Chapitre 1

Programmer avec Java

Ce chapitre ne se propose pas comme une description exhaustive de Java, mais plutôt comme un guide à l'utilisation de certains traits avancés, comme l'héritage et les classes abstraites. Il ne saurait se substituer à un cours de langages de programmation, comme par exemple celui d'année 3. On trouvera des éléments complémentaires dans [9]. On pourra consulter également les tutoriels de Java sur la page de Sun[1]. Notre optique est d'être capable d'utiliser ces notions quand nous aborderons les graphes.

Plus philosophiquement, les langages modernes tournent autour de cette question : comment écrire au kilomètre des programmes corrects, lisibles et efficaces ? Une des façons de répondre est d'utiliser des classes prédéfinies, efficaces, standardisées, et nous en reparlerons au chapitre 14. Nous nous intéresserons plutôt ici aux mécanismes qui permettent la réutilisation et le partage de ces ressources.

1.1 Modularité

1.1.1 Motivation

La modularité est un concept fondamental de la programmation. Dans un programme, on cherche toujours où on doit mettre quelles données. Pour s'y retrouver, il est beaucoup plus simple de pouvoir contrôler quelles données sont accessibles à qui (fonctions, classes, programmes, etc.). Dans certains langages, on a accès à essentiellement deux niveaux : une variable peut être *locale* à une fonction (ou plus généralement à un *bloc* de code) et n'est pas vue de l'extérieur de la fonction ; ou bien elle est *globale*, ce qui fait qu'elle est accessible à toutes les fonctions à la fois. Ces deux niveaux se révèlent rapidement insuffisants.

On est donc amené à introduire des contrôles plus stricts. On peut rassembler dans une même bibliothèque (appelée généralement *module* dans ce contexte), toutes les fonctions opérant sur des objets communs. L'idée derrière les modules est bien évidemment celui de la réutilisation et de la composition de petits modules pour en faire des plus gros, ce qui conduit à de gros programmes, etc.

[1] `http://java.sun.com/docs/books/tutorial/index.html`

De même, on peut élargir les types des données pour fabriquer un type adapté à un cas spécifique, des types peuvent être locaux ou globaux, etc. En Java, types et modules se confondent dans la notion de classe, qui est à la fois type utilisateur et module contenant les fonctions associés à ce type.

1.1.2 Modules

Très généralement, un module se compose de deux parties :
- l'*interface*[2] accessible depuis l'extérieur du module ;
- l'*implémentation* accessible de l'intérieur du module.

Un des avantages de cette dichotomie est de permettre d'utiliser le principe d'*encapsulation*. On définit les fonctionnalités du module dans la partie interface (données partagées, types, fonctions), car c'est ce qui intéresse l'utilisateur, et on en réalise une implantation dans la parite implémentation. Prenons l'exemple classique de la file d'attente. On a besoin de savoir faire trois opérations : construction, ajouter, supprimer. Mais on peut implanter les files d'au moins deux façons : par un tableau géré de façon circulaire, ou bien par une liste. L'extérieur n'a pas besoin de savoir quelle représentation est utilisée, et parfois, on peut la changer sous ses pieds.

Dans certains langages (comme Pascal), un module contient une interface et une implantation, et on ne peut avoir plusieurs implantations correspondant à la même interface. En Java, l'utilisation des classes abstraites permet de résoudre ce problème (cf. section 2.2.2).

Reprenons le cas d'une file d'attente. On peut écrire :

```
public class FIFO{
  private int debut, fin;
  private boolean pleine, vide;
  private int[] contenu;

  public FIFO(int n){
    debut = 0; fin = 0;
    pleine = n == 0; vide = true;
    contenu = new int[n];
  }
  public static void ajouter(int x, FIFO f){
    if(f.pleine) throw new Error("File Pleine.");
    f.contenu[f.fin] = x;
    f.fin = (f.fin + 1) % f.contenu.length;
    f.vide = false; f.pleine = f.fin == f.debut;
  }
  public static int supprimer(FIFO f){
    if(f.vide)
        throw new Error("File Vide.");
```

[2] Attention : en Java, interface est un mot clef pour désigner autre chose, cf. section 2.2.2.

```
    int res = f.contenu[f.debut];
    f.debut = (f.debut + 1) % f.contenu.length;
    f.vide = f.fin == f.debut; f.pleine = false;
    return res;
  }
```

On rend tous les champs privés (voir section 2.2.1) de sorte que l'utilisateur ne puisse utiliser directement l'implantation (ce qui rendrait son code sensible à tout changement de la définition de FIFO). Par contre, ce qui est public, c'est le constructeur, ainsi que les fonctions d'accès.

On pourrait également écrire une autre classe avec la même interface (et les mêmes signatures de fonction), mais cette fois ci utilisant une liste chaînée en interne :

```
public class FIFO{
  private Liste debut, fin;
  public FIFO(int n){ debut = null; fin = null; }
  public static void ajouter(int x, FIFO f){
    if(f.fin == null)
      f.debut = f.fin = new Liste(x);
    else{
      f.fin.suivant = new Liste(x);
      f.fin = f.fin.suivant;
    }
  }
  public static int supprimer(FIFO f){
    if(f.debut == null) throw new Error("File Vide.");
    else{
      int res = f.debut.val;
      if(f.debut == f.fin)
        f.debut = f.fin = null;
      else f.debut = f.debut.suivant;
      return res;
    }
  }
```

Ces deux exemples s'excluent mutuellement, si l'on cherche à garder le même nom à la classe. Nous verrons au chapitre 2 que les classes abstraites permettent de décrire plusieurs implantations réalisant les mêmes fonctionnalités. Nous pouvons également utiliser des interfaces, voir la section 1.3.

1.2 Programmation par objets

1.2.1 Motivations

Les premiers langages informatiques relèvent d'une programmation dite *procédurale* où on applique des fonctions (ou procédures) à des données. La programmation dirigée par les données (ou programmation par objet) est un paradigme plus récent. Pour simplifier, la première vision du monde est celle de calculs appliqués à des paramètres d'entrée (donc une approche très fonctionnelle), alors que la seconde considère les objets comme la quantité importante, et qu'un objet peut posséder ses propres fonctions de traitement.

En Java, un objet est une instance d'une classe. Il a un état (la valeur de ses champs de données) et un ensemble de méthodes attachées. Les données statiques (*static*) sont partagées par tous les objets de la classe.

Quels avantages a-t-on à programmer objet ? Même si la modularité n'impose pas la programmation objet, celle-ci bénéficie énormément de cette possibilité, tant les deux ont le même objectif de rassembler en un même endroit (du programme) les données et fonctions applicables à un objet. Un autre avantage est celui de l'héritage : on peut définir des classes, qu'on va étendre dans un but plus spécifique, et donc aboutir à une programmation incrémentale, qui nous rapproche de notre objectif de l'écriture de gros programmes. C'est le cas de l'AWT (*Abstract Windowing Toolkit*) qui permet de faire des dessins en Java.

On peut se poser diverses questions pour choisir : préfere-t-on contrôler les fonctions ou les données ? Les petits programmes peuvent se faire indifféremment dans les deux styles. Si le programme devient gros (disons au-delà de 10,000 lignes), la programmation objet a des avantages certains. Entrent alors en ligne de compte des considérations comme la stratégie de modification ou d'évolution du code, et du côté incrémental que cela peut représenter, ce qui est très prisé dans l'industrie.

La programmation objet est quasi impérative quand on utilise des classes prédéfinies déjà en syntaxe objet (AWT), ce que nous verrons plus loin.

Gardons la tête froide. En dernière analyse, c'est une affaire de goût. . . .

1.2.2 L'héritage par l'exemple

Nous allons considérer l'exemple ultra-classique des points et des points colorés. On commence par une classe `Point` qui code un point du plan par ses coordonnées `x` et `y` :

```
class Point{
  double x, y;
}
```

On décide de créer un nouveau type qui aura la particularité d'être un point avec une couleur :

```
class PointC extends Point{
  final static int JAUNE = 0, ROUGE = 1;
  int c;
  PointC(double x0, double y0, int col){
    x = x0; y = y0; c = col; // [1]
  }
  public static void main(String[] args){
    PointC pc = new PointC(3, 4, JAUNE);
  }
}
```

La classe `PointC` est une *sous-classe* de `Point`. Elle *hérite* des champs et des méthodes définis dans sa classe père, ce qui explique la syntaxe de la ligne [1] : à la création d'un `PointC`, on a déjà à sa disposition les deux champs `x` et `y` hérités de la classe `Point`.

Complétons l'exemple :

```
class Point{
  double x, y;
  static int NP = 1;
  static void afficherAbscisse(Point p){
    System.out.println("P: "+p.x);
  }
}
class PointC extends Point{
  final static int JAUNE = 0, ROUGE = 1;
  int c;
  PointC(double x0, double y0, int col){
    x = x0; y = y0; c = col;
  }
  public static void main(String[] args){
    PointC pc = new PointC(3, 4, JAUNE);
    afficherAbscisse(pc);
    System.out.println("NP="+NP);
  }
}
```

Il y a conversion *implicite* de `PointC` en `Point`, dans l'appel de la méthode `afficherAbscisse` de la classe `Point`. D'autre part, la variable statique `NP` est elle aussi accessible depuis `PointC`.

Dans la sous-classe, on ajoute ou on redéfinit des champs ou méthodes : on dit qu'on a *spécialisé* (en anglais, c'est l'*overriding*) ces champs ou méthodes. Par exemple, dans le code :

```
class Point{
```

```
  ...
  static void afficherAbscisse(Point p){
    System.out.println("Point: "+p.x);
  }
}
class PointC extends Point{
  ...
  static void afficherAbscisse(Point p){
    System.out.println("PointC: "+p.x);
  }
  public static void main(String[] args){
    PointC pc = new PointC(3, 4, JAUNE);
    afficherAbscisse(pc);
    Point.afficherAbscisse(pc);
  }
}
```

on a spécialisé dans `PointC` la méthode `afficherAbscisse`. On peut également spécialiser des méthodes d'objet :

```
class Point{
  ...
  void afficher(){
    System.out.println("p: "+x);
  }
}
class PointC extends Point{
  ...
  void afficher(){
    System.out.println("pc: "+x);
  }
  public static void main(String[] args){
    PointC pc = new PointC(3, 4, JAUNE);
    Point p = new Point();
    p.x = 1;

    pc.afficher();
    p.afficher();
  }
}
```

1.2.3 Propriétés de l'héritage

En Java, l'héritage est dit *simple*[3], c'est-à-dire qu'une classe peut être sous-classe d'une autre classe et d'une seule. Par contre, une classe peut avoir plusieurs sous-classes. Les diagrammes d'héritage sont donc des arbres. Cela facilite grandement le choix de la méthode à appliquer à un objet, selon le principe suivant : on choisit *statiquement* la méthode la plus spécialisée, c'est-à-dire appartenant à la plus petite sous-classe connue contenant l'objet.

La classe père est accessible par le mot clef **super** (un peu comme si l'on écrivait `(ClassePere)this`) ; on ne peut l'utiliser que dans des méthodes d'objets : `super.f` est la méthode `f` de la classe parente ; `super()` (avec d'éventuels arguments) est le constructeur (appelé par défaut) de la classe parente. Par exemple :

```
class PointC extends Point{
  final static int JAUNE = 0, ROUGE = 1;
  int c;
  PointC(double x0, double y0, int col){
    super(); // facultatif
    x = x0; y = y0; c = col;
  }
  public static void main(String[] args){
    PointC pc = new PointC(3, 4, JAUNE);
  }
}
```

Dans le cas d'un constructeur explicite pour la classe père, on est obligé d'appeler **super** :

```
class Point{
  double x, y;
  Point(double x0, double y0){
    x = x0; y = y0;
  }
}
class PointC extends Point{
  final static int JAUNE = 0, ROUGE = 1;
  int c;
  PointC(double x0, double y0, int col){
    super(x0, y0); // nécessaire
    c = col;
  }
  public static void main(String[] args){
    PointC pc = new PointC(3, 4, JAUNE);
  }
```

[3] contrairement à C++, Smalltalk ou Ocaml, dans lesquels l'héritage est *multiple*.

```
}
```

Exemple. En Java, on peut masquer des champs. Par exemple, le code suivant est (malheureusement ?) licite :

```
class PointC extends Point{
  final static int JAUNE = 0, ROUGE = 1;
  int x;
  int c;
  ...
}
```

Le nouveau `x` masque l'ancien, qui est toujours accessible par **`super`**.

1.2.4 Contrôle d'accès et sous-classes

Les champs ou méthodes d'une classe peuvent être :
- **`public`** pour permettre l'accès depuis toutes les classes ;
- **`private`** pour restreindre l'accès aux seules expressions ou fonctions de la classe courante ;
- par défaut pour autoriser l'accès depuis toutes les classes du même paquetage ;
- **`protected`** pour restreindre l'accès aux seules expressions ou fonctions de sous-classes de la classe courante.
- Un champ **`final`** ne peut être redéfini (on peut donc optimiser sa compilation).
- une classe **`final`** ne peut être spécialisée, c'est le cas de la classe `String` par exemple.
- une méthode peut être déclarée **`final`** (pas besoin de déclarer toute la classe **`final`**).

Les méthodes spécialisées doivent fournir au moins les mêmes droits d'accès que les méthodes originellement définies.

1.2.5 Un peu de typage

Théorie

Rappelons qu'un type est un ensemble de valeurs partageant une même propriété. En Java, il y a des types primitifs : **`int`**, **`char`**, etc ; une classe est un type. On a déjà rencontré des classes prédéfinies comme la classe `String`.

Comment le compilateur vérifie-t-il le typage ? Il s'agit d'une vérification statique (à la compilation) contrairement au typage dynamique (à l'exécution). L'intérêt du typage statique est de permettre la détection d'erreurs le plus tôt possible, mais aussi d'optimiser le code, et d'assurer une certaine sécurité (pas d'atteinte à la mémoire par exemple). Le typage dynamique peut être dans certains cas plus précis (prendre l'exemple du cas de **`if(a) return 1; else return false;`**).

Il n'est pas question ici de faire un cours de théorie des types. Nous renvoyons au cours Principes des Langages de Programmation d'année 3, ainsi qu'aux livres de Benjamin Pierce ou de Abadi-Cardelli.

Et en Java

En Java, chaque objet possède un certain type lors de sa création. L'opérateur **`instanceof`** teste l'appartenance d'un objet à une classe. Ainsi :

```
if(p instanceof PointC)
  System.out.println ("couleur = " + p.c);
```

Donc l'expression `(PointC)p` équivaut à :

```
if(p instanceof PointC)
  p
else
  throw new ClassCastException();
```

On parle de sous-classe comme on parle de sous-type. On note $x : t$ pour dire que x est de type t et $t <: t'$ signifie $x : t \Rightarrow x : t'$ pour tout x. On dit que t est un *sous-type* de t'. En Java, les types suivants sont des sous-types les uns des autres, et il y a conversion explicite si nécessaire :

`byte` $<:$ **`short`** $<:$ **`int`** $<:$ **`long`**
`float` $<:$ **`double`**
`char` $<:$ **`int`** $<:$ **`long`**

Une classe est un type ; une sous-classe un sous-type. On propage la notation : $C <: C'$ si C est une sous-classe de C'.

Ex. `PointC` $<:$ `Point`.

On a déjà dit qu'en Java, l'héritage était simple. Il existe ainsi un arbre qui contient toute la hiérarchie des classes, dont la racine est la classe `Object` :

$$\forall C \quad C <: \texttt{Object}$$

(C est une classe ou un tableau quelconque). Les méthodes de `Object` sont
`clone`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `wait`, `equals`, `toString`.
Seules `equals`, `toString`, `finalize` peuvent être redéfinies. Comme conséquence de ce qui précède, *tous* les objets contiennent ces méthodes.

Comment se raccrochent les types primitifs dans ce tableau ? On convertit les scalaires **`int`**, **`float`**, **`char`**, ... en objets avec un « conteneur »(*autoboxing*) :

```
int x = 3;
Objet xobj = new Integer(x);
int y = xobj.intValue();
```

Quelques exemples extrêmes

Le code qui suit est valide, mais est (souvent) déconseillé[4] :

```
class A{
  int x;

  A(){ x = 1; }
}
class B extends A{
  B(){ x = 2; }
  public static void main(String[] args){
    B b = new B();
    A a = new B();
    System.out.println(b.x);
    System.out.println(a.x);
  }
}
```

Le programme affiche :

```
2
2
```

Le *type créé* de `a` est `B`, mais son *type apparent* est `A`. La conversion `A a = new B()` marche puisque `B` est sous-classe de `A`.

Cette propriété permet d'écrire :

```
class Point{
  double x, y;
  static void translation(Point p,
                          double dx, double dy){
    p.x += dx; p.y += dy;
  }
}
class PointC extends Point{
  final static int JAUNE = 0, ROUGE = 1;
  int c;
  PointC(double x0, double y0, int col){
    x = x0; y = y0; c = col;
  }
  public static void main(String[] args){
    PointC pc = new PointC(3, 4, JAUNE);
    translation(pc, 1, 2);
  }
```

[4]Cette section peut être sautée en première lecture.

```
}
```

Par contre, le code suivant ne peut marcher :

```
class Point{
  ...
  static Point translation(Point p,
                           double dx, double dy){
    Point pt = new Point();
    pt.x = p.x + dx; pt.y = p.y + dy;
    return pt;
  }
}
class PointC extends Point{
  ...
  public static void main(String[] args){
    PointC pc = new PointC(3, 4, JAUNE);
    pc = (PointC)translation(pc, 1, 2);
  }
}
```

Il n'y a pas de conversion père → fils. La compilation marche (avec le cast), mais pas l'exécution (`ClassCastException`).

Héritage et surcharge

La surcharge est déterminée à la *compilation*. On parle de liaison *retardée* : l'héritage permet d'appeler une méthode de la même classe sans savoir exactement quelle sous-classe va l'instancier.

Pour illustrer cette notion, on demande quelle est la valeur affichée par le programme suivant ?

```
class C{
  void f(){
    g();
  }
  void g(){
    System.out.println(1);
  }
}
class D extends C{
  void g(){
    System.out.println(2);
  }
  public static void main(String[] args){
    D x = new D();
```

```
    x.f();
  }
}
```

La réponse est 2. En effet, que se passe-t-il ? Le compilateur ne se plaint pas, car à la création, x étant de type $D <: C$, il possède bien une méthode f. À l'exécution, on utilise la règle suivant laquelle on applique la méthode de la plus petite classe possible contenant x, ici la classe D, qui a bien une méthode g, qui affiche 2 !

Notons que ce genre de propriété est difficile à utiliser quand on débute. Il n'accroît pas la lisibilité du code non plus, et donc nous engageons le programmeur à ne l'utiliser que quand la situation l'exige, et elle peut très bien ne jamais se produire...

Exercice 1.2.1 Quel est le résultat produit par le programme suivant quand $T, T' \in \{A, B\}$ et $U, U' \in \{A, B\}$ avec $T' <: T$ et $U' <: U$?

```
class A{
  public static void main(String[] args){
    T x = new T'(); U y = new U'();
    System.out.println (x.f(y));
  }

  int f(A y) { return 1; }
}

class B extends A{
  int f(B y) { return 2; }
}
```

1.3 Les interfaces de Java

Les *interfaces* à la Java permettent de passer des *contrats* entre programmeurs... Par exemple, on peut imaginer une interface décrivant ce qu'on attend d'une voiture, laissant une implantation particulière à chaque constructeur.

Une interface possède les propriétés suivantes :

- tous les champs sont indéfinis. Ses champs de données sont constants ; ses méthodes sont abstraites et publiques.
- Une interface peut *spécialiser* une autre interface.
- Une classe peut *implémenter* une ou plusieurs interfaces.
- Il n'y a pas de fonctions statiques, ou champs de données modifiables dans une interface.
- Les interfaces sont une bonne manière d'exiger la présence de certains champs dans une classe.

Reprenons l'exemple de la FIFO. On peut définir une interface qui décrit les propriétés attendues d'une FIFO :

```
interface FIFO{
  boolean estVide();
  void ajouter(int x);
  int supprimer();
}
```

On peut alors en réaliser deux implantations distinctes :

```
public class FIFO_T implements FIFO{
  private int debut, fin;
  private boolean pleine, vide;
  private int[] contenu;

  public FIFO_T(int n){
    debut = 0; fin = 0;
    pleine = n == 0; vide = true;
    contenu = new int[n];
  }
  public boolean estVide(){
    return vide;
  }
  public void ajouter(int x){
    if(pleine) throw new Error("File Pleine.");
    contenu[fin] = x;
    fin = (fin + 1) % contenu.length;
    vide = false; pleine = fin == debut;
  }
  public int supprimer(){
    if(vide)
        throw new Error("File Vide.");
    int res = contenu[debut];
    debut = (debut + 1) % contenu.length;
    vide = fin == debut; pleine = false;
    return res;
  }
}
```

et celle utilisant une liste :

```
public class FIFO_L implements FIFO{
  private Liste debut, fin;
  public FIFO(int n){ debut = null; fin = null; }
  public boolean estVide(){
    return (debut == null);
  }
```

```
  public void ajouter(int x){
    if(fin == null)
      debut = fin = new Liste(x);
    else{
      fin.suivant = new Liste(x);
      fin = fin.suivant;
    }
  }
  public int supprimer(){
    if(debut == null) throw new Error("File Vide.");
    else{
      int res = debut.val;
      if(debut == fin)
        debut = fin = null;
      else debut = debut.suivant;
      return res;
    }
  }
}
```

On peut alors écrire un algorithme général qui utilise les `FIFO` :

```
public class C{
  public static void Algo(FIFO f){
    while(! f.estVide()){
      int x = f.supprimer();
      System.out.println(x);
    }
  }
  public static void main(String[] args){
    FIFO_L L = new FIFO_L(10);
    for(int i = 0; i < 10; i++)
      L.ajouter(i);
    Algo(L);
    FIFO_T T = new FIFO_T(20);
    for(int i = 0; i < 10; i++)
      T.ajouter(i);
    Algo(T);
  }
}
```

En clair, on peut utiliser une **`interface`** comme un type.

1.3.1 Spécialisations multiples

Donnons encore un autre exemple, où on spécialise deux interfaces à la fois. On va se donner une interface de pile :

```
interface Pile{
  boolean estVide();
  void empiler(int x);
  void depiler();
}
```

On peut réaliser une classe de liste qui implante les deux et qui va utiliser un tableau :

```
public class MaListe implements FIFO, Pile{
  private int debut, fin;
  private boolean pleine, vide;
  private int[] contenu;

  MaListe(int n){
    debut = 0; fin = 0;
    pleine = n == 0; vide = true;
    contenu = new int[n];
  }

  boolean estVide(){ ... } // même signature pour FIFO et Pile!
  void ajouter(){ ... }
  int supprimer(){ ... }
  void empiler(int x){ ... }
  int depiler(int x){ ... }
}
```

On peut alors imaginer un programme de distribution de cartes :

```
public class Jouer{
  public static void main(String[] args){
    MaListe jeu = new MaListe(32);
    for(int i = 0; i < 32; i++)
      jeu.ajouter(i); // on enfile dans l'ordre 0, 1, ..., 31
    while(! jeu.estVide()){
      // on affiche dans l'ordre 0, 1, ..., 31
      System.out.println(jeu.depiler());
    }
  }
}
```

Nous ne sommes plus très loin de la classe prédéfinie `LinkedList`.

1.4 Les génériques de Java

Java possède (depuis la version 5.0) une notion de polymorphisme, ce qui permet d'écrire une seule fois certaines classes, comme par exemple une classe de liste d'objets[5].

Pour assurer la réutilisabilité du code, il est commode de fabriquer du code général, mais qui peut conduire à des problèmes, qui ne seront détectés qu'à l'exécution.

L'exemple de base est :

```
public class Box {

  private Object object;

  public void add(Object object) {
    this.object = object;
  }

  public Object get() {
    return object;
  }
}
```

qui se contente mettre un objet dans une boîte. On peut alors place n'importe quel type d'objet dans la boîte :

```
public class BoxDemo1 {

  public static void main(String[] args) {

    // ONLY place Integer objects into this box!
    Box integerBox = new Box();

    integerBox.add(new Integer(10));
    Integer someInteger = (Integer)integerBox.get();
    System.out.println(someInteger);
  }
}
```

Ce code ne nécessite pas de cast, puisque tout type est sous-type d'Object, donc on peut convertir un fils en son père sans problème.

Malheureusement, rien n'interdit d'écrire également :

```
public class BoxDemo2 {

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

[5]Nous nous inspirons ici très fortement d'un des tutoriaux de Sun, `http://java.sun.com/docs/books/tutorial/java/generics/generics.html`.

```
    // ONLY place Integer objects into this box!
    Box integerBox = new Box();

    // Imagine this is one part of a large application
    // modified by one programmer.
    integerBox.add("10"); // note how the type is now String

    // ... and this is another, perhaps written
    // by a different programmer
    Integer someInteger = (Integer)integerBox.get();
    System.out.println(someInteger);
  }
}
```

qui va se compiler correctement (car `String` est sous-classe d'`Object` lui aussi). Mais à l'exécution :

```
Exception in thread "main"
       java.lang.ClassCastException:
          java.lang.String cannot be cast to java.lang.Integer
          at BoxDemo2.main(BoxDemo2.java:6)
```

Java implante un type de polymorphisme qui nous protège contre ce genre d'erreur :

```
public class Box<T> {

  private T t; // T stands for "Type"

  public void add(T t) {
    this.t = t;
  }

  public T get() {
    return t;
  }
}
```

qui permet d'écrire :

```
    public class BoxDemo3 {

        public static void main(String[] args) {
            Box<Integer> integerBox = new Box<Integer>();
            integerBox.add(new Integer(10));
            Integer someInteger = integerBox.get(); // no cast!
```

```
            System.out.println(someInteger);
        }
    }
```

Remarquez la syntaxe

```
Box<Integer> integerBox = new Box<Integer>();
```

qui permet de définir une boîte d'`Integer`. En cas d'écriture erronnée, on obtiendra une erreur à la compilation :

```
BoxDemo3.java:5: add(java.lang.Integer) in Box<java.lang.Integer>

cannot be applied to (java.lang.String)
    integerBox.add("10");
               ^

1 error
```

Le symbole `T` est une sorte de paramètre de type, qui n'apparaît pas sous forme de `T.class` à aucun endroit.

On peut définir des boîtes à types multiples simplement en utilisant des symboles différents : `Box<T,U>` .

Par convention, on utilise des symboles mono-caractères en majuscule suivant le tableau

```
    * E - Element (used extensively by the Java Collections
                   Framework)
    * K - Key
    * N - Number
    * T - Type
    * V - Value
    * S,U,V etc. - 2nd, 3rd, 4th types
```

On peut également écrire des interfaces génériques, comme :

```
public class LinkedList<E> ... {
  void add(int i, E x) { ... }
  E get(int i) { ... }
  E getFirst() { ... }
  ListIterator<E> listIterator(int i) { ... }
}
public interface ListIterator<E>
        extends Iterator<E> ... {
  boolean hasNext();
  E next();
}
```

Par exemple, une liste d'entiers sera simplement utilisée de la façon suivante :

```
LinkedList<Integer> L = new LinkedList<Integer>();
L.add(1);
```

Signalons un inconvénient (expliqué dans la documentation de Sun). On ne peut écrire `E x = new E();` ou `E[] t = new E[10];` pour des raisons de typage. Cela nous conduira à quelques contorsions plus loin, mais ces classes sont très faciles à utiliser malgré tout.

1.4.1 Restreindre à certains types

Comment empêcher l'utilisation d'un type trop général ? Il suffit de donner une *borne supérieure* au type utilisable, comme dans

```
public class Box<T extends Number> {
```

On ne pourra alors utiliser la boîte que pour des sous-classes de la classe `Number`. On aurait la même syntaxe pour des interfaces.