FACULTAD DE INGENIERIA
Universidad de Buenos Aires

RTTI y reflexión
Modelos de datos

Carlos Fontela
cfontela@fi.uba.ar

A3F

Temario

RTTI (información de tipos en tiempo de ejecución)

Reflexión

Modelo de datos y memoria de Java

Otros modelos

Transformaciones de tipos

Transformación automática

```
Elipse e = new Elipse();
Figura f = e;
```

Transformación explícita

```
Elipse e2 = (Elipse)f;
// f.setRadioMayor(3);  f es una Elipse, pero...
(Elipse)f.setRadioMayor(3);
```

Se hace chequeo

¡Todo objeto conoce su tipo!

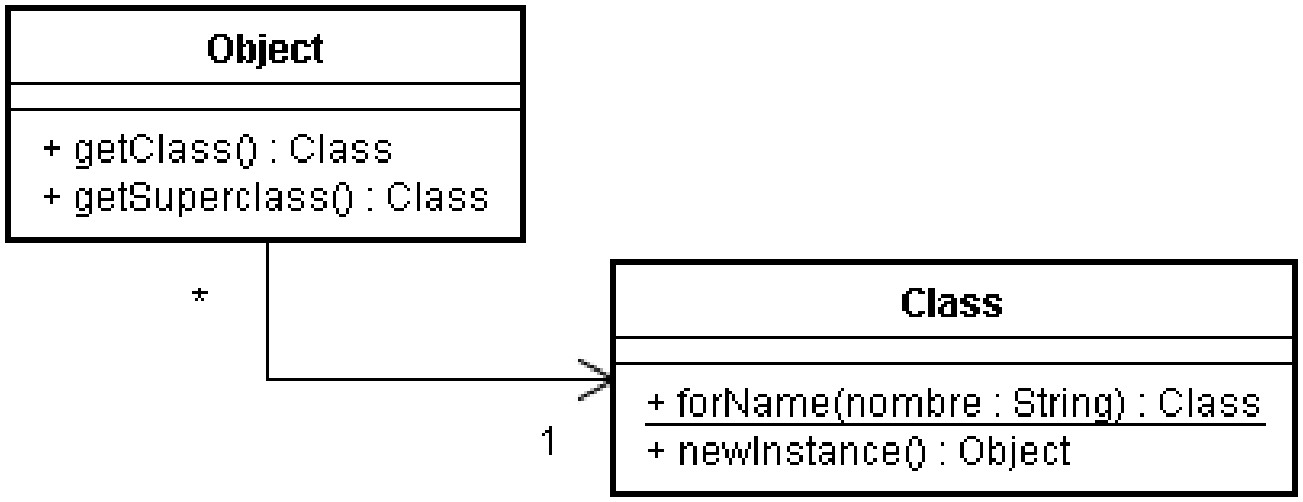
¡El tipo del objeto nunca cambia!

¿Cómo es esto? => RTTI

RTTI (1)

Hay una clase Class y está Object

- Cuando se crea un objeto, se guarda en un atributo de Object una referencia a un objeto Class
- El objeto Class se crea al compilar la clase



RTTI (2)

Versión no polimorfa:

```
if (f.getClass( ) == Elipse.class)  return ((Elipse) f).getCentro( );

if (f.getClass( ) == Class.forName("Elipse"))  return ((Elipse) f).getCentro( );
```

Versión polimorfa:

```
if (x instanceof Elipse) return ((Elipse) f).getCentro( );
```

C#

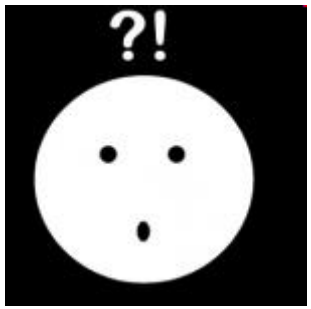
No polimorfo: GetType()

Polimorfo: operador “is”

Reflexión (1)

Cosas raras (faltan excepciones):

```
public void metodoRaro (Object o1) throws Exception {
    Class claseAncestro = o1.getSuperclass();
    Object o2 = claseAncestro.newInstance();
    Method m = o2.getClass().getConstructors()[0];
    m.invoke();
}
```



Reflexión (2)

RTTI: el compilador debe conocer los tipos

¿Qué pasa si recibo un objeto por una red?

Puedo obtener el objeto Class y preguntar por métodos, atributos, interfaces, etc.

Paquete java.lang.reflect

La información debe estar en tiempo de ejecución

En general se maneja en forma automática

Interrogación a componentes para saber qué eventos soporta

Reflexión (3)

Clases

- Method, Constructor, Field

Métodos

- getMethods(), getConstructors(), getFields(), getInterfaces(), getSuperclass()
- newInstance()
- En clase Field: get() y set()
- En clase Method: invoke()

Mucho más

Un uso de reflexión: creación de objetos

```
Fraccion f = new Fraccion();
```

La clase está especificada en el código y se conoce en tiempo de compilación

¿Qué pasaría si hiciésemos?:

```
Serializable f = x.crearObjeto();      // Serializable es una interfaz
```

El método podría ser:

```
public Serializable crearObjeto ( ) {
    String nombreClase = leerArchivoConfiguracion();
    Class claseInstanciar = Class.forName(nombreClase);
    Object nuevo = claseInstanciar.newInstance();
    return (Serializable) nuevo;
}
```

Ahora la clase se conoce recién en tiempo de ejecución

Reflexión: ¿ya la usamos?

Framework JUnit

¿Cómo funciona?

Usa polimorfismo

Métodos setUp() y tearDown()

Y reflexión

Métodos “public void testXxx()”

Algo bastante común en todos los frameworks

Encajonamiento

"Autoboxing"

(la traducción es mía)

Java

```
int x = 4;
Integer i = x;        // equivale a Integer i = new Integer(x);
int y = (int) i;  // equivale a int y = i.intValue( );
Collection le = new ArrayList( ); le.add(5);
```

Otros lenguajes

C#

- Misma funcionalidad que en Java
- Encajonamiento con mejor desempeño
- Los “arreglos primitivos” son todos subclases de System.Array
- Todos los tipos por valor “heredan” de System.ValueType

C++

- Poca información de tipos y sin reflexión
- Distintas formas de transformaciones de tipos: muy complejo
- No hay encajonamiento ni todos los tipos son clases

Malos usos: ojo con RTTI

Compromete la extensibilidad

```
public Punto algunaPosicion ( ) {
    if (this instanceof Elipse)
      return ((Elipse) this).getCentro( );
    if (this instanceof Poligono)
      return
          ((Poligono) this).getContorno( )[0];
    throw new IllegalArgumentException( );
}
```

Evita el polimorfismo

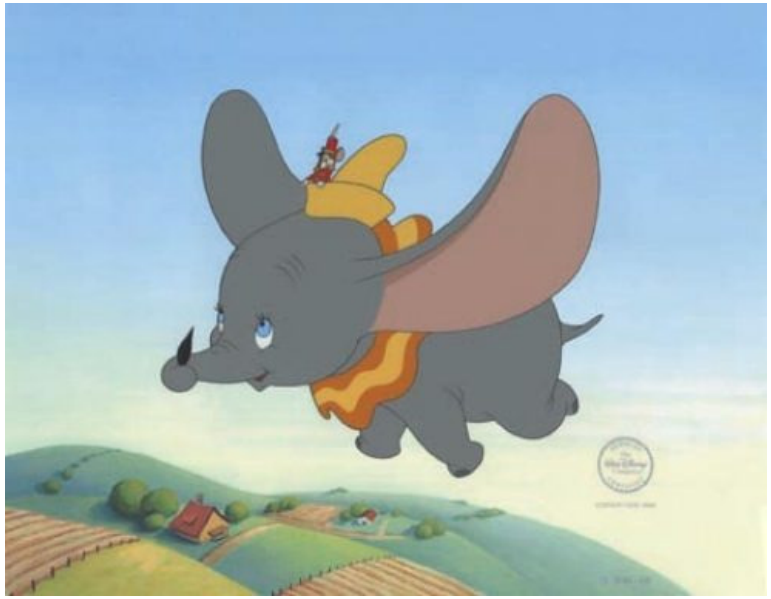
```
if (x instanceof Elipse) ((Elipse)x).dibujar( );
```

Muy contrario a POO en general

Malos usos: ojo con reflexión

Podemos terminar generando cualquier cosa

- Síndrome del elefante alado



Difícil de testear

Difícil de leer

No cualquiera la usa bien

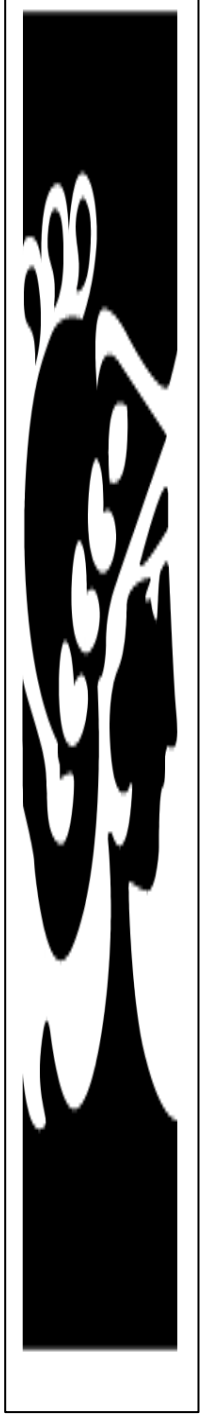
Modelo de memoria de Java

Referencias

- Ojo con comparaciones
  - == y equals()
- Ojo con asignaciones
  - = y clone()

Recolección de basura

- Ojo con acciones finales
  - finalize(), System.runFinalization() y System.gc()



Igualdad e identidad

```
if (a == b)          // compara referencias
if a.equals(b)       // compara contenido
```

equals:

- Está definido en Object
- Está implementado en colecciones
- No hace comparaciones profundas

Asignación simple

```
Object a = b;     // asigna referencias

Object o, b;
o.m(b);         // b pasa como referencia

int x = y;// asigna valores

int y;
Object o;
o.m(y);         // y pasa como valor
```

Clonación

a = b.clone();

En Object está definido:

protected Object clone() throws CloneNotSupportedException

{ // hace una copia de bits... }

Hay que redefinir clone como público y que llame a super.clone()

Si quiero una copia profunda, implementarla

Ver ejemplo en libros

Hay que implementar Cloneable

Si no, obtenemos CloneNotSupportedException

Recolección de basura

- No determinística
- Asegura que
  - No me voy a quedar sin memoria mientras haya objetos sin referenciar
  - No se va a liberar ningún objeto que esté siendo referenciado desde un objeto referenciado
- Extremadamente cómoda
- Y evita errores muy difíciles de encontrar y reparar

Finalización: finalize()

No es un destructor

Se ejecuta cuando pasa el recolector de basura

- No sabemos cuándo
- Puede que nunca
- No sabemos nada del orden, como para llamar un finalize() desde otro finalize()

Solución a medias para liberar recursos

Liberación forzada

System.gc();

- JVM hace “su mejor esfuerzo” por recolectar todo.
- Lleva mucho tiempo.
- Puede usarse antes de crear una gran estructura.

System.runFinalization();

- JVM hace “su mejor esfuerzo” por finalizar todo, sin recolectar.
- Iguales consideraciones.

C# (1)

El modelo es de referencias para clases y otros tipos

- Con recolección automática de basura
- Hay “destructores”, pero es otro nombre para un finalizador
- Hay métodos Clone y Equals
- Pero también sobrecarga de == y !=

C# (2)

Hay tipos de datos estáticos

- structs, enums, etc.
- Se manejan como en C++
- Pueden implementar interfaces
- Pero no tienen herencia

Hay aritmética de punteros

- “unsafe”, “fixed”, y otras restricciones

Object Pascal

- El modelo es de referencias
  - Pero hay clases por valor
- No hay recolección automática de basura
  - Se deben crear destructores
    - En general son virtuales y se los llama desde un método Free

      destructor Destroy; virtual;
- Hay aritmética de punteros
  - Por herencia de Borland Pascal

C++

El modelo es estático

- Pero hay punteros

Existen los destructores

- No necesariamente para liberar memoria

Hay aritmética de punteros

Smalltalk

El modelo es de referencias

Hay recolección automática de basura

Es un lenguaje de tipos dinámicos

- Polimorfismo automático

No hay aritmética de punteros

Interpretado

Claves

Todo objeto conoce su tipo, y es permanente: RTTI

Se le puede preguntar de todo a un objeto: reflexión

Usar RTTI y reflexión con medida

La clonación y equals() solucionan problemas de usar referencias

finalize(), gc() y runFinalization() proveen sucedáneos a medias de los destructores

Lecturas opcionales

Thinking in Java, Bruce Eckel

- Capítulo 4, “Initialization & Cleanup”
- Capítulo 12, “Run-time Type Identification”
- Apéndice A, “Passing & Returning Objects”
- Está en biblioteca
- Hay versión castellana

Orientación a objetos, diseño y programación, Carlos Fontela 2008

- Capítulo 20 “Los datos, los tipos y la memoria”

Qué sigue

Metaprogramación y polimorfismo en Smalltalk