

TP 1 : Bases de Java

Le projet de programmation Java vise à faire acquérir une expérience de programmation d'un logiciel de taille conséquente. Les difficultés principales du projet ne sont pas algorithmiques mais touchent à l'organisation du développement : architecture du code, répartition des tâches dans un groupe, documentation, gestion des versions, tests, déploiement, ...

Le projet est précédé par une série de travaux pratiques pour découvrir le langage Java, quelques méthodes pour la programmation par objets, des outils d'aide au développement, et quelques-unes des bibliothèques fournies avec Java. Plutôt qu'un véritable cours, ces TP vous permettront d'apprendre à partir d'exemples.

On trouvera de nombreux pointeurs depuis la page web pour cet enseignement à l'adresse http://www.lsv.ens-cachan.fr/~schmitz/teach/2008_java/.

1 Java

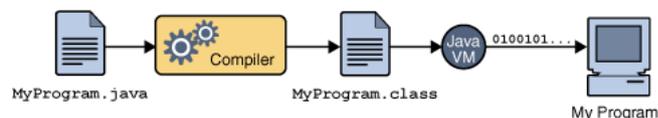
Le langage Java est un langage généraliste développé par Sun Microsystems depuis 1991. Il emprunte une bonne part de sa syntaxe à C++ mais est sensiblement plus simple. Depuis 2007, les outils qui constituent l'environnement de développement (*Java Development Kit*, JDK) sont distribués sous license GPL.

Le langage est conçu pour encourager une programmation simple (et un peu verbeuse). Parmi ses autres attraits, Java bénéficie de l'existence de très nombreuses bibliothèques, d'une large base d'utilisateurs, et de nombreux outils d'aide au développement.

Le contenu de cette section reprend de manière très condensée des informations prises dans le tutoriel Java disponible depuis la page <http://java.sun.com/docs/books/tutorial/java/>.

1.1 Machine virtuelle

Au lieu d'une compilation directe d'un programme source dans du code exécutable natif pour une plate-forme donnée, les programmes Java sont habituellement compilés dans du code intermédiaire, appelé *bytecode*, qui est interprété par une machine virtuelle (*Java Virtual Machine*, JVM) implémentée sur chaque plate-forme.

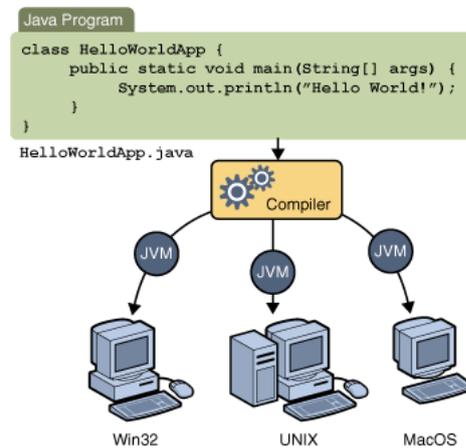


Ainsi, la compilation d'un programme est effectuée par la commande

```
javac MyProgram.java
```

et résulte en la création d'un fichier `MyProgram.class` en bytecode Java, qui pourra ensuite être interprété par la machine virtuelle. À supposer que le programme contienne un `main`, la commande à utiliser pour cela est alors

```
java MyProgram
```



Cette étape intermédiaire permet de distribuer des fichiers déjà compilés en bytecode et de les interpréter sur différentes architectures. Un des slogans de Sun pour Java était du coup « *Write once, run everywhere* », même s'il n'existe pas forcément d'implémentation de la JVM sur toutes les architectures...

Un impact de cette architecture de l'exécution de programmes Java est le temps et surtout l'espace mémoire utilisé par les implémentation de la JVM. Les performances à l'exécution de programmes Java sont sinon très acceptables, grâce en particulier à des mécanismes de compilation au vol de bytecode en code natif (*Just in Time*, JIT).

Implémentations de la JVM Il a fallu de nombreuses années avant que Java ne devienne un langage libre, et grâce à cela il existe un compilateur natif de code Java dans `gcc` : `gcj`. Il n'est malheureusement pas parfaitement à jour avec les évolutions du langage, mais a le mérite de permettre la compilation native. Il existe par ailleurs des implémentations de la JVM par IBM, développées pour Eclipse. Des versions allégées du langage sont utilisées sur des plates-formes mobiles (téléphones, consoles), avec des JVM elles aussi allégées.

1.2 Programmation impérative

Même si cet aspect n'est pas central en Java, sa syntaxe est tout de même constituée d'instructions et d'expressions pour une programmation impérative. Cette section pourra vous servir de référence très succincte sur le socle impératif de Java.

1.2.1 Types primitifs

Tout, ou presque, est un objet en Java. Ce qui n'est pas un objet appartient forcément à l'un des types primitifs suivants :

entiers les types `byte`, `short`, `int` et `long`,

réels les types `float` et `double`,

booléens le type `boolean`, qui peut avoir les valeurs `true` ou `false`,

caractères le type `char`, qui représente des caractères UTF-16.

Exemple 1.1.

```
byte b = 100;
short s = 10000;
int i = 100000;
int decVal = 26; // The number 26, in decimal
int octVal = 032; // The number 26, in octal
int hexVal = 0x1a; // The number 26, in hexadecimal
double d1 = 123.4;
double d2 = 1.234e2; // same value as d1, but in scientific notation
float f1 = 123.4f;
boolean result = true;
char capitalC = 'C';
char capitalCcirc = '\u0108';
char tab = '\t';
```

Tous ces types ont une classe associée (`Byte`, `Short`, etc.) pour permettre de les voir comme des objets. Le compilateur Java est capable de faire des conversions implicites entre ces objets et leurs valeurs primitives (*boxing/unboxing*).

Les arguments des méthodes Java sont passés par valeur s'ils sont d'un type primitif, et par référence s'ils sont d'un type objet.

1.2.2 Tableaux

Les tableaux Java sont semblables à des tableaux C : ce sont des espaces mémoire consécutifs de taille fixe, et permettent un adressage direct d'un élément par son index (un entier). Un tableau est déjà un objet, que l'on crée via `new`, que l'on manipule comme une référence, et qui peut avoir la valeur `null`. L'espace mémoire utilisé par les objets n'a pas besoin d'être libéré explicitement : la JVM utilise un ramasse-miettes (*garbage collector*).

Exemple 1.2.

```
int[] anArray; // declares an array of integers
String[] anotherArray; // declares an array of strings
anArray = new int[10]; // allocates memory for 10 integers
int[] numbers = {1,2,3,4,5,6,7,8,9,10}; // initialized array
```

```
anArray[0] = 100; // initialize first element
anArray[1] = 200; // initialize second element

System.out.println(anArray.length); // print the length (10)
```

1.2.3 Expressions

Java fournit les opérations suivantes :

arithmétiques pré/post incrément/décément (`i++`, `--j`), moins unaire (`-`), opérations binaires (`+`, `-`, `*`, `/`, `%`),

relationnelles `<`, `>`, `<=`, `>=`, `==`, `!=`,

booléennes ou (`||`), et (`&&`), non (`!`),

bit à bit ou inclusif (`|`), et (`&`), ou exclusif (`^`), non (`~`), décalages (`<<`, `>>`),

d'affectation = et les raccourcis `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`,

ternaire par exemple l'expression `(a == 1) ? 2 : 3` vaut 2 si la variable `a` vaut 1, et 3 sinon,

changement de type (*casts*), par exemple `(byte) i` pour forcer `i` dans le domaine de valeur des octets.

Une expression peut aussi être

- la construction d'un objet (comme `new int[10]`),
- un appel de méthode (comme `System.out.println("Hello, World!")`).

1.2.4 Instructions

On vient de voir indirectement deux types d'instructions Java dans les exemples précédents (noter l'ajout du point-virgule ; en fin d'instruction) :

expression comme `a = b`; ou `i++`; et

déclaration comme `int i`;

Les instructions suivantes permettent de changer le flot de contrôle des instructions :

Exemple 1.3 (instructions conditionnelles).

```
int month = 8;

if (month == 1)
    System.out.println("January");
else if (month == 2)
    System.out.println("February");
else if (month == 3)
    System.out.println("March");
...
else if (month == 12)
```

```
        System.out.println("December");
else
    System.err.println("Invalid month.");

switch (month) {
    case 1: System.out.println("January"); break;
    case 2: System.out.println("February"); break;
    case 3: System.out.println("March"); break;
    ...
    case 12: System.out.println("December"); break;
    default: System.err.println("Invalid month."); break;
}
```

Exemple 1.4 (boucles).

```
int count = 1;
while (count < 11) {
    System.out.println("Count is: " + count);
    count++;
}

count = 1;
do {
    System.out.println("Count is: " + count);
    count++;
} while (count <= 11);

for (int i = 1; i < 11; i++)
    System.out.println("Count is: " + i);

int[] numbers = {1,2,3,4,5,6,7,8,9,10};
for (int item : numbers)
    System.out.println("Count is: " + item);
```

La dernière syntaxe simplifiée pour la boucle `for` est possible pour tous les types qui implémentent l'interface `Iterable<T>`.

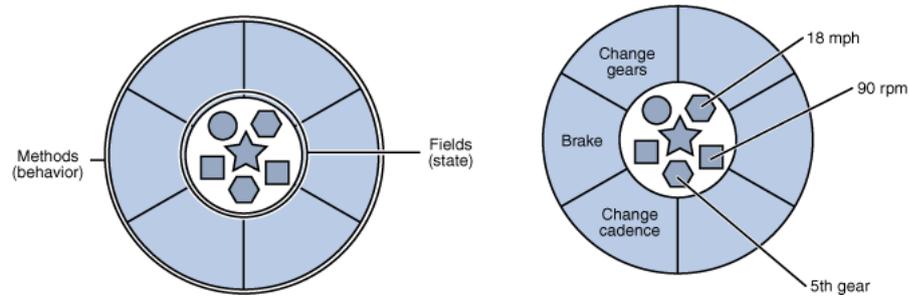
Une séquence d'instructions peut être assemblée en un bloc en la délimitant par des accolades `{` et `}`; ce bloc est vu comme une seule instruction, et on peut bien sûr les imbriquer.

1.3 Classes et objets

Il ne manque à la section précédente que la possibilité d'écrire de nouveaux types, fonctions et procédures pour pouvoir programmer de manière impérative classique. Java remplace ces notions par celles de classes et de méthodes.

1.3.1 Notions

Une application Java est décomposée en composants, des *objets*, qui vont collaborer. Chaque objet fournit une interface publique (les services qu'il offre : ses *méthodes*), et gère à sa sauce comment il fournit ces services à partir de son état interne (ses *champs*). On peut ainsi remplacer un objet par un autre similaire s'il fournit les mêmes services.



Une *classe* correspond grossièrement à un type d'objets, et déclare quels sont les champs et les méthodes fournis par chaque objet de la classe. Ainsi, la classe `Bicycle` qui suit définit l'état interne et les opérations possibles d'un vélo.

Exemple 1.5.

```
public class Bicycle {
    // fields
    private int cadence = 0;
    protected int speed = 0;
    private int gear = 1;

    // methods
    public void changeCadence(int newValue) {
        cadence = newValue;
    }

    public void changeGear(int newValue) {
        gear = newValue;
    }

    public void speedUp(int increment) {
        speed = speed + increment;
    }

    public void applyBrakes(int decrement) {
        speed = speed - decrement;
    }
}
```

Une séquence d'instructions qui utilise cette classe va créer des objets de la classe `Bicycle` (l'instancier), et appeler ses méthodes :

Exemple 1.6.

```
// Create two different Bicycle objects
Bicycle bike1 = new Bicycle();
Bicycle bike2 = new Bicycle();

// Invoke methods on those objects
bike1.changeCadence(50);
bike2.changeCadence(60);
```

Constructeurs On peut fournir des paramètres lors de la création d'un objet. On définit pour cela des *constructeurs* dans la classe. Par exemple, si on ajoute le constructeur

Exemple 1.7.

```
public Bicycle(int startCadence, int startSpeed, int startGear) {
    gear    = startGear;
    cadence = startCadence;
    speed   = startSpeed;
}
```

à la classe `Bicycle`, on pourra créer des instances déjà initialisées par un appel de la forme `new Bicycle(30, 0, 8)`.

Modificateurs De manière à encapsuler proprement ses champs et ses méthodes, une classe peut les déclarer avec une politique d'accès particulière :

`public` le champ ou la méthode est visible par tous ; cela constitue l'interface publique de la classe,

`private` le champ ou la méthode n'est visible que de l'intérieur de la classe (par exemple par les autres méthodes), ce qui permet de dissimuler les détails d'implémentation,

`protected` le champ ou la méthode n'est visible que depuis l'intérieur de la classe ou l'une de ses sous-classe (voir section suivante) : cela permet de dissimuler des détails d'implémentation qui sont néanmoins susceptibles d'être utiles aux sous-classes.

Deux autres modificateurs sont aussi très couramment utilisés

`static` le champ ou la méthode est partagé par toutes ses instances ; une telle méthode ne peut faire référence qu'à des objets créés localement, ou d'autres champs et méthodes statiques,

`final` le champ n'est pas mutable.

Par exemple, si on complétait la classe `Bicycle` par

Exemple 1.8.

```
private static final double PI = 3.141592653589793;
```

```
private static int numberOfBicycles = 0;

public Bicycle(int startCadence, int startSpeed, int startGear) {
    gear = startGear;
    cadence = startCadence;
    speed = startSpeed;

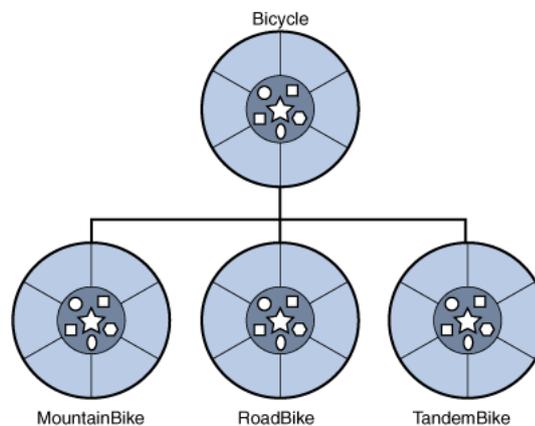
    ++numberOfBicycles;
}

public static int getNumberOfBicycles() {
    return numberOfBicycles;
}
```

alors `Bicycle.PI` désignerait la constante π , et le nombre d'objets instanciés pour la classe serait retourné par l'appel à la méthode `Bicycle.getNumberOfBicycles()`.

1.3.2 Hiérarchie de classes

Un langage orienté objet fournit des primitives qui permettent de manipuler directement les classes. Ainsi, on peut faire en sorte que la classe `MountainBike` hérite de tout le comportement de la classe `Bicycle`, sans avoir besoin de réécrire ses méthodes :



Cependant, les comportements d'un vélo de montagne ne sont pas forcément identiques à ceux d'un vélo standard, et on peut redéfinir (*overriding*) les méthodes qui ont besoin d'une mise à jour :

Exemple 1.9.

```
public class MountainBike extends Bicycle {
    // automatically inherits from cadence, speed, gear, changeCadence,
    // changeGear, speedUp, applyBrakes and the constructor
}
```

```
// mountain bikes have really good brakes
@Override // an optional annotation
public void applyBrakes(int decrement) {
    speed = speed - 2 * decrement;
}
}
```

Un objet de la classe `MountainBike` peut être vu comme un objet de la classe `Bicycle`, mais utilisera son comportement propre. On parle de *late binding*, l'implémentation à utiliser lors d'un appel de méthode est déterminée de manière dynamique :

Exemple 1.10.

```
Bicycle mb = new MountainBike();
mb.speedUp(10);
mb.applyBrakes(5); // the bike is back at speed 0
```

Classe Object Une classe Java ne peut hériter que d'une seule classe à la fois. En l'absence d'une classe mère déclarée, comme dans l'exemple 1.5, c'est la classe `Object` qui est la classe mère : c'est donc le sommet de la hiérarchie Java.

Cette classe fournit des services qui sont de ce fait communs à toutes les classes de Java, notamment les méthodes

`equals(Object)` qui détermine si l'objet courant est équivalent à l'objet passé en argument,
`hashCode()` qui calcule un hash pour l'objet courant,
`toString()` qui renvoie une chaîne de caractères correspondant à l'objet.

Interfaces Une *interface* est une classe abstraite, sans implémentation, qui ne fait que rescenser des méthodes à implémenter. Si une classe Java ne peut hériter que d'une seule autre classe, elle peut en revanche implémenter autant d'interfaces qu'elle le souhaite.

Exemple 1.11.

```
public interface MotorizedVehicle {
    public void fillTank(int gallons);
}

public class Scooter extends Bicycle implements MotorizedVehicle {
    private final int tankCapacity;
    private int tankContents;

    public Scooter(int startCadence, int startSpeed, int startGear,
        int capacity, int startTank) {
        // reuse the constructor inherited from Bicycle
        super(startCadence, startSpeed, startGear);
    }
}
```

```
        // initialize the tank
        tankCapacity = capacity;
        tankContents = startTank;
    }

    // overload the constructor
    public Scooter(int capacity) {
        // reuse the other constructor
        this(0, 0, 1, capacity, 0);
    }

    // implement the MotorizedVehicle interface
    public void fillTank(int gallons) {
        tankContents = Math.max(tankContents + gallons, tankCapacity);
    }
}
```

Polymorphisme Un objet qui implémente plusieurs interfaces peut être vu de différentes manières.

Exemple 1.12.

```
Bicycle b = new Scooter(5);
((MotorizedVehicle) b).fillTank(2); // view b as a MotorizedVehicle
b.speedUp(10);                       // view b as a Bicycle
```

Deux opérations permettent de manipuler le type courant d'un objet :

le *cast* déjà mentionné dans la section 1.2.3,

l'**opérateur** `instanceof` qui permet de vérifier si un objet peut être vu comme implémentant un type, par exemple `b instanceof MotorizedVehicle` vaudrait `true` dans l'exemple 1.12 ci-dessus.

1.3.3 Énumérations

Un type énuméré contient un nombre fini d'instances possibles.

Exemple 1.13.

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

C'est en fait équivalent à la déclaration d'une classe avec des champs privés finals :

Exemple 1.14.

```
public class Day {
```

```
public static final Day SUNDAY = new Day();
public static final Day MONDAY = new Day();
public static final Day TUESDAY = new Day();
public static final Day WEDNESDAY = new Day();
public static final Day THURSDAY = new Day();
public static final Day FRIDAY = new Day();
public static final Day SATURDAY = new Day();
}
```

On peut fournir des arguments lors de la construction des champs d'une énumération :

Exemple 1.15.

```
public enum Day {
    SUNDAY(7), MONDAY(1), TUESDAY(2), WEDNESDAY(3),
    THURSDAY(4), FRIDAY(5), SATURDAY(6);

    private final int dayInWeek;

    private Day(int i) {
        dayInWeek = i;
    }

    public int getDayInWeek() {
        return dayInWeek;
    }
}
```

et même redéfinir des méthodes

Exemple 1.16.

```
public enum Day {
    SUNDAY {
        public int getDayInWeek() {
            return 7;
        }
    },
    MONDAY {
        public int getDayInWeek() {
            return 1;
        }
    },
    TUESDAY {
        public int getDayInWeek() {
            return 2;
        }
    }
}
```

```
    },
    WEDNESDAY {
        public int getDayInWeek() {
            return 3;
        }
    },
    THURSDAY {
        public int getDayInWeek() {
            return 4;
        }
    },
    FRIDAY {
        public int getDayInWeek() {
            return 5;
        }
    },
    SATURDAY {
        public int getDayInWeek() {
            return 6;
        }
    };

    public abstract int getDayInWeek();
}
```

2 Travaux pratiques

Les travaux pratiques de cette première séance vont vous permettre de vous familiariser avec les concepts de la programmation objet en Java.

Avant de commencer, charger dans votre navigateur un lien indispensable pour s'y retrouver : l'API Java <http://java.sun.com/javase/6/docs/api/> (et le mettre dans vos favoris).

2.1 Hello, World!

Comme de coutume, le premier programme que nous allons voir en Java imprime sur la sortie standard le message non moins standard « *Hello, World!* ».

1. Aller dans le répertoire `Hello/v1` du TP. Lire le contenu de fichier `Hello.java`, le compiler et exécuter le programme obtenu.
2. Dans le répertoire `Hello/v2` : une nouvelle version du programme est donnée de manière incomplète. En particulier, il faut redéfinir les méthodes `toString()`, `equals(Object)` et `hashCode()` héritées de la classe `Object`, en se basant sur le contenu du message d'accueil.

3. Comment pourrait-on éviter de réécrire ces méthodes `toString()`, `equals(Object)` et `hashCode()` ?

2.2 Jeu d'échecs

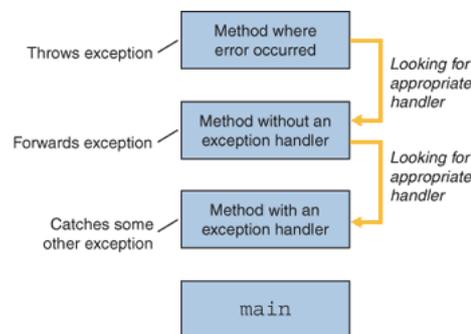
2.2.1 Programmation objet

Le répertoire `Chess/v1` du TP contient une implémentation sommaire du comportement des pièces d'un jeu d'échecs. La classe centrale, `Piece`, est fournie intégralement, ainsi que la classe `Square` qui sert à identifier une case du plateau de jeu. Cependant, la classe `Piece` délègue plusieurs opérations à l'objet présent dans le champ `PieceType` `type`.

Réfléchir à comment exploiter les mécanismes objets pour implémenter `PieceType`. Comment assurer que le comportement et le nom d'une pièce ne seront pas dissociés par erreur ? Fournir une implémentation pour le roi, la reine, le fou et la tour. On ignorera bien sûr les contraintes liées à l'existence d'autres pièces sur le plateau, puisque le code n'implémente pas de plateau !

2.2.2 Exceptions

Dans les classes écrites jusqu'à présent, on a quelques restrictions et hypothèses sur les attributs ou les arguments des méthodes. On souhaite notamment que les pièces soient sur une case de l'échiquier, et non en dehors, et que les déplacements n'amènent pas la pièce concernée en dehors de l'échiquier. Java fournit un dispositif de gestion des erreurs de haut niveau pour traiter ce genre de problèmes et permettre un meilleur contrôle.



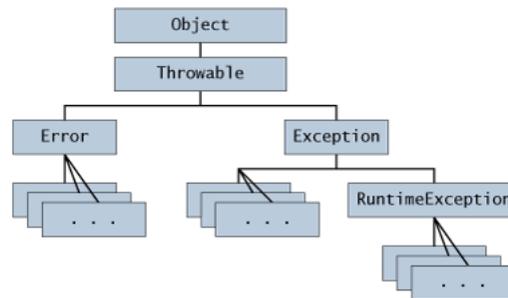
En Java, les erreurs et les exceptions sont des objets (presque) comme les autres, qui héritent d'une classe commune : celle des objets `Throwable`. Les objets quiinstancient `Throwable` ou une de ses sous-classes peuvent être "soulevés" pendant l'exécution d'une méthode. On utilise pour cela le mot-clef `throw` :

Exemple 2.1.

```
if (/* condition to satisfy */)
```

```
    /* do something */;  
else  
    throw new Throwable();
```

Lorsque un objet `Throwable` est soulevé, le flot d'exécution est interrompu. Par défaut, l'exécution du programme s'arrête, et la machine virtuelle écrit dans la sortie d'erreur le nom de l'exception, un message d'information, ainsi que la pile des appels de méthode au moment où l'objet a été soulevé. Il est possible pour le programmeur de capturer un objet soulevé afin de traiter le problème sans arrêter l'exécution du programme. Nous verrons comment faire un peu plus loin.



Cependant, nous n'utiliserons jamais la classe `Throwable` directement, mais toujours une de ses sous-classes `Error` ou `Exception` (ou une de leur sous-classe). Une `Error` sera soulevée suite à une condition anormale et ne devrait pas être capturée : si une erreur est soulevée, le programme devrait s'arrêter immédiatement. Une `Exception` en revanche, peut-être capturée par le programme qui pourra décider de continuer l'exécution du programme.

Quand une `Exception` peut être soulevée dans une méthode, on doit le signaler dans la signature de celle-ci. On utilise pour cela le mot-clef `throws` :

Exemple 2.2.

```
public static void f1 () throws Exception {  
    if (/* condition to satisfy */)   
        /* do something */;  
    else  
        throw new Exception();  
}
```

Si une sous-classe est utilisée, on peut utiliser le nom de cette sous-classe à la place d'`Exception` après le `throws`. En fait, on peut utiliser n'importe quelle sous-classe de `Throwable`, voire plusieurs séparées par des virgules. Cependant, il est inutile de préciser que des `Error` peuvent être soulevées puisqu'elles ne sont pas censées être capturées.

Quand on invoque une méthode pouvant soulever des exceptions, on peut décider de les capturer. On utilise pour cela les mot-clef `try`, `catch` et `finally` :

Exemple 2.3.

```
public static void f2 () {
    try {
        /* some code */
        f1(); /* a function that throws an exception */
        /* some other code */
    } catch (Exception e) {
        System.err.println("Caught some exception");
    } finally { // optional
        /* some code that will always be executed,
           even if there was an exception */
    }
}
```

La méthode va essayer d'exécuter le bloc de code dans `try`. Si une exception est soulevée, l'exécution du bloc s'arrête. Le bloc `catch` capture l'exception si elle est une instance de la classe indiquée entre parenthèse. Le code du bloc `catch` est alors exécuté. Le code du bloc optionnel `finally` est exécuté dans tous les cas. Si plusieurs types d'exceptions différents peuvent être soulevés, on peut écrire un bloc `catch` pour chaque sous-classe d'`Exception` possible. La fonction `f2` n'est pas obligée de capturer les exceptions soulevées par `f1` et peut se contenter de les transmettre au niveau supérieur : il suffit pour cela de déclarer dans sa signature qu'elle peut soulever les mêmes erreurs que `f1` grâce à `throws`.

L'argument `e` dans la parenthèse du `catch` est l'objet de type `Exception` capturé. Il peut être utilisé pour obtenir ou afficher des informations. En effet, `Exception` (et ses sous-classes) héritent des méthodes suivantes de la classe `Throwable` :

`getMessage()` renvoie un message d'information qui peut être passé en argument au constructeur lors de la création d'une nouvelle exception, et

`printStackTrace()` affiche la pile des appels au moment où l'exception est soulevée.

Beaucoup de sous-classes d'`Exception` existent déjà dans l'API de base de Java. Vous avez peut-être déjà rencontré les `NullPointerException` ou `IndexOutOfBoundsException`. Mais on y trouve aussi la classe `IllegalArgumentException`, qui correspond bien aux problèmes que nous souhaitons signaler. Modifiez la signature et le code des constructeurs et des méthodes `move` des classes `Square` et `Piece` pour qu'elles soulèvent des `IllegalArgumentException` quand il le faut. Modifiez votre code de test pour qu'il puisse capturer d'éventuelles erreurs.

2.2.3 Assertions

Java dispose également d'un mot-clef qui permet de s'assurer que des conditions qui devraient être évidentes sont vérifiées : `assert`. Il doit être suivi d'une expression booléenne censée être vraie, et éventuellement de deux points et d'un objet à afficher si la condition n'est pas vérifiée.

Exemple 2.4.

```
if (i%2 == 0) {
    /* do something */;
} else {
    assert i%2 == 1 : i;
    /* do something else */
}
```

Les assertions sont vérifiées à l'exécution quand on fournit l'option `-ea` à la JVM :

```
java -ea MyProgramWithAssertions
```

Quelles portions du code qui vous a été fourni pourraient bénéficier d'assertions ?

2.2.4 Entrées-sorties

Le package `java.io` contient de nombreuses classes pour gérer les entrées et sorties pour lire depuis ou écrire sur l'entrée standard, un fichier, ou d'autres sources (chaînes de caractères, *buffer* ...). On distingue les flux d'octets (*byte*) et les flux de caractères (*char*) : en effet, en Java, la taille d'un caractère dépend de l'encodage, et peut dépasser un octet (par exemple en UTF-8). On a alors les classes suivantes :

- Flux d'octets
 - Entrée : `InputStream`
 - Sortie : `OutputStream`
- Flux de caractères
 - Entrée : `Reader`
 - Sortie : `Writer`

Les flux d'entrées disposent des fonctions suivantes :

```
public int read ();
public int read (byte[] buf);
public int read (byte[] buf, int off, int len);
```

`buf` est un `char[]` dans un `Reader`. Toutes ces fonctions renvoient `-1` si le flux est fini. Sinon `read()` lit un caractère ou un byte et renvoie sa valeur sous forme d'un `int` qu'il suffit de transtyper pour utiliser. `read(buf, off, len)` lit au plus `len` octets ou caractères, qu'il enregistre à partir de l'indice `off` de `buf` (`off = 0` et `len = buf.length` si ils ne sont pas passés en argument), et renvoie le nombre d'octets ou de caractères lus.

Les flux de sorties disposent des fonctions suivantes :

```
public void write (int c);
public void write (byte[] buf);
public void write (byte[] buf, int off, int len);
```

`buf` est un `char[]` dans un `Writer`. Vous devinez facilement leur comportement d'après ce qui précède. Toutes ces fonctions peuvent soulever des `IOException`.

On peut obtenir un `Reader` ou un `Writer` depuis un `InputStream` ou un `OutputStream` (transformer un flux d'octet en un flux de caractères) à l'aide des classes d'encapsulation `InputStreamReader` et `OutputStreamWriter`

Deux autres classes d'encapsulations permettent de faciliter l'entrée et la sortie de chaînes de caractères : `PrintWriter` et `BufferedReader`. La première encapsule un `Writer` et permet d'écrire facilement des chaînes de caractères à partir de n'importe quel type de donnée *via* les fonctions `print` et `println`. La seconde encapsule un `Reader` et gère toute seule les entrées dans un tampon. Aussi, on peut directement lui demander de lire des lignes entières sans savoir à l'avance leurs tailles, avec `readLine()`.

Les flux disponibles sont ceux présents dans la classe `System` : `System.out`, `System.err` et `System.in`. `out` et `err` sont des `PrintStream`, tandis que `in` est un simple `InputStream` qu'on peut vouloir encapsuler. D'autre part on peut utiliser des flux de fichiers comme `FileReader` et `FileWriter` qui ont un constructeur qui ne demande que le nom du fichier voulu. Avec les fichiers, il faut explicitement fermer les flux après utilisation avec la méthode `close()`.

Voici un exemple d'utilisation de certaines de ces classes pour copier un fichier textuel ligne par ligne (noter l'utilisation du bloc `finally` pour fermer les fichiers même en cas d'exception) :

Exemple 2.5.

```
/*
 * Copyright (c) 1995 - 2008 Sun Microsystems, Inc. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * - Redistributions of source code must retain the above copyright
 *   notice, this list of conditions and the following disclaimer.
 *
 * - Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 *
 * - Neither the name of Sun Microsystems nor the names of its
 *   contributors may be used to endorse or promote products derived
 *   from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
 * IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
```

```
* PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
* PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
* LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
* NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
* SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/

import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;

public class CopyLines {
    public static void main(String[] args) throws IOException {
        BufferedReader inputStream = null;
        PrintWriter outputStream = null;

        try {
            inputStream =
                new BufferedReader(new FileReader("xanadu.txt"));
            outputStream =
                new PrintWriter(new FileWriter("characteroutput.txt"));

            String l;
            while ((l = inputStream.readLine()) != null) {
                outputStream.println(l);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

Dans votre test, faites une invite de commande qui propose à l'utilisateur de taper de choisir une pièce de l'échiquier, et de lui faire faire des mouvements (par exemple, sous la forme "+2 -2"). Essayez ensuite de sauvegarder dans un fichier les positions des pièces à la fin du test. Faites en sorte qu'au début du test, il puisse rappeler les anciennes positions des pièces, et éventuellement les prendre comme nouvelles positions de départ.