Modelo de Objeto en Java

Agustín J. González
ELO 326: Seminario II
2do. Sem 2001

Objetos, Punteros y Referencias

- Referencias:
  - En Java no existen los pasos por referencia. No se puede cambiar el valor del parámetro.
  - Como en java todos los objetos son accedidos via referencias, podemos cambiar el objeto, pero no el nombre usado para referenciarlo.
  - Las variables parecen referencias pero actúan como punteros.
- Argumentos implícitos
  - Son los miembros datos a los cuales tenemos acceso sin ser pasados como parámetros.
  - Se acceden directamente. Si hay conflicto usamos la palabra reservada this.
  - Ojo otro usa para esta palabra reservada es cuando llamamos a otro constructor.

Objetos, Punteros y Referencias

- Retorno de Referencias
  - Si retornamos un objeto, éste puede ser alterado por el código que llamó!
  - Class Employee {
    public Department getDepartment() { return _department }
    private Department _department;
    }
  - En C++ este código no es problema (se retorna una copia de la fecha de contratación). En Java se retorna una referencia que puede ser alterada si la clase Department posee algún método para ello.
  - Lo que se debe hacer en estos casos es crear un clono y retornarlo.
  - En C++ no tiene sentido retornar referencias a variables locales. En Java esto no constituye problema.

```
Double & FOO::getData() {
        double z=0.5;
        return z; // plop!!
}
```

```
public Point puntoCruce (Segmento b) {
    .....
    return new Point(x,y);  // OK!
}
```

Objetos, Punteros y Referencias

- Copia de Objetos y referencias
  - En C++ las variables son usadas para mantener los valores de los objetos, no para acceder a ellos. La semántica es similar al acceso a enteros.
  - En Java es muy diferente. Las variables nos permiten acceder a objetos, no para almacenarlos. Para forzar la copia una operación especial ***clone*** debe ser incorporada.
  - Es una buena idea mantener como clases inmutables aquellas que poseen la semántica de un valor, como String, Date, etc.. Así no hay problemas de múltiples referencias al mismo objeto.

Manejos de tipos en tiempo de ejecución

- Obtención del tipo
  - La clase Object posee el método getClass, el cual retorna en tiempo de ejecución la clase de un objeto.

```
Class c = s.getClass();
System.out.println(c.getName()); // para depuración de programas.
```

- Casteo
  - En Java se usa el antiguo estilo de C.

```
Rectangle r = (Rectangle) s;
```

  - El operador instanceof permite chequear el tipo de un objeto.

```
If (s instanceof Rectangle) {
  Rectangle r = (Rectangle) s;
 .....
}
```

Copia baja y profunda en C++
(Shallow and Deep copying)

- En C++ si querremos copiar todas las entradas de un vector que contiene punteros a figuras, podemos copiar sólo los punteros. Tal copia es la copia baja (shallow)

```
vector <Shapes *> fig1;
.....
vector <Shapes *> fig2 (fig1.size());
for (int i=0; i<fig1.size(); i++)
        fig2[i] = fig1[i];
```

- Si una forma de fig1 es cambiada también cambia la forma de fig2.
- Algunas veces este comportamiento no es deseado. En estos casos se incorpora una función para copiar cada uno de los objetos en su totalidad. Ésta es la copia en profundidad. Se implementa con la función llamada clone.

Copia baja y profunda en C++
(Shallow and Deep copying)

- ```
  Class Shape {
  public:
          virtual Shape * clone() const=0;
          ....
  };
  ....
  Shape* Point::clone() const  {
          return new Point(*this);
  }
  ....
  Shape*Rectangle::clone() const {
          return new Rectangle(*this);
  }
  ```
- Luego usamos:
  ```
  for (int i = 0; i <fig1.size(); i++)
          fig2[i] = fig1[i] ->clone();
  ```
  ```

Copia baja y profunda en Java
(Shallow and Deep copying)

- Las copias en Java son siempre bajas (shallow).
- Es así como necesariamente debemos hacer las copias en profundidad cuando lo requiramos.
- En Java es un poco más sofisticado la implementación de la función clone.
- La clase Object tiene esta función definida como protegida y chequea si la clase implementa la interfaz Cloneable. Si no lo hace, envía una excepción.
- La implementación típica es como sigue:

  .....

Copia baja y profunda en Java
(Shallow and Deep copying)

- La implementación típica es como sigue:

```
class Rectangle implements Cloneable {
	public Object clone() {
		try {
			Rectangle r = (Rectangle) super.clone();
			r._topleft =(Point) _topleft.clone();
			r._bottomright =(Point) _bottomright.clone();
		} catch (CloneNotSupportedException e ) {
			return null;
		}
	}
	.....
	private boolean _edge;
	private Point _topleft;
	private Point _bottomright;
}
```

Copia baja y profunda en Java
(Shallow and Deep copying)

- Algunas consecuencias:
- b = a; // C++   ====>   b = a.clone();   // java
- f(a); // C++   =====> f(a.clone());   // java
  A menos que sepamos que f no cambia el contenido de a.

Igualdad de Objetos

- En Java el operador igualdad == siempre chequea por igualdad de referencias, nunca por la igualdad de contenidos.
  Ej. String oh = “o”;
  if (“Hello” == “Hell”+oh) ...
  Siempre fallará, ya que la concatenación genera un nuevo string en otra posición de memoria.
- En su lugar debemos usar:
  if(“Hello”.equals (“Hell”+oh))....
- Se debe definir la operación equals para las clases que deseemos usar en un contenedor (por verse más adelante).
- Class Rectangle extends Shape {

```
        public boolean equals (Rectangle b) {
                if (!getClass().equals(b.getClass())) return false;
                return _topleft.equals(b._topleft) &&
                        _bottomright.equals(b._bottomright) &&
                        (edge == b.edge);
        }
        ...
}
```

Constructores virtuales en C++

- Supongamos que tenemos que almacenar una figura (conjunto de formas) en un archivo y luego recuperarlas.
- En C++

```
fstream out ("shapes.dat");
for (int i=0; i<shapes.size(); i++)
{       out << typeid(shapes[i]).name() << "  ";
        shapes [i]-> print(out);
        out << endl;
}
```

- La salida podría ser:

```
        Point  (30,40)
        Rectangle  (10,10) (50,60)
        Ellipse  (100, 100)  10  20
```

- Para la lectura de los objetos, nos enfrentamos al problema de construir un objeto a partir de un string. Esta es la tarea conocida como el constructor virtual. (Obviamente los constructores no se pueden definir como virtuales)

Constructores virtuales en C++

- Para hacer la lectura usamos una tabla de mapeo.
- Map <string, Shape*> shapeTypes;
  shapeTypes[“Point”] = new Point();
  shapeTypes[“Rectangle”] = new Rectangle();
  shapeTypes[“Ellipse”] = new Ellipse();
  ....
- Luego usamos:

```
fstream in (“shapes.dat”)
while (!in.fail())
{       String typename;
        in >> typename;
        Shape* newShape = shapeTypes[typename] ->clone();
        newShape->read(in);
        shapes.push_back(newShape);
}
```

Constructores virtuales en Java

- A diferencia de C++, Java posee soporte para la construcción de un objeto a partir del nombre de la clase.
- La clase Class posee un método estático forName(s) el cual retorna un objeto Class correspondiente al string s. Ej Class.forName("Ellipse") retorna un objeto Class que describe la clase Ellipse.
- Una vez con el objeto Class, se invoca el método newInstance() para crear una nueva instancia de esa clase. El objeto es construido usando el constructor por defecto.
- En Java el código para la lectura de objetos desde archivo sería como:

```
Shape newShpe = (Shape) Class.forName(s).newInstance();
newShape.read(in);
shapes.addElement(newShape);
```

Más sobre clase Class

- Dado un objeto, en java podemos consultar los campos, métodos, y constructores de la clase.
- Hay tres clases para describir estos miembros:Field, Method, Constructor.
- Ej de uso:

```
Method [] operations = obj.getClass().getMethods();
for (int i=0; i<operations.length; i++)
{       Method op = operations[i];
        Class retType = op.getReturnType();
        Class [] paramTypes = op.getParameterTypes();
        String name = op.getName();
        System.out.print(retType.getName() + " " + name + "(" );
        for (int j=0; j < paramTypes.length ; j++)
        {           if (j>0) System.out.print(",  ");
                    System.out.print(paramTypes[i],getName();
        }
        System.out.println( ");" );
}
```

Métodos y Valores final

- En java por defecto todas las funciones son ligadas dinámicamente.
- Si deseamos que esto no ocurra, declaramos la función como final.

```
Class Employee
{       public final double salary()  { return _salary}
        .....
}
```

- El compilador puede remplazar la función por su código.
- Se impide que una clase derivada redefina el método.
- En el caso de variables, éstas no pueden ser reasignadas. Equivalen a constantes.
- En el caso de objetos, el calificativo final indica que el nombre no se puede referir a otro objeto; sin embargo el objeto puede ser cambiado.
- Si una clase es definida como final, no puede dar origen a clases derivadas.

Datos Estáticos

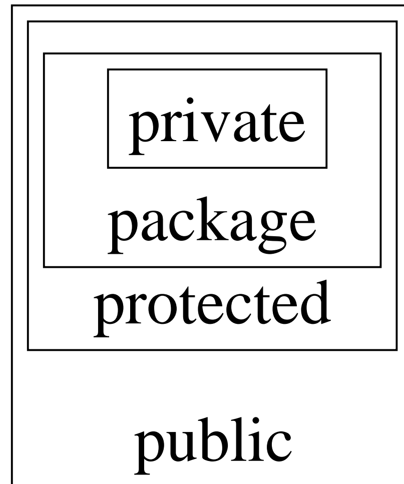
- Permite declarar datos compartidos entre todas las clases, igual idea que en C++.
- Ej:

```
class Date // Java
{ ...
        Private static int days_per_month[] = {
                                31, 28, 31, 30, 31, 30,
                                31, 31, 30, 31, 30, 31 };
 }
```

- Si la inicialización es más compleja, puede ser hecha en un bloque llamado static, este código es llamando cuando la clase es cargada.

```
class Date // Java
{  ....
        Private static int days_per_month [];
        static
        {  days_per_month = new int [12];
           for (int i=0; i < 12 ; i++)
           {    if (i > 0 && i %5==0 || i%5 ==3 )
                        days_per_month [i]=30;
                else ...
           }
        }
}
```

Calificadores de alcance



- Private member: Sólo puede ser accedido por miembros de la misma clase
- Package member: Puede ser accedido libremente por código en el paquete. Es el nivel de acceso por defecto (cuando omitimos este control de acceso).
- Protected member: Sólo pude ser accedido desde el mismo paquete o por una subclase que lo hereda.
- Public member: Puede ser accedido por cualquier código que tiene acceso a la clase.
- Ej.

```
package my_package;
class myClass
{
        int i;   // acceso a todas las clases del paquete
private int j; // sólo a la case myClass
...
}
```

Compilación separada en Java

- Java no sigue el esquema de C++ para la compilación separada.
- EL compilador mira automáticamente dentro de los archivos de clases de las clases referenciadas en el archivo bajo compilación. Así descubre las características de esas clases.
- Java usa una variable de ambiente llamada ***classpath*** para que el programador le indique en qué directorios buscar por las clases en uso.
- Ej:
  export JAVA=~/jdk1.3.1
  export CLASSPATH = $JAVA/jre/lib/ext/mlibwrapper_jai.jar:
  ../netSupport/: ../common/: ../sender/: ../recv/: ./

- Otra forma de especificar este path es indicarlo en la compilación:
- java -classpath <directories and zip/jar files separated by :>
  set search path for application classes and resources
- Es posible hacer un makefiles para la compilación (así se debe hacer en tareas del curso).

Makefile para Java

- #
- ###
  ###############
- # Macro definitions for Java compilations
- #
- # Edit the next 3 definitions. After that, "make" or "make java" will
- #   build the program.
- #
- # As you define new classes, add them to the following list.
- # It may not be absolutely necessary, but it will help guarantee that
- # necessary recompilation gets done.
- #
- CLASSES= prog1.class prog2.class
- #
- # Define special compilation flags for Java compilation. These may change when
- # we're done testing and debugging.
- JOPTIONS=-g -nowarn
- #
- # What is the name of the class containing the main() function?
- JTARGET=prog1
- #
- # Because locations/versions of Java compilers may vary, you
- # may need to alter these to reflect the system on which you are working/
- # Sigue al frente.......

- JAVAPATH=/research/java/jdk1.3/bin
- CLASSPATH=.:/research/java/jdk1.3/lib/classes.zip

- # The following is "boilerplate" to set up the standard compilation
- # commands:
- #
- .SUFFIXES:
- .SUFFIXES: .class .java
- .java.class: export CLASSPATH; CLASSPATH=$(CLASSPATH); $(JAVAPATH)/javac $(JOPTIONS) $*.java

- #
- # Targets:
- #
- all: $(CLASSES)

- java: $(CLASSES)

- javarun: $(CLASSES)
- export CLASSPATH; CLASSPATH=$(CLASSPATH); $(JAVAPATH)/java $(JTARGET)

- clean:
- -rm -f $(CLASSES)