

Traducteur : [Daniel Le Berre](#)

07.05.2001 - Version 2.1 :

- Mise en forme html (armel).

27.01.2001 - Version 2.0 :

- Problèmes :

Class literal ? Regular classes ? Reflection ? Profiler ?

## 12: Identification dynamique de type

Le principe de l'identification dynamique de type (*Run-Time Type Identification, RTTI*) semble très simple à première vue : connaître le type exact d'un objet à partir d'une simple référence sur un type de base.

Néanmoins, le besoin de RTTI dévoile une pléthore de problèmes intéressants (et souvent complexes) en conception orientée objet, et renforce la question fondamentale de comment structurer ses programmes.

Ce chapitre indique de quelle manière Java permet de découvrir dynamiquement des informations sur les objets et les classes. On le retrouve sous deux formes : le RTTI « classique », qui suppose que tous les types sont disponibles à la compilation et à l'exécution, et le mécanisme de « réflexion », qui permet de découvrir des informations sur les classes uniquement à l'exécution. Le RTTI « classique » sera traité en premier, suivi par une discussion sur la réflexion.

### Le besoin de RTTI

Revenons à notre exemple d'une hiérarchie de classes utilisant le polymorphisme. Le type générique est la classe de base **Forme**, et les types spécifiques dérivés sont **Cercle**, **Carre** et **Triangle** :



C'est un diagramme de classe hiérarchique classique, avec la classe de base en haut et les classes dérivées qui en découlent. Le but usuel de la programmation orientée objet est de manipuler dans la majorité du code des références sur le type de base (**Forme**, ici), tel que si vous décidez de créer une nouvelle classe (**Rhomböide**, dérivée de **Forme**, par exemple), ce code restera inchangé. Dans notre exemple, la méthode liée dynamiquement dans l'interface **Forme** est **draw()**, ceci dans le but que le programmeur appelle **draw()** à partir d'une référence sur un objet de type **Forme**. **draw()** est redéfinie dans toutes les classes dérivées, et parce que cette méthode est liée dynamiquement, le comportement attendu arrivera même si l'appel se fait à partir d'une référence générique sur **Forme**. C'est ce que l'on appelle le polymorphisme.

Ainsi, on peut créer un objet spécifique (**Cercle**, **Carre** ou **Triangle**), le transtyper à **Forme** (oubliant le type spécifique de l'objet), et utiliser une référence anonyme à **Forme** dans le reste du programme.

Pour avoir un bref aperçu du polymorphisme et du transtypage ascendant (*upcast*), vous pouvez coder l'exemple ci-dessous :

```
//: c12:Formes.java
```

```
import java.util.*;

class Forme {
    void draw() {
        System.out.println(this + ".draw()");
    }
}

class Cercle extends Forme {
    public String toString() { return "Cercle"; }
}

class Carre extends Forme {
    public String toString() { return "Carre"; }
}

class Triangle extends Forme {
    public String toString() { return "Triangle"; }
}

public class Formes {
    public static void main(String[] args) {
        ArrayList s = new ArrayList();
        s.add(new Cercle());
        s.add(new Carre());
        s.add(new Triangle());
        Iterator e = s.iterator();
        while(e.hasNext())
            ((Shape)e.next()).draw();
    }
} ///:~
```

La classe de base contient une méthode **draw()** qui utilise indirectement **toString()** pour afficher un identifiant de la classe en utilisant **this** en paramètre de **System.out.println()**. Si cette fonction rencontre un objet, elle appelle automatiquement la méthode **toString()** de cet objet pour en avoir une représentation sous forme de

chaîne de caractères.

Chacune des classes dérivées redéfinit la méthode **toString()** (de la classe **Object**) pour que **draw()** affiche quelque chose de différent dans chaque cas. Dans **main()**, des types spécifiques de **Forme** sont créés et ajoutés dans un **ArrayList**. C'est à ce niveau que le transtypage ascendant intervient car un **ArrayList** contient uniquement des **Objects**. Comme tout en Java (à l'exception des types primitifs) est **Object**, un **ArrayList** peut aussi contenir des **Formes**. Mais lors du transtypage en **Object**, il perd toutes les informations spécifiques des objets, par exemple que ce sont des **Formes**. Pour le **ArrayList**, ce sont juste des **Objects**.

Lorsqu'on récupère ensuite un élément de l'**ArrayList** avec la méthode **next()**, les choses se corsent un peu. Comme un **ArrayList** contient uniquement des **Objects**, **next()** va naturellement renvoyer une référence sur un **Object**. Mais nous savons que c'est en réalité une référence sur une **Forme**, et nous désirons envoyer des messages de **Forme** à cet objet. Donc un transtypage en **Forme** est nécessaire en utilisant le transtypage habituel « (**Forme**) ». C'est la forme la plus simple de RTTI, puisqu'en Java l'exactitude de tous les typages est vérifiée à l'exécution. C'est exactement ce que signifie RTTI : à l'exécution, le type de tout objet est connu.

Dans notre cas, le RTTI est seulement partiel : l'**Object** est transtypé en **Forme**, mais pas en **Cercle**, **Carre** ou **Triangle**. Ceci parce que la seule chose que nous *savons* à ce moment là est que l'**ArrayList** est rempli de **Formes**. A la compilation, ceci est garanti uniquement par vos propres choix (ndt : le compilateur vous fait confiance), tandis qu'à l'exécution le transtypage est effectivement vérifié.

Maintenant le polymorphisme s'applique et la méthode exacte qui a été appelée pour une **Forme** est déterminée selon que la référence est de type **Cercle**, **Carre** ou **Triangle**. Et en général, c'est comme cela qu'il faut faire ; on veut que la plus grosse partie du code ignore autant que possible le type spécifique des objets, et manipule une représentation générale de cette famille d'objets (dans notre cas, **Forme**). Il en résulte un code plus facile à écrire, lire et maintenir, et vos conceptions seront plus faciles à implémenter, comprendre et modifier. Le polymorphisme est donc un but général en programmation orientée objet.

Mais que faire si vous avez un problème de programmation qui peut se résoudre facilement si vous connaissez le type exact de la référence générique que vous manipulez ? Par exemple, supposons que vous désiriez permettre à vos utilisateurs de colorier toutes les formes d'un type particulier en violet. De cette manière, ils peuvent retrouver tous les triangles à l'écran en les coloriant. C'est ce que fait le RTTI : on peut demander à une référence sur une **Forme** le type exact de l'objet référencé.

## L'objet Class

Pour comprendre comment marche le RTTI en Java, il faut d'abord savoir comment est représentée l'information sur le type durant l'exécution. C'est le rôle d'un objet spécifique appelé *l'objet Class*, qui contient toutes les informations relative à la classe (on l'appelle parfois *meta-class*). En fait, l'objet **Class** est utilisé pour créer tous les objets « habituels » d'une classe.

Il y a un objet **Class** pour chacune des classes d'un programme. Ainsi, à chaque fois qu'une classe est écrite et compilée, un unique objet de type **Class** est aussi créé (et rangé, le plus souvent, dans un fichier **.class** du même nom). Durant l'exécution, lorsqu'un nouvel objet de cette classe doit être créé, la Machine Virtuelle Java (*Java Virtual Machine, JVM*) qui exécute le programme vérifie d'abord si l'objet **Class** associé est déjà chargé. Si non, la JVM le charge en cherchant un fichier **.class** du même nom. Ainsi un programme Java n'est pas totalement chargé en mémoire lorsqu'il démarre, contrairement à beaucoup de langages classiques.

Une fois que l'objet **Class** est en mémoire, il est utilisé pour créer tous les objets de ce type.

Si cela ne vous semble pas clair ou si vous ne le croyez pas, voici un programme pour le prouver :

```
//: c12:Confiseur.java
// Étude du fonctionnement du chargeur de classes.
```

```
class Bonbon {
    static {
        System.out.println("Charge Bonbon");
    }
}
```

```
class Gomme {
    static {
        System.out.println("Charge Gomme");
    }
}
```

```
class Biscuit {
    static {
        System.out.println("Charge Biscuit");
    }
}
```

```
public class Confiseur {
    public static void main(String[] args) {
        System.out.println("Début méthode main");
        new Bonbon();
        System.out.println("Après création Gomme");
        try {
            Class.forName("Gomme");
        } catch(ClassNotFoundException e) {
            e.printStackTrace(System.err);
        }
        System.out.println(
            "Après Class.forName(\"Gomme\")");
    }
}
```

```
        new Biscuit();

        System.out.println("Après création Biscuit");
    }
} ///:~
```

Chacune des classes **Bonbon**, **Gomme** et **Biscuit** a une clause **static** qui est exécutée lorsque la classe est chargée la première fois. L'information qui est affichée vous permet de savoir quand cette classe est chargée. Dans la méthode **main()**, la création des objets est dispersée entre des opérations d'affichages pour faciliter la détection du moment du chargement.

Une ligne particulièrement intéressante est :

```
Class.forName("Gomme");
```

Cette méthode est une méthode **static** de **Class** (qui appartient à tous les objets **Class**). Un objet **Class** est comme tous les autres objets, il est donc possible d'obtenir sa référence et de la manipuler (c'est ce que fait le chargeur de classes). Un des moyens d'obtenir une référence sur un objet **Class** est la méthode **forName()**, qui prend en paramètre une chaîne de caractères contenant le nom (attention à l'orthographe et aux majuscules !) de la classe dont vous voulez la référence. Elle retourne une référence sur un objet **Class**.

Le résultat de ce programme pour une JVM est :

```
Début méthode main
Charge Bonbon
Après création Bonbon
Charge Gomme
Après Class.forName("Gomme")
Charge Biscuit
Après création Biscuit
```

On peut noter que chaque objet **Class** est chargé uniquement lorsque c'est nécessaire, et que l'initialisation **static** est effectuée au chargement de la classe.

## Les littéraux Class

Java fournit une deuxième manière d'obtenir une référence sur un objet de type **Class**, en utilisant *le littéral class*. Dans le programme précédent, on aurait par exemple :

```
Gomme.class;
```

ce qui n'est pas seulement plus simple, mais aussi plus sûr puisque vérifié à la compilation. Comme elle ne nécessite pas d'appel à une méthode, elle est aussi plus efficace.

Les littéraux `Class` sont utilisables sur les classes habituelles ainsi que sur les interfaces, les tableaux et les types primitifs. De plus, il y a un attribut standard appelé **TYPE** qui existe pour chacune des classes englobant des types primitifs. L'attribut **TYPE** produit une référence à l'objet **Class** associé au type primitif, tel que :

... est équivalent à ...	
<code>boolean.class</code>	<code>Boolean.TYPE</code>
<code>char.class</code>	<code>Character.TYPE</code>
<code>byte.class</code>	<code>Byte.TYPE</code>
<code>short.class</code>	<code>Short.TYPE</code>
<code>int.class</code>	<code>Integer.TYPE</code>
<code>long.class</code>	<code>Long.TYPE</code>
<code>float.class</code>	<code>Float.TYPE</code>
<code>double.class</code>	<code>Double.TYPE</code>
<code>void.class</code>	<code>Void.TYPE</code>

Ma préférence va à l'utilisation des « `.class` » si possible, car cela est plus consistant avec les classes habituelles.

## Vérifier avant de transtyper

Jusqu'à présent, nous avons vu différentes utilisations de RTTI dont :

1. Le transtypage classique ; i.e. « **(Forme)** », qui utilise RTTI pour être sûr que le transtypage est correct et lancer une **ClassCastException** si un mauvais transtypage est effectué.
2. L'objet **Class** qui représente le type d'un objet. L'objet **Class** peut être interrogé afin d'obtenir des informations utiles durant l'exécution.

En C++, le transtypage classique « **(Forme)** » *n'effectue pas* de RTTI. Il indique seulement au compilateur de traiter l'objet avec le nouveau type. En Java, qui effectue cette vérification de type, ce transtypage est souvent appelé « transtypage descendant sain ». La raison du terme « descendant » est liée à l'historique de la représentation des diagrammes de hiérarchie de classes. Si transtyper un **Cercle** en une **Forme** est un transtypage ascendant, alors transtyper une **Forme** en un **Cercle** est un transtypage descendant. Néanmoins, on sait que tout **Cercle** est aussi une **Forme**, et le compilateur nous laisse donc librement effectuer un transtypage descendant ; par contre toute **Forme** *n'est pas nécessairement* un **Cercle**, le compilateur ne permet donc pas de faire un transtypage descendant sans utiliser un transtypage explicite.

Il existe une troisième forme de RTTI en Java. C'est le mot clef **instanceof** qui vous indique si un objet est d'un type particulier. Il retourne un **boolean** afin d'être utilisé sous la forme d'une question, telle que :

```
if(x instanceof Chien)
    ((Chien)x).aboyer();
```

L'expression ci-dessus vérifie si un objet **x** appartient à la classe **Chien** *avant* de transtyper **x** en **Chien**. Il est important d'utiliser **instanceof** avant un transtypage descendant lorsque vous n'avez pas d'autres informations vous indiquant le type de l'objet, sinon vous risquez d'obtenir une **ClassCastException**.

Le plus souvent, vous rechercherez un type d'objets (les triangles à peindre en violet par exemple), mais vous pouvez aisément identifier tous les objets en utilisant **instanceof**. Supposons que vous ayez une famille de classes d'animaux de compagnie (**Pet**) :

```
//: c12:Pets.java

class Pet {}

class Chien extends Pet {}

class Carlin extends Chien {}

class Chat extends Pet {}

class Rongeur extends Pet {}

class Gerbil extends Rongeur {}

class Hamster extends Rongeur {}


class Counter { int i; } ///:~
```

La classe **Counter** est utilisée pour compter le nombre d'animaux de compagnie de chaque type. On peut le voir comme un objet **Integer** que l'on peut modifier.

En utilisant **instanceof**, tous les animaux peuvent être comptés :

```
//: c12:PetCount.java

// Utiliser instanceof.

import java.util.*;

public class PetCount {

    static String[] typenames = {

        "Pet", "Chien", "Carlin", "Chat",

        "Rongeur", "Gerbil", "Hamster",

    };

    // Les exceptions remontent jusqu'à la console :

    public static void main(String[] args)

    throws Exception {

        ArrayList pets = new ArrayList();

        try {

            Class[] petTypes = {

                Class.forName("Chien"),

                Class.forName("Carlin"),
```

```
        Class.forName("Chat"),
        Class.forName("Rongeur"),
        Class.forName("Gerbil"),
        Class.forName("Hamster"),
    };

    for(int i = 0; i < 15; i++)
        pets.add(
            petTypes[
                (int)(Math.random()*petTypes.length)]
                .newInstance());
    } catch(InstantiationException e) {
        System.err.println("Instantiation impossible");
        throw e;
    } catch(IllegalAccessException e) {
        System.err.println("Accès impossible");
        throw e;
    } catch(ClassNotFoundException e) {
        System.err.println("Classe non trouvée");
        throw e;
    }
}

HashMap h = new HashMap();
for(int i = 0; i < typenames.length; i++)
    h.put(typenames[i], new Counter());
for(int i = 0; i < pets.size(); i++) {
    Object o = pets.get(i);
    if(o instanceof Pet)
        ((Counter)h.get("Pet")).i++;
    if(o instanceof Chien)
        ((Counter)h.get("Chien")).i++;
    if(o instanceof Carlin)
        ((Counter)h.get("Carlin")).i++;
    if(o instanceof Chat)
        ((Counter)h.get("Chat")).i++;
    if(o instanceof Rongeur)
```



```

        ((Counter)h.get("Rongeur")).i++;
    if(o instanceof Gerbil)
        ((Counter)h.get("Gerbil")).i++;
    if(o instanceof Hamster)
        ((Counter)h.get("Hamster")).i++;
}

for(int i = 0; i < pets.size(); i++)
    System.out.println(pets.get(i).getClass());

for(int i = 0; i < typenames.length; i++)
    System.out.println(
        typenames[i] + " quantité : " +
        ((Counter)h.get(typenames[i])).i);
}
} ///:~

```

Il existe cependant une limitation concernant **instanceof** : un objet peut être comparé à un identifiant de type, pas à un objet **Class**. Dans l'exemple précédent, vous pouvez trouver fastidieux d'écrire tous ces **instanceof**, et vous avez raison. Mais il n'y a pas moyen d'automatiser intelligemment le **instanceof** en créant un **ArrayList** d'objets **Class** et en comparant l'objet à ces derniers (nous verrons plus loin une alternative). Vous pouvez penser que ce n'est pas une très grande restriction, car vous finirez par comprendre que votre conception est mauvaise si vous finissez par écrire un grand nombre de **instanceof**.

Bien sûr, cet exemple est imaginaire - vous utiliseriez probablement un attribut de classe (**static**) pour chaque type que vous incrémenteriez dans le constructeur pour mettre à jour les compteurs. Vous feriez cela *si* vous avez accès au code source de ces classes et pouvez le modifier. Comme ce n'est pas toujours le cas, le RTTI est bien pratique.

## Utiliser les littéraux de classe

Il est intéressant de voir comment l'exemple précédent **PetCount.java** peut être réécrit en utilisant les littéraux de classe. Le résultat est plus satisfaisant sur bien des points :

```

///: c12:PetCount2.java

// Utiliser les littéraux de classe.

import java.util.*;

public class PetCount2 {

    public static void main(String[] args)

        throws Exception {

```

```
ArrayList pets = new ArrayList();

Class[] petTypes = {

    // Littéraux de classe:

    Pet.class,

    Chien.class,

    Carlin.class,

    Chat.class,

    Rongeur.class,

    Gerbil.class,

    Hamster.class,

};

try {

    for(int i = 0; i < 15; i++) {

        // on ajoute 1 pour éliminer Pet.class:

        int rnd = 1 + (int)(

            Math.random() * (petTypes.length - 1));

        pets.add(

            petTypes[rnd].newInstance());

    }

} catch(InstantiationException e) {

    System.err.println("Instantiation impossible");

    throw e;

} catch(IllegalAccessException e) {

    System.err.println("Accès impossible");

    throw e;

}

HashMap h = new HashMap();

for(int i = 0; i < petTypes.length; i++)

    h.put(petTypes[i].toString(),

        new Counter());

for(int i = 0; i < pets.size(); i++) {

    Object o = pets.get(i);

    if(o instanceof Pet)

        ((Counter)h.get("class Pet")).i++;

}
```

```

        if(o instanceof Chien)
            ((Counter)h.get("class Chien")).i++;
        if(o instanceof Carlin)
            ((Counter)h.get("class Carlin")).i++;
        if(o instanceof Chat)
            ((Counter)h.get("class Chat")).i++;
        if(o instanceof Rongeur)
            ((Counter)h.get("class Rongeur")).i++;
        if(o instanceof Gerbil)
            ((Counter)h.get("class Gerbil")).i++;
        if(o instanceof Hamster)
            ((Counter)h.get("class Hamster")).i++;
    }
    for(int i = 0; i < pets.size(); i++)
        System.out.println(pets.get(i).getClass());
    Iterator keys = h.keySet().iterator();
    while(keys.hasNext()) {
        String nm = (String)keys.next();
        Counter cnt = (Counter)h.get(nm);
        System.out.println(
            nm.substring(nm.lastIndexOf('.') + 1) +
            " quantité: " + cnt.i);
    }
}
} ///:~

```

Ici, le tableau **typenames** a été enlevé, on préfère obtenir de l'objet **Class** les chaînes identifiant les types. Notons que ce système permet au besoin de différencier classes et interfaces.

On peut aussi remarquer que la création de **petTypes** ne nécessite pas l'utilisation d'un block **try** puisqu'il est évalué à la compilation et ne lancera donc aucune exception, contrairement à **Class.forName()**.

Quand les objets **Pet** sont créés dynamiquement, vous pouvez voir que le nombre aléatoire généré est compris entre un (ndt inclus) et **petTypes.length** (ndt exclus), donc ne peut pas prendre la valeur zéro. C'est parce que zéro réfère à **Pet.class**, et que nous supposons que créer un objet générique **Pet** n'est pas intéressant. Cependant, comme **Pet.class** fait partie de **petTypes**, le nombre total d'animaux familiers est compté.

## Un instanceof dynamique

La méthode **isInstance** de **Class** fournit un moyen d'appeler dynamiquement l'opérateur **instanceof**. Ainsi, toutes ces ennuyeuses expressions **instanceof** peuvent être supprimées de l'exemple **PetCount** :

```
///  
// Utiliser isInstance().  
import java.util.*;  
  
public class PetCount3 {  
    public static void main(String[] args)  
        throws Exception {  
        ArrayList pets = new ArrayList();  
        Class[] petTypes = {  
            Pet.class,  
            Chien.class,  
            Carlin.class,  
            Chat.class,  
            Rongeur.class,  
            Gerbil.class,  
            Hamster.class,  
        };  
        try {  
            for(int i = 0; i < 15; i++) {  
                // Ajoute 1 pour éliminer Pet.class:  
                int rnd = 1 + (int)(  
                    Math.random() * (petTypes.length - 1));  
                pets.add(  
                    petTypes[rnd].newInstance());  
            }  
        } catch(InstantiationException e) {  
            System.err.println("Instantiation impossible");  
            throw e;  
        } catch(IllegalAccessException e) {  
            System.err.println("Accès impossible");  
            throw e;  
        }  
    }  
}
```

```

HashMap h = new HashMap();

for(int i = 0; i < petTypes.length; i++)

    h.put(petTypes[i].toString(),

        new Counter());

for(int i = 0; i < pets.size(); i++) {

    Object o = pets.get(i);

    // Utiliser instanceof pour automatiser
    // l'utilisation des instanceof :
    // Ndt: Pourquoi ce ++j ???
    for (int j = 0; j < petTypes.length; ++j)

        if (petTypes[j].isInstance(o)) {

            String key = petTypes[j].toString();

            ((Counter)h.get(key)).i++;

        }

    }

for(int i = 0; i < pets.size(); i++)

    System.out.println(pets.get(i).getClass());

Iterator keys = h.keySet().iterator();

while(keys.hasNext()) {

    String nm = (String)keys.next();

    Counter cnt = (Counter)h.get(nm);

    System.out.println(

        nm.substring(nm.lastIndexOf('.') + 1) +

        " quantity: " + cnt.i);

    }

}

} ///:~

```

On peut noter que l'utilisation de la méthode **isInstance()** a permis d'éliminer les expressions **instanceof**. De plus, cela signifie que de nouveaux types d'animaux familiers peuvent être ajoutés simplement en modifiant le tableau **petTypes** ; le reste du programme reste inchangé (ce qui n'est pas le cas lorsqu'on utilise des **instanceof**).

## instanceof vs. équivalence de classe

Lorsque vous demandez une information de type, il y a une différence importante entre l'utilisation d'une forme de **instanceof** (**instanceof** ou **isInstance()**, qui produisent des résultats équivalents) et la comparaison directe

des objets **Class**. Voici un exemple qui illustre cette différence :

```
//: cl2:FamilyVsExactType.java
// La différence entre instanceof et class

class Base {}

class Derived extends Base {}

public class FamilyVsExactType {
    static void test(Object x) {
        System.out.println("Teste x de type " +
            x.getClass());
        System.out.println("x instanceof Base " +
            (x instanceof Base));
        System.out.println("x instanceof Derived " +
            (x instanceof Derived));
        System.out.println("Base.isInstance(x) " +
            Base.class.isInstance(x));
        System.out.println("Derived.isInstance(x) " +
            Derived.class.isInstance(x));
        System.out.println(
            "x.getClass() == Base.class " +
            (x.getClass() == Base.class));
        System.out.println(
            "x.getClass() == Derived.class " +
            (x.getClass() == Derived.class));
        System.out.println(
            "x.getClass().equals(Base.class) " +
            (x.getClass().equals(Base.class)));
        System.out.println(
            "x.getClass().equals(Derived.class) " +
            (x.getClass().equals(Derived.class)));
    }

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

```

    test(new Base());

    test(new Derived());
}

} ///:~

```

La méthode **test()** effectue une vérification du type de son argument en utilisant les deux formes de **instanceof**. Elle récupère ensuite la référence sur l'objet **Class** et utilise **==** et **equals()** pour tester l'égalité entre les objets **Class**. Le résultat est le suivant :

```

Teste x de type class Base
x instanceof Base true
x instanceof Derived false
Base.isInstance(x) true
Derived.isInstance(x) false
x.getClass() == Base.class true
x.getClass() == Derived.class false
x.getClass().equals(Base.class) true
x.getClass().equals(Derived.class) false

Teste x de type class Derived
x instanceof Base true
x instanceof Derived true
Base.isInstance(x) true
Derived.isInstance(x) true
x.getClass() == Base.class false
x.getClass() == Derived.class true
x.getClass().equals(Base.class) false
x.getClass().equals(Derived.class) true

```

Il est rassurant de constater que **instanceof** et **isInstance()** produisent des résultats identiques, de même que **equals()** et **==**. Mais les tests eux-mêmes aboutissent à des conclusions différentes. **instanceof** teste le concept de type et signifie « es-tu de cette classe, ou d'une classe dérivée ? ». Autrement, si on compare les objets **Class** en utilisant **==**, il n'est plus question d'héritage - l'objet est de ce type ou non.

## La syntaxe du RTTI

Java effectue son identification dynamique de type (RTTI) à l'aide de l'objet **Class**, même lors d'un transtypage. La classe **Class** dispose aussi de nombreuses autres manières d'être utilisée pour le RTTI.

Premièrement, il faut obtenir une référence sur l'objet **Class** approprié. Une manière de le faire, comme nous

l'avons vu dans l'exemple précédent, est d'utiliser une chaîne de caractères et la méthode **Class.forName()**. C'est très pratique car il n'est pas nécessaire d'avoir un objet de ce type pour obtenir la référence sur l'objet **Class**. Néanmoins, si vous avez déjà un objet de ce type, vous pouvez retrouver la référence à l'objet **Class** en appelant une méthode qui appartient à la classe racine **Object** : **getClass()**. Elle retourne une référence sur l'objet **Class** représentant le type actuel de l'objet. **Class** a de nombreuses méthodes intéressantes, comme le montre l'exemple suivant :

```
///  
// c12:ToyTest.java  
// Teste la classe Class.  
  
interface HasBatteries {}  
interface Waterproof {}  
interface ShootsThings {}  
class Toy {  
    // Commenter le constructeur par  
    // défaut suivant pour obtenir  
    // NoSuchMethodError depuis (*1*)  
    Toy() {}  
    Toy(int i) {}  
}  
  
class FancyToy extends Toy  
    implements HasBatteries,  
        Waterproof, ShootsThings {  
    FancyToy() { super(1); }  
}  
  
public class ToyTest {  
    public static void main(String[] args)  
        throws Exception {  
        Class c = null;  
        try {  
            c = Class.forName("FancyToy");  
        } catch(ClassNotFoundException e) {  
            System.err.println("Ne trouve pas FancyToy");  
            throw e;  
        }  
    }  
}
```



```

    }

    printInfo(c);

    Class[] faces = c.getInterfaces();

    for(int i = 0; i < faces.length; i++)

        printInfo(faces[i]);

    Class cy = c.getSuperclass();

    Object o = null;

    try {

        // Nécessite un constructeur par défaut :

        o = cy.newInstance(); // (*!*)

    } catch(InstantiationException e) {

        System.err.println("Instanciación impossible");

        throw e;

    } catch(IllegalAccessException e) {

        System.err.println("Accès impossible");

        throw e;

    }

    printInfo(o.getClass());

}

static void printInfo(Class cc) {

    System.out.println(

        "Class nom: " + cc.getName() +

        " est une interface ? [" +

        cc.isInterface() + "]" );

}

} ///:~

```

On peut voir que la classe **FancyToy** est assez compliquée, puisqu'elle hérite de **Toy** et implémente les interfaces **HasBatteries**, **Waterproof** et **ShootThings**. Dans **main()**, une référence de **Class** est créée et initialisée pour la classe **FancyToy** en utilisant **forName()** à l'intérieur du block **try** approprié.

La méthode **Class.getInterfaces()** retourne un tableau d'objets **Class** représentant les interfaces qui sont contenues dans l'objet en question.

Si vous avez un objet **Class**, vous pouvez aussi lui demander la classe dont il hérite directement en utilisant la méthode **getSuperclass()**. Celle-ci retourne, bien sûr, une référence de **Class** que vous pouvez interroger plus en détail. Cela signifie qu'à l'exécution, vous pouvez découvrir la hiérarchie de classe complète d'un objet.

La méthode **newInstance()** de **Class** peut, au premier abord, ressembler à un autre moyen de **cloner()** un objet. Néanmoins, vous pouvez créer un nouvel objet avec **newInstance()** sans un objet existant, comme nous le voyons ici, car il n'y a pas d'objets **Toy** - seulement **cy** qui est une référence sur l'objet **Class** de **y**. C'est un moyen de construire un « constructeur virtuel », qui vous permet d'exprimer « je ne sais pas exactement de quel type vous êtes, mais créez-vous proprement ». Dans l'exemple ci-dessus, **cy** est seulement une référence sur **Class** sans aucune autre information à la compilation. Et lorsque vous créez une nouvelle instance, vous obtenez une référence sur un **Object**. Mais cette référence pointe sur un objet **Toy**. Bien entendu, avant de pouvoir envoyer d'autres messages que ceux acceptés par **Object**, vous devez l'examiner un peu plus et effectuer quelques transtypes. De plus, la classe de l'objet créé par **newInstance()** doit avoir un constructeur par défaut. Dans la prochaine section, nous verrons comment créer dynamiquement des objets de classes utilisant n'importe quel constructeur, avec l'API de réflexion Java.

La dernière méthode dans le listing est **printInfo()**, qui prend en paramètre une référence sur **Class**, récupère son nom avec **getName()**, et détermine si c'est une interface avec **isInterface()**. Le résultat de ce programme est :

```
Class nom: FancyToy est une interface ? [false]
Class nom: HasBatteries est une interface ? [true]
Class nom: Waterproof est une interface ? [true]
Class nom: ShootsThings est une interface ? [true]
Class nom: Toy est une interface ? [false]
```

Ainsi, avec l'objet **Class**, vous pouvez découvrir vraiment tout ce que vous voulez savoir sur un objet.

## Réflexion : information de classe dynamique

Si vous ne connaissez pas le type précis d'un objet, le RTTI vous le dira. Néanmoins, il y a une limitation : le type doit être connu à la compilation afin que vous puissiez le détecter en utilisant le RTTI et faire quelque chose d'intéressant avec cette information. Autrement dit, le compilateur doit connaître toutes les classes que vous utilisez pour le RTTI.

Ceci peut ne pas paraître une grande limitation à première vue, mais supposons que l'on vous donne une référence sur un objet qui n'est pas dans l'espace de votre programme. En fait, la classe de l'objet n'est même pas disponible lors de la compilation. Par exemple, supposons que vous récupériez un paquet d'octets à partir d'un fichier sur disque ou via une connexion réseau et que l'on vous dise que ces octets représentent une classe. Puisque le compilateur ne peut pas connaître la classe lorsqu'il compile le code, comment pouvez-vous utiliser cette classe ?

Dans un environnement de travail traditionnel cela peut sembler un scénario improbable. Mais dès que l'on se déplace dans un monde de la programmation plus vaste, il y a des cas importants dans lesquels cela arrive. Le premier est la programmation par composants, dans lequel vous construisez vos projets en utilisant le *Rapid Application Development* (RAD) dans un constructeur d'application. C'est une approche visuelle pour créer un programme (que vous voyez à l'écran comme un « formulaire » (*form*)) en déplaçant des icônes qui représentent des composants dans le formulaire. Ces composants sont alors configurés en fixant certaines de leurs valeurs. Cette configuration durant la conception nécessite que chacun des composants soit instanciable, qu'il dévoile une partie de lui-même et qu'il permette que ses valeurs soient lues et fixées. De plus, les composants qui gèrent des événements dans une GUI doivent dévoiler des informations à propos des méthodes appropriées pour que

l'environnement RAD puisse aider le programmeur à redéfinir ces méthodes de gestion d'événements. La réflexion fournit le mécanisme pour détecter les méthodes disponibles et produire leurs noms. Java fournit une structure de programmation par composants au travers de JavaBeans (décrit dans le chapitre 13).

Une autre motivation pour découvrir dynamiquement des informations sur une classe est de permettre la création et l'exécution d'objets sur des plates forme distantes via un réseau. Ceci est appelé *Remote Method Invocation* (RMI) et permet à un programme Java d'avoir des objets répartis sur plusieurs machines. Cette répartition peut survenir pour diverses raisons : par exemple, vous effectuez une tâche coûteuse en calcul et vous voulez la dissocier pour utiliser certains morceaux sur des machines libres afin d'accélérer les choses. Dans certaines situations vous aimeriez placer le code d'un certain type de tâches (par exemple « Business Rules » dans une architecture client/serveur multitiers ?????) sur une machine particulière, afin que celle-ci devienne le dépositaire unique de ces tâches, pouvant être facilement modifié pour changer tout le système (c'est un développement intéressant puisque la machine existe uniquement pour faciliter les changements logiciels !). Enfin, le calcul distribué supporte aussi des architectures spécialisées qui peuvent être appropriées à des tâches spécifiques- l'inversion de matrices, par exemple- mais inappropriées ou trop chères pour la programmation classique.

La classe **Class** (décrite précédemment dans ce chapitre) supporte le concept de *réflexion*, et une bibliothèque additionnelle, **java.lang.reflect**, contenant les classes **Field**, **Method**, et **Constructor** (chacune implémentant l'interface **Member**). Les objets de ce type sont créés dynamiquement par la JVM pour représenter les membres correspondants d'une classe inconnue. On peut alors utiliser les constructeurs pour créer de nouveaux objets, les méthodes **get()** et **set()** pour lire et modifier les champs associés à des objets **Field**, et la méthode **invoke()** pour appeler une méthode associée à un objet **Method**. De plus, on peut utiliser les méthodes très pratiques **getFields()**, **getMethods()**, **getConstructors()**, etc. retournant un tableau représentant respectivement des champs, méthodes et constructeurs (pour en savoir plus, jetez un oeil à la documentation en ligne de la classe **Class**). Ainsi, l'information sur la classe d'objets inconnus peut être totalement déterminée dynamiquement, sans rien en savoir à la compilation.

Il est important de noter qu'il n'y a rien de magique dans la réflexion. Quand vous utilisez la réflexion pour interagir avec des objets de type inconnu, la JVM va simplement regarder l'objet et voir qu'il appartient à une classe particulière (comme une RTTI ordinaire) mais, avant toute autre chose, l'objet **Class** doit être chargé. Le fichier **.class** pour ce type particulier doit donc être disponible pour la JVM, soit localement sur la machine ou via le réseau. La vraie différence entre le RTTI et la réflexion est donc qu'avec le RTTI, le compilateur ouvre et examine le fichier **.class** à la compilation. Dit autrement, vous pouvez appeler toutes les méthodes d'un objet &ldquo;normalement&rdquo;. Avec la réflexion, le fichier **.class** n'est pas disponible à la compilation ; il est ouvert et examiné à l'exécution.

## Un extracteur de méthodes de classe

Vous aurez rarement besoin d'utiliser directement les outils de réflexion ; ils sont utilisés pour supporter d'autres caractéristiques de Java, telles que la sérialisation (Chapitre 11), JavaBeans (Chapitre 13) et RMI (Chapitre 15). Néanmoins, il est quelquefois utile d'extraire dynamiquement des informations sur une classe. Un outil très utile est un extracteur de méthode de classe. Comme mentionné précédemment, chercher le code définissant une classe ou sa documentation en ligne montre uniquement les méthodes définies ou redéfinies dans cette définition de classe. Mais il peut y en avoir des douzaines d'autre qui proviennent des classes de base. Les retrouver est fastidieux et long [60]. Heureusement, la réflexion fournit un moyen d'écrire un outil simple qui va automatiquement montrer l'interface entière. Voici comment il fonctionne :

```
///  
// Utiliser la réflexion pour montrer toutes les méthodes
```

```
// d'une classe, même si celles ci sont définies dans la
// classe de base.

import java.lang.reflect.*;

public class ShowMethods {

    static final String usage =

        "usage: \n" +

        "ShowMethods qualified.class.name\n" +

        "Pour montrer toutes les méthodes or: \n" +

        "ShowMethods qualified.class.name word\n" +

        "Pour rechercher les méthodes contenant 'word'";

    public static void main(String[] args) {

        if(args.length < 1) {

            System.out.println(usage);

            System.exit(0);

        }

        try {

            Class c = Class.forName(args[0]);

            Method[] m = c.getMethods();

            Constructor[] ctor = c.getConstructors();

            if(args.length == 1) {

                for (int i = 0; i < m.length; i++)

                    System.out.println(m[i]);

                for (int i = 0; i < ctor.length; i++)

                    System.out.println(ctor[i]);

            } else {

                for (int i = 0; i < m.length; i++)

                    if(m[i].toString()

                        .indexOf(args[1])!= -1)

                        System.out.println(m[i]);

                for (int i = 0; i < ctor.length; i++)

                    if(ctor[i].toString()

                        .indexOf(args[1])!= -1)

                        System.out.println(ctor[i]);

            }

        }

    }

}
```

```

    }
} catch(ClassNotFoundException e) {
    System.err.println("Classe non trouvée : " + e);
}
}
} ///:~

```

Les méthodes de **Class** `getMethods()` et `getConstructors()` retournent respectivement un tableau de **Method** et **Constructor**. Chacune de ces classes a de plus des méthodes pour obtenir les noms, arguments et valeur retournée des méthodes qu'elles représentent. Mais vous pouvez aussi utiliser simplement `toString()`, comme ici, pour produire une chaîne de caractères avec la signature complète de la méthode. Le reste du code sert juste pour l'extraction des informations de la ligne de commande, déterminer si une signature particulière correspond à votre chaîne cible (en utilisant `indexOf()`), et afficher le résultat.

Ceci montre la réflexion en action, puisque le résultat de `Class.forName()` ne peut pas être connu à la compilation, donc toutes les informations sur la signature des méthodes est extraite à l'exécution. Si vous étudiez la documentation en ligne sur la réflexion, vous verrez qu'il est possible de créer et d'appeler une méthode d'un objet qui était totalement inconnu lors de la compilation (nous verrons des exemples plus loin dans ce livre). Encore une fois, c'est quelque chose dont vous n'aurez peut être jamais besoin de faire vous-même- le support est là pour le RMI et la programmation par JavaBeans- mais il est intéressant.

Une expérience intéressante est de lancer :

```
java ShowMethods ShowMethods
```

Ceci produit une liste qui inclut un constructeur par défaut **public**, bien que vous puissiez voir à partir du code source qu'aucun constructeur n'ait été défini. Le constructeur que vous voyez est celui qui est automatiquement généré par le compilateur. Si vous définissez maintenant `ShowMethods` comme une classe non **public** (par exemple, amie), le constructeur par défaut n'apparaît plus dans la liste. Le constructeur pas défaut généré a automatiquement le même accès que la classe.

L'affichage de `ShowMethods` est toujours un peu ennuyeuse. Par exemple, voici une portion de l'affichage produit en invoquant `java ShowMethods java.lang.String` :

```

public boolean
    java.lang.String.startsWith(java.lang.String,int)
public boolean
    java.lang.String.startsWith(java.lang.String)
public boolean
    java.lang.String.endsWith(java.lang.String)

```

Il serait préférable que les préfixes comme `java.lang` puissent être éliminés. La classe `StreamTokenizer` introduite dans le chapitre précédent peut nous aider à créer un outil résolvant ce problème :

```
/// com:bruceeckel:util:StripQualifiers.java  
package com.bruceeckel.util;  
import java.io.*;  
  
public class StripQualifiers {  
    private StreamTokenizer st;  
    public StripQualifiers(String qualified) {  
        st = new StreamTokenizer(  
            new StringReader(qualified));  
        st.ordinaryChar(' '); // garde les espaces  
    }  
    public String getNext() {  
        String s = null;  
        try {  
            int token = st.nextToken();  
            if(token != StreamTokenizer.TT_EOF) {  
                switch(st.ttype) {  
                    case StreamTokenizer.TT_EOL:  
                        s = null;  
                        break;  
                    case StreamTokenizer.TT_NUMBER:  
                        s = Double.toString(st.nval);  
                        break;  
                    case StreamTokenizer.TT_WORD:  
                        s = new String(st.sval);  
                        break;  
                    default: //il y a un seul caractère dans ttype  
                        s = String.valueOf((char)st.ttype);  
                }  
            }  
        } catch(IOException e) {  
            System.err.println("Erreur recherche token");  
        }  
        return s;  
    }  
}
```

```

    }

    public static String strip(String qualified) {
        StripQualifiers sq =
            new StripQualifiers(qualified);
        String s = "", si;
        while((si = sq.getNext()) != null) {
            int lastDot = si.lastIndexOf('.');
            if(lastDot != -1)
                si = si.substring(lastDot + 1);
            s += si;
        }
        return s;
    }
} ///:~

```

Pour faciliter sa réutilisation, cette classe est placée dans **com.bruceeckel.util**. Comme vous pouvez le voir, elle utilise la classe **StreamTokenizer** et la manipulation des **String** pour effectuer son travail.

La nouvelle version du programme utilise la classe ci-dessus pour clarifier le résultat :

```

///: c12:ShowMethodsClean.java
// ShowMethods avec élimination des préfixes
// pour faciliter la lecture du résultat.
import java.lang.reflect.*;
import com.bruceeckel.util.*;

public class ShowMethodsClean {
    static final String usage =
        "usage: \n" +
        "ShowMethodsClean qualified.class.name\n" +
        "Pour montrer toutes les méthodes or: \n" +
        "ShowMethodsClean qualified.class.name word\n" +
        "Pour rechercher les méthodes contenant 'word'";

    public static void main(String[] args) {

```

```
if(args.length < 1) {
    System.out.println(usage);
    System.exit(0);
}

try {
    Class c = Class.forName(args[0]);
    Method[] m = c.getMethods();
    Constructor[] ctor = c.getConstructors();

    // Conversion en un tableau de chaînes simplifiées :
    String[] n =
        new String[m.length + ctor.length];
    for(int i = 0; i < m.length; i++) {
        String s = m[i].toString();
        n[i] = StripQualifiers.strip(s);
    }
    for(int i = 0; i < ctor.length; i++) {
        String s = ctor[i].toString();
        n[i + m.length] =
            StripQualifiers.strip(s);
    }
    if(args.length == 1)
        for (int i = 0; i < n.length; i++)
            System.out.println(n[i]);
    else
        for (int i = 0; i < n.length; i++)
            if(n[i].indexOf(args[1])!= -1)
                System.out.println(n[i]);
    } catch(ClassNotFoundException e) {
        System.err.println("Classe non trouvée : " + e);
    }
}

} ///:~
```

La classe **ShowMethodsClean** est semblable à la classe **ShowMethods**, excepté qu'elle transforme les tableaux de **Method** et **Constructor** en un seul tableau de **String**. Chaque **String** est ensuite appliquée à



**StripQualifiers.strip()** pour enlever les préfixes des méthodes.

Cet outil peut réellement vous faire gagner du temps lorsque vous programmez, quand vous ne vous souvenez pas si une classe a une méthode particulière et que vous ne voulez pas explorer toute sa hiérarchie dans la documentation en ligne, ou si vous ne savez pas si cette classe peut faire quelque chose avec, par exemple, un objet **Color**.

Le chapitre 13 contient une version graphique de ce programme (adapté pour extraire des informations sur les composants Swing) que vous pouvez laisser tourner pendant que vous écrivez votre code, pour des recherches rapides.

## Résumé

L'identification dynamique de type (RTTI) permet de découvrir des informations de type à partir d'une référence sur une classe de base inconnue. [Je n'arrive pas à traduire cette phrase: *Thus, it's ripe for misuse by the novice since it might make sense before polymorphic method calls do.* Prop1: Ainsi, il mûrit pour sa mauvaise utilisation par le novice puisque il pourrait être utilisé de le faire avant un appel de méthode polymorphique. Prop2 (JQ): Malheureusement ces informations peuvent conduire le novice à négliger les concepts du polymorphisme, puisqu'elles sont plus faciles à appréhender.] Pour beaucoup de gens habitués à la programmation procédurale, il est difficile de ne pas organiser leurs programmes en ensembles d'expressions **switch**. Ils pourraient faire la même chose avec le RTTI et perdraient ainsi l'importante valeur du polymorphisme dans le développement et la maintenance du code. L'intention de Java est de vous faire utiliser des appels de méthodes polymorphiques dans votre code, et de vous faire utiliser le RTTI uniquement lorsque c'est nécessaire.

Néanmoins, utiliser des appels de méthodes polymorphiques nécessite que vous ayez le contrôle de la définition des classes de base car il est possible que lors du développement de votre programme vous découvriez que la classe de base ne contient pas une méthode dans vous avez besoin. Si la classe de base provient d'une bibliothèque ou si elle est contrôlée par quelqu'un d'autre, une solution à ce problème est le RTTI : vous pouvez créer un nouveau type héritant de cette classe auquel vous ajoutez la méthode manquante. Ailleurs dans le code, vous détectez ce type particulier et appelez cette méthode spécifique. Ceci ne détruit ni le polymorphisme ni l'extensibilité du programme car ajouter un nouveau type ne vous oblige pas à chasser les expressions **switch** dans votre programme. Cependant, lorsque vous ajoutez du code qui requiert cette nouvelle fonctionnalité dans votre programme principal, vous devez utiliser le RTTI pour détecter ce type particulier.

Mettre la nouvelle caractéristique dans la classe de base peut signifier que, pour le bénéfice d'une classe particulière, toutes les autres classes dérivées de cette classe de base devront contenir des bouts de code inutiles de la méthode. Cela rend l'interface moins claire et ennuie celui qui doit redéfinir des méthodes abstraites dérivant de la classe de base. Supposez que vous désiriez nettoyer les becs? [*spit valves*] de tous les instruments à vent de votre orchestre. Une solution est de mettre une méthode **nettoyerBec()** dans la classe de base **Instrument**, mais c'est ennuyeux car cela implique que les instruments **Electroniques** et à **Percussion** ont aussi un bec. Le RTTI fournit une solution plus élégante dans ce cas car vous pouvez placer la méthode dans une classe spécifique (**Vent** dans notre cas), où elle est appropriée. Néanmoins, une solution encore meilleure est de mettre une méthode **prepareInstrument()** dans la classe de base, mais il se peut que vous ne la trouviez pas la première fois que vous ayez à résoudre le problème, et croyiez à tort que l'utilisation du RTTI est nécessaire.

Enfin, le RTTI permettra parfois de résoudre des problèmes d'efficacité. Si votre code utilise le polymorphisme, mais qu'il s'avère que l'un de vos objets réagisse à ce code très général d'une manière particulièrement inefficace, vous pouvez reconnaître ce type en utilisant le RTTI et écrire un morceau de code spécifique pour améliorer son efficacité. Attention toutefois à programmer pour l'efficacité trop tôt. C'est un piège séduisant. Il est préférable d'avoir un programme qui marche d'abord, et décider ensuite s'il est assez rapide, et seulement à

ce moment là vous attaquer aux problèmes de performances &mdash; avec un *profiler*.

## Exercices

Les solutions de certains exercices se trouvent dans le document électronique *Guide des solutions annoté de Penser en Java*, disponible à prix réduit depuis [www.BruceEckel.com](http://www.BruceEckel.com).

1. Ajouter **Rhomboïde** à **Formes.java**. Créer un **Rhomboïde**, le transtyper en une **Forme**, ensuite le re-transtyper en un **Rhomboïde**. Essayer de le transtyper en un **Cercle** et voir ce qui se passe.
2. Modifier l'exercice 1 afin d'utiliser **instanceof** pour vérifier le type avant d'effectuer le transtypage descendant.
3. Modifier **Formes.java** afin que toutes les formes d'un type particulier puissent être mises en surbrillance (utiliser un drapeau). La méthode **toString()** pour chaque classe dérivée de **Forme** devra indiquer si cette **Forme** est mise en surbrillance ou pas.
4. Modifier **Confiseur.java** afin que la création de chaque type d'objet soit contrôlée par la ligne de commande. Par exemple, si la ligne de commande est `&ldquo;java Confiseur Bonbon&rdquo;`, seul l'objet **Bonbon** sera créé. Noter comment vous pouvez contrôler quels objets **Class** sont chargés via la ligne de commande.
5. Ajouter un nouveau type de **Pet** à **PetCount3.java**. Vérifier qu'il est correctement créé et compté dans la méthode **main()**.
6. Écrire une méthode qui prend un objet en paramètre et affiche récursivement tous les classes de sa hiérarchie.
7. Modifier l'exercice 6 afin d'utiliser **Class.getDeclaredFields()** pour afficher aussi les informations sur les champs de chaque classe.
8. Dans **ToyTest.java**, commenter le constructeur par défaut de **Toy** et expliquer ce qui arrive.
9. Ajouter une nouvelle **interface** dans **ToyTest.java** et vérifier qu'elle est correctement détectée et affichée.
10. Créer un nouveau type de conteneur qui utilise une **private ArrayList** pour stocker les objets. Déterminer le type du premier objet déposé, ne permettre ensuite à l'utilisateur que d'insérer des objets de ce type.
11. Écrire un programme qui détermine si un tableau de **char** est un type primitif ou réellement un objet.
12. Implanter **nettoyerBec()** comme décrit dans le résumé.
13. Modifier l'exercice 6 afin d'utiliser la réflexion à la place du RTTI.
14. Modifier l'exercice 7 afin d'utiliser la réflexion à la place du RTTI.
15. Dans **ToyTest.java**, utiliser la réflexion pour créer un objet **Toy** en n'utilisant pas le constructeur par défaut.
16. Étudier l'interface **java.lang.Class** dans la documentation HTML de Java à [java.sun.com](http://java.sun.com). Écrire un programme qui prend en paramètre le nom d'une classe via la ligne de commande, et utilise les méthodes

de **Class** pour extraire toutes les informations disponibles pour cette classe. Tester le programme sur une classe de la bibliothèque standard et sur une des vôtres.

---

[\[60\]](#) Spécialement dans le passé. Néanmoins, Sun a grandement amélioré la documentation HTML de Java et il est maintenant plus aisé de voir les méthodes des classes de base.