12: Identificación de tipos en tiempo de ejecución

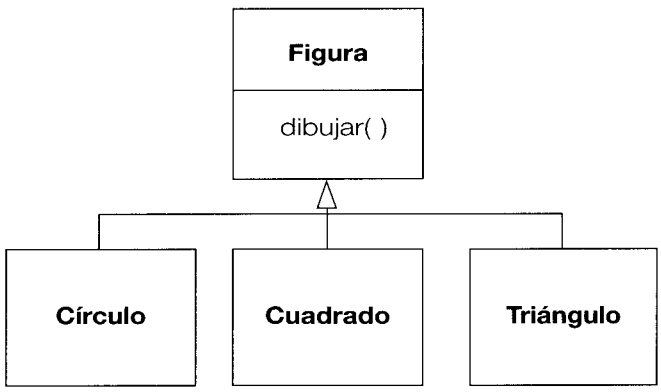
La idea de identificación de tipos en tiempo de ejecución[1] parece bastante simple a primera vista: permite encontrar el tipo exacto de un objeto cuando se tiene sólo una referencia al tipo base.

Sin embargo, la *necesidad* de RTTI no cubre una gran plétora de aspectos de diseño OO interesantes (y en ocasiones que dejan perplejos), y destapa preguntas fundamentales sobre cómo se deberían estructurar los programas.

Este capítulo repasa las formas en que Java permite descubrir información, tanto sobre clases, como relativas a objetos en tiempo de ejecución. Esto se hace de dos formas: RTTI "tradicional", que asume que todos los tipos están disponibles tanto en tiempo de ejecución como de compilación, y el mecanismo de "reflexión", que permite descubrir información de clases únicamente en tiempo de ejecución. Se cubrirá primero el RTTI "tradicional", siguiendo una discusión sobre la reflectividad a continuación.

La necesidad de RTTI

Considérese el ya familiar ejemplo de una jerarquía de clases que hace uso del polimorfismo. El tipo genérico es el de la clase base **Figura**, y los tipos específicos derivados son **Círculo**, **Cuadrado**, y **Triángulo**:



[1] N. del traductor: En inglés *Run-time Type Identification* o RTTI.

Éste es un diagrama jerárquico de clases típico, con la clase base en la parte superior y las clases derivadas creciendo hacia abajo. La meta normal en la POO es que la mayor cantidad posible de código manipule referencias de la clase base (en este caso, **Figura**) de forma que si se decide extender el programa añadiendo una clase nueva (**Romboide**, derivada de **Figura**, por ejemplo), la gran mayoría del código no se vea afectada. En este ejemplo, el método asignado estáticamente en la interfaz **Figura** es **dibujar( ),** por tanto, se pretende que el programador cliente invoque a **dibujar( )** a través de una referencia **Figura** genérica. El método **dibujar( )** está superpuesto en todas las clases derivadas, y dado que por ello es un método de correspondencia dinámica, se obtendrá el comportamiento adecuado incluso aunque se invoque a través de una referencia **Figura** genérica. Esto es el polimorfismo.

Por consiguiente, se suele crear un objeto específico (**Círculo**, **Cuadrado** o **Triángulo**), se aplica un molde hacia arriba a una **Figura** (olvidando el tipo específico del objeto), y se usa como una referencia **Figura** anónima en el resto del programa.

Para dar un breve repaso al polimorfismo y aplicar un molde hacia arriba, se puede echar un vistazo al siguiente ejemplo:

```
//: c12:Figuras.java
import java.util.*;

class Figura {
  void dibujar() {
    System.out.println(this + ".dibujar()");
  }
}

class Circulo extends Figura {
  public String toString() { return "Circulo"; }
}

class Cuadrado extends Figura {
  public String toString() { return "Cuadrado"; }
}

class Triangulo extends Figura{
  public String toString() { return "Triangulo"; }
}

public class Figura {
  public static void main(String[] args) {
    ArrayList s = new ArrayList();
    s.add(new Circulo());
    s.add(new Cuadrado());
    s.add(new Triangulo());
```

```
        Iterator e = s.iterator();
        while(e.hasNext())
          ((Figura)e.next()).dibujar();
    }
} ///:~
```

La clase base contiene un método **dibujar( )** que usa indirectamente **toString( )** para imprimir un identificador de la clase pasando **this** a **System.out.println( )**. Si esa función ve un objeto, llama automáticamente al método **toString( )** para producir una representacion **String.**

Cada una de las clases derivadas superpone el método **toString( )** (de **Object**) de forma que **dibujar( )** acaba imprimiendo algo distinto en cada caso. En el método **main( )** se crean tipos específicos de **Figura** que después se añaden a una **ArrayList**. Éste es el momento en el que se aplica un molde hacia arriba puesto que **ArrayList** sólo guarda **Object**s. Puesto que en Java todo es un **Object** (excepto los tipos primitivos), un **ArrayList** puede guardar también objetos **Figura**. Pero al aplicar un molde hacia arriba a **Object**, también se pierde información específica, incluyendo el hecho de que los objetos son **Figuras**. En lo que a **ArrayList** se refiere, son simplemente **Object**s.

En el momento de recuperar un elemento de **ArrayList** con **next( )**, todo se vuelve más complicado. Dado que **ArrayList** simplemente guarda **Object**s, naturalmente **next( )** produce una referencia **Object**. Pero sabemos que verdaderamente es una referencia a **Figura**, y se desea poder enviar a ese objeto **Figura** mensajes. Por tanto, es necesaria una conversión a **Figura** utilizando el molde tradicional **"(Figura)"**. Ésta es la forma más básica de RTTI, puesto que en tiempo de ejecución se comprueba que todas las conversiones sean correctas. Esto es exactamente lo que significa RTTI: identificar en tiempo de ejecución el tipo de los objetos.

En este caso, la conversión RTTI es sólo parcial: se convierte el **Object** a **Figura**, y no hasta **Círculo**, **Cuadrado** o **Triángulo**. Esto se debe a que lo único que *se sabe* en este momento es que **ArrayList** está lleno de **Figuras**.

En tiempo de compilación, se refuerza esto sólo por reglas autoimpuestas; en tiempo de ejecución, prácticamente se asegura.

Ahora toma su papel el polimorfismo y se determina el método exacto invocado para **Figura** para saber si es una referencia a **Círculo**, **Cuadrado** o **Triángulo**. Y así es como debería ser en general; se desea que la mayor parte del código sepa lo menos posible sobre los tipos *específicos* de los objetos, y simplemente tratar con la representación general de una familia de objetos (en este caso, **Figura**). El resultado es un código más fácil de escribir, leer y mantener, y los diseños serán más fáciles de implementar, entender y cambiar. Por tanto, el polimorfismo es la meta general en la programación orientada a objetos.

Pero, ¿qué ocurre si se tiene un problema de programación especial que es más fácil de solucionar conociendo el tipo exacto de una referencia genérica? Por ejemplo, supóngase que se desea permitir a los objetos resaltar todas las formas de determinado tipo pintándolas de violeta. De esta forma se podría, por ejemplo, localizar todos los triángulos de la pantalla. Esto es lo que logra RTTI: se puede pedir a una referencia **Figura** el tipo exacto al que se refiere.

El objeto **Class**

Para entender cómo funciona la RTTI en Java, hay que saber primero cómo se representa la información de tipos en tiempo de ejecución. Esto se logra mediante una clase especial de objeto denominada el *objeto Class*, que contiene información sobre la clase. (En ocasiones se le denomina *metaclase.*) De hecho, se usa el objeto **Class** para crear todos los objetos "normales" de la clase.

Hay un objeto **Class** por cada clase que forme parte del programa. Es decir, cada vez que se escribe y compila una nueva clase, se crea también un objeto **Class** (y se almacena, en un archivo de nombre igual y extensión **.class**). En tiempo de ejecución, cuando se desea construir un objeto de esa clase, la Máquina Virtual Java que está ejecutando el programa comprueba en primer lugar si se ha cargado el objeto **Class** para ese tipo. Sino, la JVM lo carga localizando el archivo **.class** de este nombre. Por consiguiente, los programas Java no se cargan completamente antes de empezar, a diferencia de la mayoría de lenguajes tradicionales.

Una vez que el objeto **Class** de ese tipo está en memoria, se usa para crear todos los objetos de ese tipo.

Si esto parece un poco sombrío o uno no puede creerlo, he aquí un programa demostración que lo prueba:

```
//: c12:Confiteria.java
// Examen de cómo funciona el cargador de clases.

class Caramelo {
  static {
    System.out.println("Cargando Caramelo");
  }
}

class Chicle {
  static {
    System.out.println("Cargando Chicle");
  }
}

class Galleta {
  static {
    System.out.println("Cargando Galleta");
  }
}

public class Confiteria {
  public static void main(String[] args) {
    System.out.println("dentro del método main");
    new Caramelo();
```

```
    System.out.println("Después de crear Caramelo");
    try {
      Class.forName("Chicle");
    } catch(ClassNotFoundException e) {
      e.printStackTrace(System.err);
    }
    System.out.println(
      "Después de Class.forName(\"Chicle\")");
    new Galleta();
    System.out.println("Después de crear la Galleta");
  }
} ///:~
```

Cada una de las clases **Caramelo**, **Chicle** y **Galleta** tienen una cláusula **static** que se ejecuta al cargar la clase por primera vez. Se imprimirá información para indicar cuándo se carga esa clase. En el método **main( )**, las creaciones de objetos están dispersas entre sentencias de impresión para ayudar a detectar el momento de carga.

Una línea particularmente interesante es:

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

Este método es un miembro **static** de **Class** (al que pertenecen todos los objetos **Class**). Un objeto **Class** es como cualquier otro objeto, de forma que se pueden recuperar y manipular referencias al mismo. (Eso es lo que hace el cargador.) Una de las formas de lograr una referencia al objeto **Class** es **forName( )**, que toma un **String** que contiene el nombre textual (¡hay que tener cuidado con el deletreo y las mayúsculas!) de la clase particular para la que se desea una referencia. Devuelve una referencia **Class.**

La salida de este programa para una JVM es:

```
dentro del método main
Cargando Caramelo
Despues de crear Caramelo
Cargando Chicle
Despues de Class.forName("Chicle")
Cargando Galleta
Despues de crear Galleta
```

Se puede ver que cada objeto **Class** sólo se crea cuando es necesario, y se lleva a cabo la inicialización **static** en el momento de cargar la clase.

Literales de clase

Java proporciona una segunda forma de producir la referencia al objeto **Class**, utilizando un *literal de clase*. En el programa de arriba, esto sería de la forma:

```
Chicle.class;
```

que no sólo es más simple, sino que es también seguro puesto que se comprueba en tiempo de compilación. Dado que elimina la llamada al método, también es más eficiente.

Los literales de clase funcionan con clases regulares además de con interfaces, arrays y tipos primitivos. Además, hay un campo estándar denominado **TYPE** que existe para cada una de las clases envoltorio primitivas. El campo **TYPE** produce una referencia al objeto **Class** para el tipo primitivo asociado, como:

... es equivalente a ...	
boolean.class	**Boolean.TYPE**
char.class	**Character.TYPE**
byte.class	**Byte.TYPE**
short.class	**Short.TYPE**
int.class	**Integer.TYPE**
long.class	**Long.TYPE**
float.class	**Float.TYPE**
double.class	**Double.TYPE**
void.class	**Void.TYPE**

Es preferible usar las versiones "**.class**" si se puede, puesto que son más consistentes con clases regulares.

Comprobar antes de una conversión

Hasta la fecha, se han visto las formas RTTI incluyendo:

1. La conversión clásica; por ejemplo, "**(Figura)**," que usa RTTI para asegurarse de que la conversión sea correcta y lanza una **ClassCastException** si se ha hecho una conversión errónea.
2. El objeto **Class** que representa el tipo de objeto. Se puede preguntar al objeto **Class** por información útil de tiempo de ejecución.

En C++, la conversión clásica **"(Figura)"** *no* lleva a cabo RTTI. Simplemente dice al compilador que trate al objeto como el nuevo tipo. En Java, que lleva a cabo la comprobación de tipos, a esta comprobación se le suele llamar una "conversión segura hacia abajo".

La razón por la que se usa la expresión "*aplicar molde hacia abajo*" es la disposición histórica del diagrama de jerarquía de clases. Si convertir un **Círculo** a una **Figura** es aplicar un molde hacia arriba, convertir una **Figura** en un **Círculo** es aplicar un molde hacia abajo. Sin embargo, se sabe que un **Círculo** es también una **Figura,** y el compilador permite libremente una asignación hacia

arriba, pero se *desconoce* que una **Figura** sea necesariamente un **Círculo**, de forma que el compilador no te permite llevar a cabo una asignación hacia abajo sin usar una conversión explícita.

Hay una tercera forma de RTTI en Java. Es la palabra clave **instanceof** la que dice si un objeto es una instancia de un tipo particular. Devuelve un **boolean,** de forma que si se usa en forma de cuestión, como en:

```
if(x instanceof Perro)
  ((Perro) x).ladrar();
```

la sentencia **if** de arriba comprueba si el objeto **x** pertenece a la clase **Perro** *antes* de convertir **x** en un **Perro**. Es importante utilizar **instanceof** antes de aplicar un molde hacia abajo, cuando no se tiene ninguna otra información que indique el tipo de objeto; de otra forma se acabará con una **ClassCastException**.

De forma ordinaria, se podría estar buscando un tipo (triángulos para pintarlos de violeta, por ejemplo), pero se puede llevar fácilmente la cuenta de *todos* los objetos utilizando **instanceof**. Supóngase que se tiene una familia de clases **AnimalDomestico**:

```
//: c12:AnimalesDomesticos.java
class AnimalDomestico {}
class Perro extends AnimalDomestico {}
class Doguillo extends Perro {}
class Gato extends AnimalDomestico {}
class Roedor extends AnimalDomestico {}
class Gerbo extends Roedor {}
class Hamster extends Roedor {}

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

La clase **Contador** se usa para llevar un seguimiento de cualquier tipo de **AnimalDomestico** particular. Se podría pensar que es como un **Integer** que puede ser modificado.

Utilizando **instanceof** pueden contarse todos los animales domésticos:

```
//: c12:RecuentoAnimalDomestico.java
// Usando instanceof.
import java.util.*;

public class RecuentoAnimalDomestico {
  static String[] nombresTipo = {
    "AnimalDomestico", "Perro", "Doguillo", "Gato",
    "Roedor", "Gerbo", "Hamster",
  };
  // Lanzar las excepciones a la consola:
  public static void main(String[] args)
  throws Exception {
```

```
ArrayList animalesDomesticos = new ArrayList();
try {
  Class[] tiposAnimalDomestico = {
    Class.forName("Perro"),
    Class.forName("Doguillo"),
    Class.forName("Gato"),
    Class.forName("Roedor"),
    Class.forName("Gerbil"),
    Class.forName("Hamster"),
  };
  for(int i = 0; i < 15; i++)
    animalesDomesticos.add(
      tiposAnimalDomestico[
        (int)(Math.random()*tiposAnimalDomestico.length)]
        .newInstance());
} catch(InstantiationException e) {
  System.err.println("No se puede instanciar");
  throw e;
} catch(IllegalAccessException e) {
  System.err.println("No se puede acceder");
  throw e;
} catch(ClassNotFoundException e) {
  System.err.println("No se puede encontrar la clase");
  throw e;
}
HashMap h = new HashMap();
for(int i = 0; i < nombresTipo.length; i++)
  h.put(nombresTipo[i], new Contador());
for(int i = 0; i < animalesDomesticos.size(); i++) {
  Object o = animalesDomesticos.get(i);
  if(o instanceof AnimalDomestico)
    ((Contador)h.get("animalesDomesticos")).i++;
  if(o instanceof Perro)
    ((Contador)h.get("Perro")).i++;
  if(o instanceof Doguillo)
    ((Contador)h.get("Doguillo")).i++;
  if(o instanceof Gato)
    ((Contador)h.get("Gato")).i++;
  if(o instanceof Roedor)
    ((Contador)h.get("Roedor")).i++;
  if(o instanceof Gerbo)
    ((Contador)h.get("Gerbo")).i++;
  if(o instanceof Hamster)
    ((Contador)h.get("Hamster")).i++;
}
```

```
    for(int i = 0; i < animalesDomesticos.size(); i++)
      System.out.println(animalesDomesticos.get(i).getClass());
    for(int i = 0; i < nombresTipo.length; i++)
      System.out.println(
        nombresTipo[i] + " cantidad: " +
        ((Contador)h.get(nombresTipo[i])).i);
  }
} ///:~
```

Hay una restricción bastante severa en **instanceof**: se puede comparar sólo a tipos con nombre, y no a un objeto **Class**. En el ejemplo de arriba se podría pensar que es tedioso escribir todas esas expresiones **instanceof**, lo que es cierto. Pero no hay forma de automatizar inteligentemente **instanceof** creando una **ArrayList** de objetos **Class** y compararlo con éstos en su lugar (permanezca atento —hay una alternativa). Ésta no es una restricción tan grande como se podría pensar, porque generalmente se entenderá que el diseño será más defectuoso si se acaban escribiendo una multitud de expresiones **instanceof**.

Por supuesto, este ejemplo es artificial —probablemente se pondría un miembro de datos **static** en cada tipo de incremento en el constructor para mantener un seguimiento del recuento. Se haría algo así *si* se tuviera control sobre el código fuente de la clase y se cambiaría. Dado que éste no es siempre el caso, RTTI puede venir bien.

Utilizar literales de clase

Es interesante ver cómo se puede reescribir el ejemplo **RecuentoAnimalDomestico.java** utilizando literales de clase. El resultado es más limpio en muchos sentidos:

```
//: c12:RecuentoAnimalDomestico2.java
// Utilizando literales de clase.
import java.util.*;

public class RecuentoAnimalDomestico2 {
  public static void main(String[] args)
  throws Exception {
    ArrayList animalesDomesticos = new ArrayList();
    Class[] tiposAnimalDomestico = {
      // Literales de clase:
      AnimalDomestico.class,
      Perro.class,
      Doguillo.class,
      Gato.class,
      Roedor.class,
      Gerbo.class,
      Hamster.class,
    };
    try {
```

```
    for(int i = 0; i < 15; i++) {
      // Desplazamiento de uno para eliminar AnimalDomestico.class:
      int rnd = 1 + (int)(
        Math.random() * (tiposAnimalDomestico.length - 1));
      pets.add(
        tiposAnimalDomestico[rnd].newInstance());
    }
  } catch(InstantiationException e) {
    System.err.println("No se puede instanciar");
    throw e;
  } catch(IllegalAccessException e) {
    System.err.println("No se puede acceder");
    throw e;
  }
  HashMap h = new HashMap();
  for(int i = 0; i < tiposAnimalDomestico.length; i++)
    h.put(tiposAnimalDomestico[i].toString(),
      new Contador());
  for(int i = 0; i < animalesDomesticos.size(); i++) {
    Object o = animalesDomesticos.get(i);
    if(o instanceof animalDomestico)
      ((Contador)h.get("clase AnimalDomestico")).i++;
    if(o instanceof Perro)
      ((Contador)h.get("class Perro")).i++;
    if(o instanceof Doguillo)
      ((Contador)h.get("class Doguillo")).i++;
    if(o instanceof Gato)
      ((Contador)h.get("class Gato")).i++;
    if(o instanceof Roedor)
      ((Contador)h.get("class Roedor")).i++;
    if(o instanceof Gerbo)
      ((Contador)h.get("class Gerbo")).i++;
    if(o instanceof Hamster)
      ((Contador)h.get("class Hamster")).i++;
  }
  for(int i = 0; i < animalesDomesticos.size(); i++)
    System.out.println(animalesDomesticos.get(i).getClass());
  Iterator claves = h.keySet().iterator();
  while(claves.hasNext()) {
    String nm = (String)claves.next();
    Contador cnt = (Contador)h.get(nm);
    System.out.println(
      nm.substring(nm.lastIndexOf('.') + 1) +
      " cantidad: " + cnt.i);
  }
```

```
  }
} ///:~
```

Aquí, se ha retirado el array **nombresTipo** para conseguir los strings de nombres de tipos de los objetos **Class**. Nótese que el sistema puede distinguir entre clases e interfaces.

También se puede ver que la creación de **tiposAnimalDomestico** no tiene por qué estar rodeada de un bloque **try** porque se evalúa en tiempo de compilación y por consiguiente no lanzará ninguna excepción, a diferencia de **Class.forName( )**.

Cuando se crean dinámicamente los objetos **AnimalDomestico**, se puede ver que se restringe el número aleatorio de forma que esté entre uno y **tiposAnimalDomestico.length** y sin incluir el cero. Eso es porque el cero hace referencia a **AnimalDomestico.class**, y presumiblemente un objeto **AnimalDomestico** genérico no sea interesante. Sin embargo, dado que **AnimalDomestico.class** es parte de **tiposAnimalesDomestico** el resultado es que se cuentan todos los animales domésticos.

Un **instanceof** dinámico

El método de **Class isInstance** proporciona una forma de invocar dinámicamente al operador **instanceof**. Por consiguiente, todas esas tediosas sentencias **instanceof** pueden eliminarse en el ejemplo **RecuentoAnimalDomestico**:

```
//: c12:RecuentoAnimalDomestico3.java
// Usando isInstance().
import java.util.*;

public class RecuentoAnimalDomestico3 {
  public static void main(String[] args)
  throws Exception {
    ArrayList animalesDomesticos = new ArrayList();
    Class[] tiposAnimalDomestico = {
      AnimalDomestico.class,
      Perro.class,
      Doguillo.class,
      Gato.class,
      Roedor.class,
      Gerbo.class,
      Hamster.class,
    };
    try {
      for(int i = 0; i < 15; i++) {
        // Desplazar en uno para eliminar AnimalDomestico.class:
        int rnd = 1 + (int)(
          Math.random() * (tiposAnimalDomestico.length - 1));
        animalesDomesticos.add(
          tiposAnimalDomestico[rnd].newInstance());
      }
```

```
      } catch(InstantiationException e) {
        System.err.println("No se puede instanciar");
        throw e;
      } catch(IllegalAccessException e) {
        System.err.println("No se puede acceder");
        throw e;
      }
      HashMap h = new HashMap();
      for(int i = 0; i < tiposAnimalDomestico.length; i++)
        h.put(tiposAnimalDomestico[i].toString(),
          new Contador());
      for(int i = 0; i < animalesDomesticos.size(); i++) {
        Object o = animalesDomesticos.get(i);
        // Usando isInstance para eliminar las expresiones
        // instanceof individuales:
        for (int j = 0; j < tiposAnimalDomestico.length; ++j)
          if (tiposAnimalDomestico[j].isInstance(o)) {
            String clave = tiposAnimalDomestico[j].toString();
            ((Contador)h.get(clave)).i++;
          }
      }
      for(int i = 0; i < animalesDomesticos.size(); i++)
        System.out.println(animalesDomesticos.get(i).getClass());
      Iterator clave = h.keySet().iterator();
      while(claves.hasNext()) {
        String nm = (String)claves.next();
        Contador cnt = (Contador)h.get(nm);
        System.out.println(
          nm.substring(nm.lastIndexOf('.') + 1) +
          " cantidad: " + cnt.i);
      }
    }
} ///:~
```

Veamos que el método **isInstance( )** ha eliminado la necesidad de expresiones **instanceof**. Además, esto significa que se pueden añadir nuevos tipos de animal doméstico simplemente cambiando el array **tiposAnimalDomestico**; el resto del programa no necesita modificación alguna (y sí cuando se usaban las expresiones **instanceof**).

instanceof frente a equivalencia de Class

Cuando se pregunta por información de tipos, hay una diferencia importante entre cualquier forma de **instanceof** (es decir, **instanceof** o **isInstance( )**, que produce resultados equivalentes) y la comparación directa de los objetos **Class**. He aquí un ejemplo que demuestra la diferencia:

```
//: c12:FamiliaVSTipoExacto.java
// La diferencia entre instanceof y class
```

```
class Base {}
class Derivada extends Base {}

public class FamiliaVsTipoExacto {
  static void comprobar(Object x) {
    System.out.println("Probando x de tipo " +
      x.getClass());
    System.out.println("x instanceof Base " +
      (x instanceof Base));
    System.out.println("x instanceof Derivada " +
      (x instanceof Derivada));
    System.out.println("Base.isInstance(x) " +
      Base.class.isInstance(x));
    System.out.println("Derivada.isInstance(x) " +
      Derivada.class.isInstance(x));
    System.out.println(
      "x.getClass() == Base.class " +
      (x.getClass() == Base.class));
    System.out.println(
      "x.getClass() == Derivada.class " +
      (x.getClass() == Derivada.class));
    System.out.println(
      "x.getClass().equals(Base.class)) " +
      (x.getClass().equals(Base.class)));
    System.out.println(
      "x.getClass().equals(Derivada.class)) " +
      (x.getClass().equals(Derivada.class)));
  }
  public static void main(String[] args) {
    test(new Base());
    test(new Derivada());
  }
} ///:~
```

El método **comprobar( )** lleva a cabo la comprobación de tipos con su argumento usando ambas formas de **instanceof**. Después toma la referencia **Class** y usa == y **equals( )** para probar la igualdad de los objetos **Class**. He aquí la salida:

```
Probando x de tipo class Base
x instanceof Base true
x instanceof Derivada false
Base.isInstance(x) true
Derivada.isInstance(x) false
x.getClass() == Base.class true
x.getClass() == Derivada.class false
```

```
x.getClass().equals(Base.class)) true
x.getClass.equals(Derivada.class)) false
Probando x de tipo class Derivada
x instanceof Base true
x instanceof Derivada true
Base.isInstance (x) true
Derivada.isInstance(x) true
x.getClass() == Base.class false
x.getClass() == Derivada.class true
x.getClass().equals(Base.class)) false
x.getClass().equals(Derivada.class)) true
```

En definitiva, **instanceof** e **isInstance( )** producen exactamente los mismos resultados, al igual que ocurre con **equals( )** y **==**. Pero las pruebas en sí muestran varias conclusiones. En lo que al concepto de tipo se refiere, **instanceof** dice: "¿Eres de esta clase o de una clase derivada de ésta?" Por otro lado, si se comparan los objetos **Class** utilizando ==, no importa la herencia —o es del tipo exacto o no lo es.

Sintaxis RTTI

Java lleva a cabo su RTTI utilizando el objeto **Class**, incluso aunque se esté haciendo alguna conversión. La clase **Class** tiene también otras formas de cara al uso de RTTI.

En primer lugar, hay que conseguir una referencia al objeto **Class** apropiado. Una forma de lograrlo, como se vio en el ejemplo anterior, es usar una cadena de caracteres y el método **Class.forName( )**. Esto es conveniente pues no es necesario un objeto de ese tipo para lograr la referencia **Class.** Sin embargo, si ya se tiene un objeto del tipo en el que se está interesado, se puede lograr la referencia **Class** llamando a un método que sea parte de la clase raíz **Object**: **getClass( )**. Éste devuelve la referencia **Class** que representa al tipo de objeto actual. **Class** tiene muchos métodos interesantes, que se demuestran en el ejemplo siguiente:

```
//: c12:PruebaJuguete.java
// Probando la clase Class.

interface TienePilas {}
interface ResisteAgua {}
interface DisparaCosas {}
class Juguete {
  // Marcar como comentario el siguiente constructor
  // por defecto para ver
  // NoSuchMethodError de (*1*)
  Juguete() {}
  Juguete(int i) {}
}

class JugueteFantasia extends Juguete
```

```
    implements TienePilas,
      ResisteAgua, DisparaCosas {
  JugueteFantasia() { super(1); }
}

public class PruebaJuguete {
  public static void main(String[] args)
  throws Exception {
    Class c = null;
    try {
      c = Class.forName("JugueteFantasia");
    } catch(ClassNotFoundException e) {
      System.err.println("No se puede encontrar JugueteFantasia");
      throw e;
    }
    imprimirInfo(c);
    Class[] semblantes = c.getInterfaces();
    for(int i = 0; i < semblantes.length; i++)
      imprimirInfo(semblantes[i]);
    Class cy = c.getSuperclass();
    Object o = null;
    try {
      // Requiere del constructor por defecto:
      o = cy.newInstance(); // (*1*)
    } catch(InstantiationException e) {
      System.err.println("No se puede instanciar");
      throw e;
    } catch(IllegalAccessException e) {
      System.err.println("No se puede acceder");
      throw e;
    }
    imprimirInfo(o.getClass());
  }
  static void imprimirInfo(Class cc) {
    System.out.println(
      "Nombre de clase: " + cc.getName() +
      " ¿es interfaz? [" +
      cc.isInterface() + "]");
  }
} ///:~
```

Veamos que la **class JugueteFantasia** es bastante complicada, puesto que hereda de **Juguete** e **implementa** las **interfaces TienePilas**, **ResisteAgua** y **DisparaCosas**. En el método **main( )**, se crea una referencia **Class** y se inicializa a la **Class JugueteFantasia** usando **forName( )** dentro de un bloque **try** apropiado.

El método **Class.getInterfaces( )** devuelve un array de objetos **Class** que representa las interfaces de interés contenidas en el objeto **Class**.

Si se tiene un objeto **Class** también se puede pedir su clase directa base utilizando **getSuperclass ( )**. Esto, por supuesto, devuelve una referencia **Class** por la que se puede preguntar más adelante. Esto significa que, en tiempo de ejecución, se puede descubrir toda la jerarquía de clases de un objeto.

El método **newInstance( )** de **Class** puede, en primer lugar, parecer justo otra manera de **clone( )** (clonar) un objeto. Sin embargo, se puede crear un objeto nuevo con **newInstance( )** *sin* un objeto existente, como se ha visto, porque no hay objeto **Juguete** —sólo **cy**, que es una referencia al objeto **Class** de **y**. Ésta es una forma de implementar un "constructor virtual", que te permite decir: "No sé exactamente de qué tipo eres, pero de todas formas créate a ti mismo de la forma adecuada". En el ejemplo de arriba, **cy** es simplemente una referencia **Class**, sin que se conozca más información sobre su tipo en tiempo de compilación. Y cuando se crea una instancia nueva, se obtiene una **referencia Object**. Pero esa referencia apunta a un objeto **Juguete**. Por supuesto, antes de poder enviar cualquier mensaje que no sea aceptado por **Object**, hay que investigar esta referencia un poco y hacer algún tipo de conversión. Además, la clase que se está creando con **newInstance( )** debe tener un constructor por defecto. En la sección siguiente se verá cómo crear objetos de clases dinámicamente utilizando cualquier constructor, con el API Java *reflectivo*.

El método final del listado es **imprimirInfo( )**, que toma una referencia **Class** y consigue su nombre con **getName( )**, y averigua si se trata o no de una interfaz con **isInterface( )**.

La salida de este programa es:

```
Class name: JugueteFantasia ¿es interfaz? [false]
Class name: TienePilas ¿es interfaz? [true]
Class name: ResisteAgua ¿es interfaz? [true]
Class name: DisparaCosas ¿es interfaz? [true]
Class name: Juguete ¿es interfaz? [false]
```

Por consiguiente, con el objeto **Class** se puede averiguar casi todo lo que se desee saber sobre un objeto.

Reflectividad: información de clases en tiempo de ejecución

Si no se sabe el tipo exacto de un objeto, RTTI te lo dice. Sin embargo, hay una limitación: debe conocerse el tipo en tiempo de compilación para poder detectarlo usando RTTI y hacer algo útil con la información. Dicho de otra forma, el compilador debe conocer información sobre todas las clases con las que trabaja de cara a la RTTI.

Esto no parece una gran limitación a primera vista, pero supóngase que se obtiene una referencia a un objeto que no está en el espacio del programa. De hecho, la clase del objeto ni siquiera está disponible para el programa en tiempo de compilación. Por ejemplo, supóngase que se obtiene un conjunto de bytes de un archivo de disco o de una conexión de red, y se sabe que se trata de una cla-

se. Puesto que el compilador no puede saber nada acerca de la clase mientras está compilando el código, ¿cómo podría llegar a usarla?

En los entornos de programación tradicionales éste parece un escenario poco probable. Pero a medida que nos introducimos en un mundo de programación mayor, hay casos importantes en los que esto ocurre. El primero es la programación basada en componentes, en la que se construyen proyectos utilizando *Rapid Application Development*[2] (RAD) en una herramienta para la construcción de aplicaciones. Se trata de enfoques visuales para crear programas (que se muestran en pantallas como "formularios") moviendo iconos que representan componentes dentro de los formularios. Estos componentes se configuran después estableciendo algunos de los valores en tiempo de programación. Esta configuración en tiempo de diseño exige que todos los componentes sean instanciables, que expongan partes de sí mismos, y que permitan la lectura y asignación de valores. Además, los componentes que manejen eventos de la IGU deben exponer información sobre los métodos apropiados de forma que el entorno RAD pueda ayudar al programador a superponer estos métodos de manejo de eventos. La reflectividad proporciona el mecanismo para detectar los métodos disponibles y producir los nombres de método. Java proporciona una estructura para programación basada en componentes a través de los denominados JavaBeans (descritos en el Capítulo 13).

Otra motivación que conduce al descubrimiento de información de clases en tiempo de ejecución es proporcionar la habilidad de crear y ejecutar objetos en plataformas remotas a través de la red. A esto se le llama *Remote Method Invocation*[3] (RMI) y permite a un programa Java tener objetos distribuidos por varias máquinas. Esta distribución puede darse por varias razones: por ejemplo, quizás se está haciendo una tarea de computación intensiva y se desea dividirla entre varias máquinas ociosas para acelerar el rendimiento global. En ocasiones se podría desear ubicar código que maneje tipos de tareas particulares (por ejemplo, "Reglas de negocio" en una arquitectura cliente/servidor multicapa) en una determinada máquina, de forma que esa máquina se convierte en un repositorio común describiendo esas acciones, y que puede modificarse sencillamente para afectar a todo el sistema. (¡Se trata de un desarrollo interesante, pues la máquina sólo existe para facilitar los cambios en el código!) Además de esto, la computación distribuida también permite soportar hardware especializado que puede ser necesario para alguna tarea en particular —inversión de matrices, por ejemplo— pero inapropiado o demasiado caro para programación de propósito general.

La clase **Class** (descrita previamente en este capítulo) soporta el concepto de *reflectividad*, y hay una biblioteca adicional, **java.lang.reflect**, con las clases **Field**, **Method** y **Constructor** (cada una implementa el **interfaz Member**). Los objetos de este tipo los crea la JVM en tiempo de ejecución para representar el miembro correspondiente de clase desconocida. Se pueden usar después los **Constructor**s para crear nuevos objetos, los métodos **get( )** y **set( )** para leer y modificar los campos asociados con los objetos **Field**, y el método **invoke( )** para llamar al método asociado con un objeto **Method**. Además, se puede invocar a los métodos **getFields( )**, **getMethods( )**, **getConstructors( )**, etc. para devolver arrays de objetos que representen los campos, métodos y constructores. (Se puede averiguar aún más buscando la clase **Class** en la documentación en línea.)

---

[2] N. del traductor: Desarrollo Rápido de Aplicaciones.

[3] N. del traductor: Invocación de Métodos Remotos.

Por consiguiente, se puede determinar completamente en tiempo de ejecución la información de clase de los objetos anónimos, sin tener que saber nada en tiempo de compilación.

Es importante darse cuenta de que no hay nada mágico en la reflectividad. Cuando se usa la reflectividad para interactuar con un objeto de un tipo desconocido, la JVM simplemente mira al objeto y ve que pertenece a una clase particular (como el RTTI ordinario) pero en ese momento, antes de hacer nada más, se debe cargar el objeto **Class**. Por consiguiente, debe estar disponible para la JVM el archivo **.class** de ese tipo particular, bien en la máquina local o bien a través de la red. Por tanto, la verdadera diferencia entre RTTI y la reflectividad es que con la RTTI el compilador abre y examina el fichero **.class** en tiempo de compilación. Dicho de otra forma, se puede llamar a todos los métodos del objeto de forma "normal". Con la reflectividad, el archivo **.class** no está disponible en tiempo de compilación; se abre y examina por el entorno en tiempo de ejecución.

Un extractor de métodos de clases

Las herramientas de reflectividad se usarán directamente muy pocas veces; están en el lenguaje para dar soporte a otras facetas de Java, como la serialización de objetos (Capítulo 11), JavaBeans (Capítulo 13) y RMI (Capítulo 15). Sin embargo, hay veces en las que es bastante útil ser capaz de extraer dinámicamente información sobre una clase. Una herramienta extremadamente útil es el extractor de métodos de clases. Como se mencionó anteriormente, mirar el código fuente de una definición de clase o la documentación en línea sólo muestra los métodos definidos o superpuestos *dentro de esa definición de clase*. Pero podría haber otras muchas docenas disponibles provenientes de clases base. Localizarlos es tedioso y encima consume mucho tiempo[4]. Afortunadamente, la reflectividad proporciona una forma de escribir, una herramienta sencilla que mostrará automáticamente todo la interfaz. Funciona así:

```
//: c12:MostrarMetodos.java
// Usando la reflectividad para mostrar todos los métodos
// de una clase, incluso los definidos en
// la clase base.
import java.lang.reflect.*;

public class MostrarMetodos {
  static final String uso =
    "uso: \n" +
    "MostrarMetodos nombre.clase.calificado\n" +
    "Para mostrar todos los metodos de la clase o: \n" +
    "MostrarMetodos nombre.clase.calificado palabra\n" +
    "Para buscar todos los metodos que involucran a 'palabra'";
  public static void main(String[] args) {
    if(args.length < 1) {
```

[4] Especialmente en el pasado. Sin embargo, Sun ha mejorado mucho su documentación HTML de Java de forma que es más fácil acceder a los métodos de la clase base.

```
        System.out.println(uso);
        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("Clase inexistente: " + e);
      }
    }
} ///:~
```

Los métodos de **Class**, **getMethod( )** y **getConstructors( )** devuelven un array de **Method** y **Constructor** respectivamente. Cada una de estas clases tiene más métodos para diseccionar los nombres, parámetros y valores de retorno de los métodos que representan. Pero también se puede usar **toString( )**, como en ese caso, para producir un **String** con la signatura completa del método. El resto del código simplemente sirve para extraer información de línea de comandos, determinar si una signatura en particular coincide con la cadena de caracteres destino (utilizando **indexOf( )**), e imprimir los resultados.

Esto muestra la reflectividad en acción, puesto que no se puede conocer el resultado producido por **Class.forName( )** en tiempo de compilación, y por tanto se extrae toda la información de signatura de métodos en tiempo de ejecución. Si se investiga la documentación *en línea* relativa a la reflectividad, se ve que es suficiente para establecer y construir una llamada a un objeto totalmente conocido en tiempo de compilación (habrá algunos ejemplos de esto al final del presente libro). De nuevo, puede que uno nunca necesite hacer esto —este soporte está por RMI y para que un entorno de programación pueda soportar JavaBeans— pero es interesante.

Un experimento interesante es ejecutar

```
java MostrarMetodos MostrarMetodos
```

Esta invocación produce un listado que incluye un constructor por defecto **public**, incluso aunque se pueda ver en el código que no se definió ningún constructor. El constructor que se ve es el que ha sido automáticamente sintetizado por el compilador. Si después se convierte **MostrarMetodos** en una clase no **public** (es decir, amiga), el constructor sintetizado por defecto deja de mostrar la salida. Al constructor por defecto sintetizado se le da automáticamente el mismo acceso que a la clase.

La salida de **MostrarMetodos** sigue siendo algo tediosa. He aquí, por ejemplo, una porción de la salida producida al invocar **java MostrarMetodos 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)
```

Sería incluso mejor si se eliminaran los calificadores del estilo de **java.lang**. La clase **StreamTokenizer** presentada en el capítulo anterior puede ayudar a crear una herramienta que solucione este problema:

```
//: com:bruceeckel:util:EliminarCalificadores.java
package com.bruceeckel.util;
import java.io.*;

public class EliminarCalificadores {
  private StreamTokenizer st;
  public EliminarCalificadores(String calificado) {
    st = new StreamTokenizer(
      new StringReader(calificado));
    st.ordinaryChar(' '); // Mantener los espacios
  }
  public String obtenerSiguiente() {
    String s = null;
    try {
      int simbolo = st.nextToken();
      if(simbolo != 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: // único carácter en ttype
            s = String.valueOf((char)st.ttype);
        }
      }
    } catch(IOException e) {
      System.err.println("Error recuperando simbolo");
    }
    return s;
  }
  public static String eliminar(String calificado) {
    EliminarCalificadores ec =
      new EliminarCalificadores(calificado);
    String s = "", si;
    while((si = ec.obtenersiguiente()) != null) {
      int ultimoPunto = si.lastIndexOf('.');
      if(ultimoPunto != -1)
        si = si.substring(ultimoPunto + 1);
      s += si;
    }
    return s;
  }
} ///:~
```

Para facilitar la reutilización, esta clase está ubicada en **com.bruceeckel.util**. Como puede verse, hace uso de **StreamTokenizer** y la manipulación de **String**s para hacer su trabajo.

La versión nueva del programa usa la clase de arriba para limpiar la salida:

```
//: c12:LimpiarMostrarMetodos.java
// Mostrar Métodos sin calificadores
// para que los resultados sean más fáciles
// de leer.
import java.lang.reflect.*;
import com.bruceeckel.util.*;

public class LimpiarMostrarMetodos {
  static final String uso =
    "uso: \n" +
    "LimpiarMostrarMetodos nombre.clase.calificado\n" +
    "Para mostrar todos los metodos de la clase o: \n" +
    "LimpiarMostrarMetodos nombre.clase.calificado palabra\n" +
    "Para buscar todos los metodos que involucran a 'palabra'";
  public static void main(String[] args) {
    if(args.length < 1) {
      System.out.println(uso);
```

```
        System.exit(0);
      }
      try {
        Class c = Class.forName(args[0]);
        Method[] m = c.getMethods();
        Constructor[] ctor = c.getConstructors();
        // Convertirlo en un array de Strings limpios:
        String[] n =
          new String[m.length + ctor.length];
        for(int i = 0; i < m.length; i++) {
          String s = m[i].toString();
          n[i] = EliminarCalificadores.eliminar(s);
        }
        for(int i = 0; i < ctor.length; i++) {
          String s = ctor[i].toString();
          n[i + m.length] =
            EliminarCalificadores.eliminar(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("Clase inexistente: " + e);
      }
    }
} ///:~
```

La clase **LimpiarMostrarMetodos** es bastante similar a la **MostrarMetodos** previa, excepto en que toma los arrays de **Method** y **Constructor** y los convierte en un único array de **String**s. Cada uno de estos objetos **String** se pasa después a través de **EliminarCalificadores.eliminar( )** para eliminar toda la cualificación de métodos.

Esta herramienta puede ser un verdadero ahorro de tiempo al programar, en las ocasiones en que no se puede recordar si una clase tiene un método en particular y no se desea ir recorriendo toda la jerarquía de clases en la documentación en línea, o si se desconoce si la clase puede hacer algo, por ejemplo, con objetos **Color**.

El Capítulo 13 contiene una versión IGU de este programa (personalizada para extraer información para componentes Swing) de forma que se puede dejar que se ejecute mientras se escribe el código para permitir búsquedas rápidas.

Resumen

RTTI permite descubrir información de tipos desde una referencia a una clase base anónima. Por consiguiente, es muy posible que sea mal utilizada por un novato, puesto que podría cobrar sentido antes de lo que lo cobran los métodos polimórficos. Para mucha gente proveniente de un trasfondo procedural, es difícil no organizar sus programas en conjuntos de sentencias **switch**. Podrían lograr esto con RTTI, y por consiguiente perder el valor importante del polimorfismo en el desarrollo y mantenimiento de código. La intención de java es que se usen llamadas a métodos polimórficos a través del código, y usar RTTI sólo cuando se deba.

Sin embargo, el uso de llamadas a métodos polimórficos como se pretende requiere de un control de la definición de la clase base porque en algún momento de la extensión del programa se podría descubrir que la clase base no implementa el método que se necesita. Si la clase base proviene de una biblioteca o está controlada de alguna forma por alguien más, una solución al problema sería la RTTI: se puede heredar un nuevo tipo y añadir el método extra. En cualquier otro lugar del código es posible detectar el tipo particular e invocar a ese método en especial. Esto no destruye el polimorfismo y la extensibilidad del programa porque la adición de un nuevo tipo no exigirá buscar sentencias switch por todo el programa. Sin embargo, cuando se añade código nuevo en el cuerpo principal que requiera una nueva faceta, hay que usar RTTI para detectar el tipo en particular.

Poner una característica en la clase base podría significar que, en beneficio de una clase particular, todas las otras clases derivadas de esa base requieran algún fragmento insignificante de un método. Esto hace la interfaz menos limpia, y molesta a aquéllos que deben superponer métodos abstractos cuando se derivan de esa clase base. Por ejemplo, considérese una jerarquía de clases que represente los instrumentos musicales. Supóngase que se desea limpiar las válvulas de soplado de todos los instrumentos de la orquesta que las tengan. Una opción sería poner un método **limpiarValvulaSoplado( )** en la clase base **Instrumento**, pero esto es confuso pues implicaría que los instrumentos de **Percusión** y **Electrónicos** también tuvieran válvulas de soplado. RTTI proporciona una solución mucho más razonable porque se puede ubicar el método en la clase específica (en este caso **Viento**), donde es apropiado. Sin embargo, una solución más adecuada es poner un método **prepararInstrumento( )** en la clase base, pero podría no verse cuando se solucione el problema por primera vez y podría asumir erróneamente que hay que usar RTTI.

Finalmente, RTTI solucionará algunos problemas de eficiencia. Si el código hace uso elegantemente del polimorfismo, pero resulta que uno de los objetos reacciona a este código de propósito general de forma horriblemente ineficiente, se puede extraer este tipo usando RTTI y escribir código específico del caso para mejorar la eficiencia. Sin embargo, hay que ser cauto y no buscar la eficiencia demasiado pronto. Es una trampa seductora. Es mejor hacer *primero* que el programa funcione, y decidir después si se ejecuta lo suficientemente rápido, y sólo en ese momento enfrentarse a aspectos de eficiencia.

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *http://www.BruceEckel.com*.

1. Añadir **Romboide** a **Figuras.java**. Crear un **Romboide**, aplicar un molde hacia arriba a **Figura**, y después hacer una conversión de vuelta hacia un **Romboide**. Intentar aplicar un molde hacia abajo a **Círculo** y ver qué pasa.

2. Modificar el Ejercicio 1 de forma que use **instanceof** para comprobar el tipo antes de hacer la conversión hacia abajo.

3. Modificar **Figuras.java** de forma que "resalte" (ponga un *flag*) en todos los polígonos de un tipo en particular. El método **toString( )** de cada **Figura** derivado debería indicar si esa **Figura** está "resaltada".

4. Modificar **Confiteria.java** de forma que se controle la creación de cada tipo de objeto por un parámetro de línea de comandos. Es decir, si la línea de comandos es "**java Confiteria Caramelo**", que sólo se cree el objeto **Caramelo**. Darse cuenta de cómo es posible controlar los objetos **Class** que se crean vía la línea de comandos.

5. Añadir un nuevo tipo de **AnimalDomestico** a **RecuentoAnimalDomestico3.java**. Verificar que se crea y que cuenta correctamente en el método **main( )**.

6. Escribir un método que tome un objeto e imprima recursivamente todas las clases en su jerarquía de objetos.

7. Modificar el Ejercicio 6 de forma que use **Class.getDeclaredFields( )** para mostrar también información de los campos de una clase.

8. En **PruebaJuguete.java**. marcar como comentario el constructor por defecto de **Juguete** y explicar lo que ocurre.

9. Incorporar un nuevo tipo de interfaz a **PruebaJuguete.java** y verificar que se detecta y muestra correctamente.

10. Crear un nuevo tipo de contenedor que use un **private ArrayList** para guardar los objetos. Capturar el tipo del primer tipo que se introduce en él y permitir al usuario insertar objetos sólo de ese tipo a partir de ese momento.

11. Escribir un programa para determinar si un array de **char** es un tipo primitivo o un objeto auténtico.

12. Implementar **limpiarValvulaSoplado( )** como se describe en el resumen.

13. Implementar el método **rotar(Figura)** descrito en este capítulo de forma que compruebe si está rotando un **Círculo** (y si es el caso, que no lleve a cabo la operación).

14. Modificar el Ejercicio 6 de forma que use la reflectividad en vez de RTTI.

15. Modificar el Ejercicio 7 de forma que use la reflectividad en vez de RTTI.

16. En **PruebaJuguete.java**, utilizar la reflectividad para crear un objeto **Juguete** usando un constructor distinto del constructor por defecto.

17. Buscar la interfaz de **java.lang.Class** en la documentación HTML de Java que hay en *http://java.sun.com*. Escribir un programa que tome el nombre de una clase como parámetro de línea de comandos, y después use los métodos **Class** para volcar toda la información disponible para esa clase. Probar el programa con una biblioteca estándar y una clase creada por uno mismo.