.setSecurityManager(new RMISecurityManager());
        try {
            DoueDeParole b = (DoueDeParole) Naming.lookup("MatsuoBasho");
            String message = b.parle();
            System.out.println("BashoClient : " + message);
        } catch (Exception e) {
            System.out.println("Exception dans BashoClient.main() : "
                               + e);
        }
    }
}
```

VII.5 Paquetage java.rmi

5.1 L'interface `Remote`

– Le paquetage `java.rmi` contient tout ce dont a besoin un client. Il contient 1 interface, 2 classes et une poignée d'exceptions.
– L'interface `Remote` ne déclare aucune méthode. Elle sert de marqueur pour les objets distants.
– On implante une interface sous classe de `Remote`.

5.2 La classe `Naming`

– `java.rmi.Naming` est un gestionnaire de registre qui associe à un URL de la forme `rmi ://machine.domaine.surdomaine/monObjetDistant` un objet distant.
– Chaque entrée du registre a un nom et une référence d'objet. Les clients donnent le nom, et récupèrent une référence à un objet distant.
– `Naming` est le seul type de registre actuellement disponible avec les RMI.
– Son plus gros défaut est que le client doit connaître le serveur où réside l'objet distant. En outre, le schéma de nommage n'est pas hiérarchisé.

– La classe `Naming` a 5 méthodes :

`methode()`	But
`public static Remote lookup(String url) throws RemoteException, NotBoundException, AccessException, UnknownHostException`	Renvoie une souche (stub) associé à l'objet distant référencé par l'URL `url`. Cette souche agit pour le client comme l'objet distant. `RemoteException` est levée si `lookup()` n'arrive pas à trouver le registre spécifié (souvent à la suite d'un "connection time out"). `NotBoundException` est levée si le nom spécifié dans `url` n'est pas répertorié dans le registre utilisé. `AccessException` est levée si le client n'est pas autorisé à se connecter à ce registre. `UnknownHostException` est levée si la machine référencée dans `url` ne peut être localisée.
`public static void bind(String url, Remote ro) throws RemoteException, AlreadyBoundException, AccessException, UnknownHostException`	Un serveur utilise `bind()` pour associer un nom comme MatsuoBasho à un objet distant. Si `url` est déja associée à un objet local, `bind()` lève une `AlreadyBoundException`.
`public static void unbind(String url, Remote ro) throws RemoteException, NotBoundException, AccessException, UnknownHostException`	Enlève l'objet sécifié par `url` du registre (effectue l'opposé de `bind()`).
`public static void rebind(String url, Remote ro) throws RemoteException, AccessException, UnknownHostException`	Travail similaire à celui de `bind()`, sauf que l'association s'effectue même si `url` était déja associée à un objet distant. Dans ce cas, l'ancienne association est perdue.
`public static String[] list(String url) throws RemoteException, AccessException, UnknownHostException`	Renvoie tous les URL qui sont associés, sous forme de tableau de chaînes. On doit fournir le nom du registre à la fin de `url`.

5.3 La classe `RMISecurityManager`

– Les souches sont en un sens comme des Applets, du point de vue de la sécurité. Ils transportent du code d'une autre machine qui vient s'exécuter

localement.
- La machine virtuelle Java ne laisse s'exécuter un tel code que s'il y a un gestionnaire de sécurité en place.
- Celui par défaut avec les RMI `RMISecurityManager`, ne permet qu'un accès minimum à des objets distants.
- Un seul constructeur, sans argument. Utilisation usuelle :
`System.setSecurityManager(new RMISecurityManager());`
- La méthode `getSecurityContext()` détermine l'environnement pour l'accès à certaines opérations. Plus d'opérations sont permises par une classe chargée localement qu'une chargée de manière distante.
- À l'avenir, une authentification par signature à clé publique permettra de donner à différents utilisateurs différents niveaux d'accès.
- Il y a 23 méthodes destinées à tester diverses opérations afin de voir si elles sont permises. Chacune lève une `StubSecurityException` si l'opération est interdite. Sinon, la méthode se termine et ne renvoie rien.
- Liste des méthodes sécuritaires :

`methode()`	But	Souche locale	Souche distante
`checkCreateClassLoader()`	Une souche peut-elle créer un `ClassLoader` ?	Non	Non
`checkAccess(Thread t)` `checkAccess(ThreadGroup g)`	Un souche peut-elle manipuler des activités en dehors de son propre groupe d'activités ?	Non	Non
`checkExit(int status)`	Un souche peut-elle appeler `System.exit()` ?	Non	Non
`checkExec(String cmd)`	Une souche peut-elle appeler `System.exec()` ?	Non	Non
`checkLink(String lib)`	Une souche peut-elle être lié à une bibliothèque dynamique ?	Non	Non
`checkPropertiesAccess()`	Une souche peut-elle lire les propriétés (variables d'environnement) de la machine locale ?	Non	Non
`checkPropertyAccess(String key)`	Une souche peut-elle v'erifier une propriété doonnée	Non	Non
`checkRead(String file)`	Une souche peut-elle lire un fichier ?	Non	Non
`checkRead(String file, URL base)`		Non	Non
`checkRead(String file, Object context)`		Non	Non
`checkRead(FileDescriptor fd)`		Non	Non
`checkWrite(String file)`	Une souche peut-elle écrire sur un fichier ?	Non	Non

checkWrite(FileDescriptor fd)		Non	Non
checkListen(int port)	Une souche peut-elle écouter les demandes de connexion sur un port ?	Non	Non
checkAccept(String host, int port)	Une souche peut-elle accepter les demandes de connexions sur un port ?	Non	Non
checkConnect(String host, int port)	Une souche peut-elle ouvrir une connexion à host sur port ?	Non	Oui, si la machine est l'une à partir desquelles la souche a été chargée ; sinon, non.
checkConnect(String host, int port, Object context)		Non	Non
checkConnect(String fromHost, String toHost)		Non	Non
checkTopLevelWindow(Object Window)	La Souche peut-elle créer une nouvelle fenêtre ?	Non	Non
checkPackageAccess(String pkg)	Une souche peut-elle créer le paquetage pkg ?	Non (pourra être ajustable par l'utilisateur dans le futur)	Non (pourra être ajustable par l'utilisateur dans le futur)
checkPackageDefinition(String pkg)	Une souche peut-elle créer des classes au sein du paquetage pkg ?	Non (pourra être ajustable par l'utilisateur dans le futur)	Non (pourra être ajustable par l'utilisateur dans le futur)
checkSetFactory()	Une souche peut-elle monter une usine distante (network factory) ?	Non	Non

- Il y a également 17 exceptions qui sont définies dans `java.rmi`. Un programme réseau peut avoir beaoucoup de raisons de ne pas fonctionner correctement. ⇒ Pour avoir des programmes relativement fiables, il est nécessaire de gérer (par `catch`) les diverses exceptions susceptibles d'être levées par les méthodes utilisées.
- En 1 ière approximation, on peut se contenter de ne gérer que `RemoteException`, dont les 16 autres dérivent.

VII.6 Paquetage java.rmi.registry

6.1 Interface `Registry`

- 1 interface : `Registry`, qui spécifie le comportement d'un gestionnaire de registre.
- L'interface `Registry` a un champ de données : `public final static int REGISTRY_PORT` qui contient le numéro de port associé au gestionnaire de registre. La valeur par défaut est 1099.
- Elle spécifie les méthodes :
  - `bind()`,
  - `list()`,
  - `lookup()`,
  - `rebind()`,
  - `unbind()`

  qui ont été détaillées en voyant la classe `Naming`.

6.2 Classe `LocateRegistry`

- 1 classe : `LocateRegistry`, qui sert à localiser des gestionnaires de registres (obtenir une référence en donnant un n° de port) et à en créer d'autres.
- Toutes les méthodes sont `static` et il n'y a pas de constructeur.
- Les méthodes sont les suivantes :

methode()	But
`public static Registry createRegistry(int port) throws RemoteException`	Crée un gestionnaire de registre associé au numéro de port `port`. l'utilisateur doit avoir le droit d'associer un service au numéro de port spécifié (sous Unix, il faut être `root` pour pouvoir fournir un numéro de port inférieur à 1024). Une exception `RemoteException` est levée en cas d'échec.
`public static Registry getRegistry() throws RemoteException`	Renvoie une référence au gestionnaire de regsitre situé sur la machine locale et associé au numéro de port 1099. Une exception `RemoteException` est levée en cas d'échec.
`public static Registry getRegistry(int port) throws RemoteException`	Renvoie une référence au gestionnaire de regsitre situé sur la machine locale et associé au numéro de port `port`. Une exception `RemoteException` est levée en cas d'échec.

`public static Registry getRegistry(String host) throws RemoteException, UnkownHostException`	Renvoie une référence au gestionnaire de regsitre situé sur la machine `host` et associé au numéro de port 1099. Si la localisation de `host` échoue, une exception `UnkwonHostException` est levée. Toute autre cause d'échec génère une `RemoteException`.
`public static Registry getRegistry(String host, int port) throws RemoteException, UnkownHostException`	Renvoie une référence au gestionnaire de regsitre situé sur la machine `host` et associé au numéro de port `port`. Si la localisation de `host` échoue, une exception `UnkwonHostException` est levée. Toute autre cause d'échec génère une `RemoteException`.

VII.7 Paquetage java.rmi.server

7.1 Rapide description générale

- Ce paquetage contient 6 exceptions, 7 interfaces et 10 classes. Il ne doit être en général qu'utilisé par les serveurs.
- Les classes importantes sont :
  - `RemoteObject`, la classe de base pour tous les objets distants,
  - `RemoteServer`, classe abstraite qui étend `RemoteObject` et spécifie un serveur RMI.
  - `UnicastRemoteObject`, qui implante concrètement `RemoteServer`.

7.2 Classe `RemoteObject`

- La classe `RemoteObject` joue le rôle d' `Object` pour des objets distants.
- Elle fournit une implantation des méthodes suivantes :
  - `toString()`, qui renvoie une description en `String` d'un objet distant. Elle renvoie dans cette chaîne le nom de machine, le numéro de port du gestionnaire de registre et un numéro de référence de l'objet.
  - `hashCode()`,
  - `clone()` et
  - `equals()`, qui compare 2 références distantes et renvoie `true` si elles pointent sur le même objet.

7.3 Classe `RemoteServer`

- Cette classe étend `RemoteObject`. C'est une classe abstraite donnant lieu à des implantations concrètes, comme `UnicastRemoteObject` (la seule implantation disponible à l'heure actuelle).
- Constructeurs : `protected RemoteServer()` `protected RemoteServer(RemoteRef r)` On ne les instanciera pas. On créera des instances de `UnicastRemoteObject`.
- `RemoteServer` a 2 méthodes pour obtenir des informations sur le client avec lequel on communique :
  - `public static String getClientHost() throws ServerNotActiveException` qui renvoie une `String` contenant le nom de la machine du client qui a invoqué la méthode contenant l'appel `getClientHost()` ;
  - `public int getClientPort() throws ServerNotActiveException` qui renvoie une `String` contenant le numéro de port du client qui a invoqué la méthode contenant l'appel `getClientPort()`. Rappelons que les numéros de ports client permettent au serveur d'identifier les différents clients.

  Chacune de ces méthodes peut lever une `ServerNotActiveException` si l'activité (thread) courante n'exécute pas une méthode distante.
- Pour le debugging, il est parfois utile de voir les appels effectués à des objets distants ainsi que leurs réponses. On peut enregistrer ces informations dans un fichier log *via* : `public static void setLog(OutputStream os)`. Si `os` est égal à `null`, l'enregistrement est stoppé.
- On peut obtenir une référence au flux d'enregistrement par `public static void getLog(OutputStream os)`. Ainsi par exemple, on peut ajouter ses propres informations :

```
PrintStream p = RemoteServer.getLog();
p.println("Il y a eu " + n + " appels a l'objet distant");
```

7.4 Classe `UnicastRemoteObject`

- On définira en général une sous classe de `UnicastRemoteObject`.
- 2 méthodes :
  - `public Object clone() throws CloneNotSupportedException` qui crée un clone de l'objet distant appelant.
  - `public static RemoteStub exportObject() throws RemoteException` On utilise cette méthode pour exporter des objets qui ne sont pas des sous classes de `UnicastRemoteObject`. L'objet doit alors implanter l'interface `Remote` et s'exporter lui-même avec `exportObject()`.

Bibliographie

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, Redwood City, CA, 1994. éd. française : *Analyse & conception orientées objets*, Addison–Wesley France, Paris, 1994. Un classique de l'ingéniérie du logiciel.

[Bud98] T. Budd. *Understanding Object-Oriented Programming with JAVA.* Addison-Wesley, Reading, MA, 1998. `http ://www.awl.com/cseng`. Excellent livre. Explique le "pourquoi" (vue compréhensive du langage) de la programmation java et pas seulement le "comment" (simple vue descriptive).

[GHJV95] E. Gamma, R. Helm, R. Johnson, et J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995. Livre fondateur d'une technique maintenant classique en génie logiciel.

[GJS96] J. Gosling, B. Joy, et G. Steele. *The Java Language Specification.* Addison-Wesley, Reading, MA, 1996. Description très détaillée du langage par leurs fondateurs ; assez aride.

[Gro02] W. Grosso. *Java RMI.* O'Reilly, Beijing, 2002. Un excellent livre de programmation et conception de serveurs en RMI.

[Har97] E.R. Harold. *Java Network Programming.* O'Reilly, Cambridge, 1997. Un fort bon livre sur les threads en Java.

[Lea97] D. Lea. *Concurrent Programming in Java. Design Principles and Patterns.* Addison-Wesley, Reading, MA, 1997. Un des meilleurs, sinon le meilleur, livre de programmation concurrente en Java.

[OW97] S. Oaks et H. Wong. *Java Threads.* O'Reilly, Cambridge, 1997. Un très bon livre sur les threads en Java.

[RDBF02] G. Roussel, E. Duris, N. Bedon, et R. Forax. *Java et Internet – Concepts et programmation.* Vuibert, Paris, 2002. Tome 1 – Côté

client. Un excellent livre de programmation réseau. Très bonne introduction à diverse aspects du langage hors réseau.

[Ven99] B. Venners. Design techniques. articles about java program design, 1998–1999. Divers articles excellents sur le design en Java. À lire absolument.

[Wei98] M. A. Weiss. *Data Stuctures & Problem Solving Using Java.* Addison-Wesley, Reading, MA, 1998. `http ://www.awl.com/cseng/titles/0-201-54991-3`. Bon livre sur un sujet ultra-classique.