Object Oriented Software Design

Inner classes, RTTI, Tree implementation

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

October 21, 2011

Outline

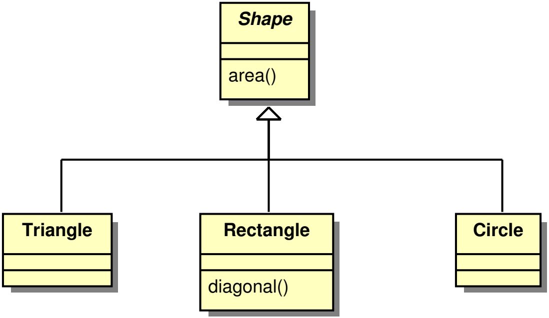
1. Run-Time Type Identification
2. Anonymous inner classes
3. Binary Trees

Outline

1. Run-Time Type Identification
2. Anonymous inner classes
3. Binary Trees

The shape example

- Consider a hierarchy of **Shape** classes



- The **Shape** class is abstract, it has an abstract method to compute the area of the shape.
- Now suppose we have an array of **Shape**s, and we would like to compute the area for all of them.

Shapes

- This is the base class

oosd/shapes/Shape.java

```
package oosd.shapes;

public abstract class Shape {
    protected String name;

    public Shape(String s) { name = s; }

    public abstract double area();
}
```

Triangle

- And one derived class:

oosd/shapes/Triangle.java

```
package oosd.shapes;

public class Triangle extends Shape {
    private double base = 0, height = 0;
    public Triangle() { this("Triangle"); }
    public Triangle(String s) { super(s); }
    public Triangle(String s, double b, double h) {
        this(s);
        base = b;
        height = h;
    }

    public double area() {
        System.out.println("Computing the area of Triangle " + name);
        return base * height / 2;
    }
}
```

A list of Shapes

- Let's use the QueueListIt class we have seen last lecture

```
QueueListIt<Shape> myq = new QueueListIt<Shape>();

// upcast, never fails
myq.push(new Circle("red", 5.0));
myq.push(new Triangle("yellow", 3.0, 4.0));
myq.push(new Rectangle("blue", 3.0, 4.0));

for (Shape s : myq)
  System.out.println(s.area());
```

- Everything as expected

A new method

- **Rectangle** derives from **Shape**, and adds a new method `diagonal()` to compute the diagonal

oosd/shapes/Rectangle.java

```
public class Rectangle extends Shape {
    private double base = 0, height = 0;
    public Rectangle(String s) { super(s); }
    public Rectangle() { this("Rectangle"); }
    public Rectangle(String s, double b, double h) {
        this(s);
        height = h;
        base = b;
    }

    public double area() {
        System.out.println("Computing the area of Rectangle " + name);
        return base * height;
    }

    public double diagonal() {
        System.out.println("Computing the diagonal");
        return Math.sqrt(base*base+height*height);
    }
```

How to call diagonal?

- We would like to call diagonal only for **Rectangle**s because it does not make sense to call diagonal for **Circle**s and **Triangle**s
- But, we have a problem:

```
for (Shape s : myq) {
  System.out.println(s.area());
  System.out.println(s.diagonal());
}
```

Compilation error: `diagonal()` is not part of the interface of **Shape**

Downcast

- We could force s to become a reference to **Rectangle**, so that `diagonal()` is in the interface now.

```
for (Shape s : myq) {
  System.out.println(s.area());
  System.out.println(((Rectangle)s).diagonal());
}
```

- This is called **downcast**, and must be explicit, because a Shape is not (always) a rectangle
- **Downcast** is not safe
  - Unfortunately, if s is pointing to a **Triangle**, Java run-time raises an exception **ClassCastException**

Solution 1: catch the exception

- By catching the exception, everything works fine ClientExc.java

```
import java.util.*;
import oosd.shapes.*;
import oosd.containers.*;

class ClientExc {
    public static void main(String args[]) {
        QueueListIt<Shape> myq = new QueueListIt<Shape>();
        myq.push(new Circle("red", 5.0));
        myq.push(new Triangle("yellow", 3.0, 4.0));
        myq.push(new Rectangle("blue", 3.0, 4.0));

        for (Shape s: myq) {
            System.out.println(s.area());
            try {
                double d = ((Rectangle)s).diagonal();
                System.out.println(d);
            } catch(ClassCastException e) {
                System.out.println("Not a rectangle");
            }
        }
    }
}
```

Upcasting and downcasting

- When we insert in the QueueListIt class, the perform an **upcast**
  - Upcast is always safe.
- To understand if there is a Rectangle, we perform a **downcast**.
  - Downcast is not safe at all (raises an exception), and it should be avoided when necessary.
  - to perform downcast Java has to identify the actual object type and see if the cast can be performed.

instanceof

- To avoid the exception (which is clumsy and inefficient), you can use the keyword **instanceof**

ClientRTTI.java

```
import oosd.containers.*;

class ClientRTTI {
    public static void main(String args[]) {
        QueueListIt<Shape> myq = new QueueListIt<Shape>();
        myq.push(new Circle("red", 5.0));
        myq.push(new Triangle("yellow", 3.0, 4.0));
        myq.push(new Rectangle("blue", 3.0, 4.0));
        Iterator<Shape> it = myq.iterator();
        while (it.hasNext()) {
            Shape s = it.next();
            System.out.println(s.area());
            if (s instanceof Rectangle)
                System.out.println(((Rectangle)s).diagonal());
        }
    }
}
```

- **instanceof** works well with inheritance

The Class object

- All information on a specific class are contained in a special object of type **Class**.
- The class **Class** contains a certain number of methods to analyse the interface:
  - **forName(String s)** returns a Class Object given the class name
  - **isInstance(Object o)** returns true if the specified object is an instance of the class
- An example in the next slide

The usage of Class

ClientRTTI2.java

```
class ClientRTTI2 {
    public static void main(String args[]) {
        QueueListIt<Shape> myq = new QueueListIt<Shape>();

        // upcast, never fails
        myq.push(new Circle("red", 5.0));
        myq.push(new Triangle("yellow", 3.0, 4.0));
        myq.push(new Rectangle("blue", 3.0, 4.0));
        Iterator<Shape> it = myq.iterator();
        while (it.hasNext()) {
            Shape s = it.next();
            System.out.println("Object of class: " + s.getClass().getName(
                               " in package: " + s.getClass().getPackage()
            System.out.println("Object is compatible with Rectangle: " +
                               Rectangle.class.isInstance(s));
            // Rectangle.class is equivalent to
            // Class.forName("Rectangle").isInstance(s)
            System.out.println(s.area());
        }
    }
}
```

Downcasting?

- In the previous example, there was another option: put `diagonal()` in the interface of the base class **Shape**
  - The `diagonal()` function in the **Shape** class needs to be a void function, that could also raise an exception (for example **OperationNotImplemented**)
  - This approach may generate *fat interfaces*
- In this case, we chose to follow the other option
- However, the downcast option is not always the best one, it depends on the context
- This has nothing to do with the specific Java Language: it is a design problem, not a coding problem
- We will come back to the problem of downcasting when studying the *Liskov's substitution principle*.

Outline

Inner classes

- Let's have a closer look again at the **QueueListIt**$<$**E**$>$:
  `./examples/10.java-examples/oosd/containers/QueueL`
- The **QLIterator** class is a private inner class of **QueueListIt**
  - The reason for making it private is that **QLIterator** is an implementation of the more general notion of **Iterator**
  - A different implementation is fine, as long as it conforms with the interface
  - The user does not need to know the implementation, only the interface (i.e. *how to use it*)
  - The user will never directly create a **QLIterator** object: it asks the container class to do the creation for him.
- Advantages:
  - We can change the internal implementation without informing the user, that can continue to use its code without modifications
  - We have achieved perfect modularity

Anonymous inner classes

- Sometimes, interfaces are so simple that creating a private inner class with its own name seems too much;
- Java provides a way do define classes **on the fly**

```
interface MyInterface {
  int myfun();
}
class MyClass {
  ...
  MyInterface get() {
    return new MyInterface() {
      public int myfun() { ...}
    };
  }
}
```

A simple interface

An anonymous class

- Notice the special syntax: **new** followed by *the name of the interface*, followed by the *implementation*
  - The class has no name, so you cannot define a constructor

Anonymous iterator

oosd/containers/QueueListItAn.java

```
    public Iterator<E> iterator() {
        return new Iterator<E>() {
            private Node curr = head;
            private Node prev = null;
            public boolean hasNext() {
                if (curr == null) return false;
                else return true;
            }
            public E next() {
                if (curr == null) return null;
                E elem = curr.elem;
                prev = curr;
                curr = curr.next;
                return elem;
            }
            public void remove() {
                if (prev == null) return;
                // remove element
                Node p = prev.prev;
                Node f = prev.next;
                if (p == null) head = f;
                else p.next = f;
                if (f == null) tail = p;
                else f.prev = p;
                prev = null;
            }
        };
    }
```

Anonymous classes

- It is surely shorter:
  - However, in certain cases it can become cumbersome and confusing, especially when there is too much code to write
    - If there is too much code to write (as in our example), I prefer to write a regular inner class
  - I recommend to minimise the use of anonymous classes
  - However, it is important to understand what do they mean when you meet them in somebody else code

Outline

Binary trees

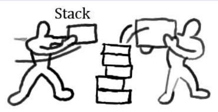
- We will need a binary tree to organise the data for the assignment
- Before we look into trees, however, let's have a look at another common container, which is widely used in many applications: the **Stack**
- The stack may be useful for storing partial results
  - For example, when we have to multiply the results of two sub-expressions, we must first compute the sub-expressions;
  - The partial results may be stored into a stack, and retrieved when needed
- Example: $(3 + 2) * (6 - 4)$
  - Compute $3 + 2$, and put the result 5 on the stack
  - Compute $6 - 4$ and put the result on the stack
  - Extract the last two results from the stack, and multiply them

Stack

- A stack is a very simple data structure.
- A stack can hold a set of uniform data, like an array (for example, integers)
- Data is ordered according to the LIFO (Last-In-First-Out) strategy

Two main operations are defined on the data structure:

- Push: a new data in inserted in the top of the stack
- Pop: data is extracted from the top stack



Usually, we can also read the element at the top of the stack with a peek operation

Stack usage

- In the following program, we use the standard Java implementation of Stack

StackDemo.java

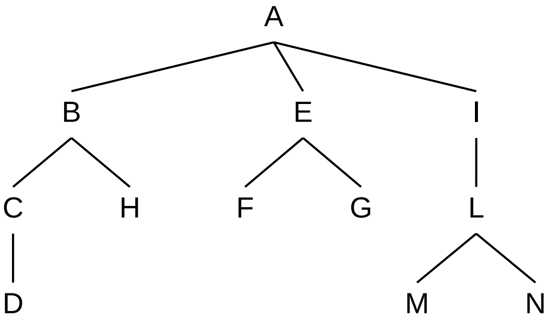
```
import java.util.*;

class StackDemo {
    public static void main(String args[]) {
        Stack<Integer> mystack = new Stack<Integer>();

        for (int i=0; i<10; i++)
            mystack.push(new Integer(i));

        while (!mystack.empty())
            System.out.print(" " + mystack.pop());
        System.out.println("");
    }
}
```

Tree

- A tree is a data structure defined as follows:
  - A tree may contain one or more nodes
  - A **node** in a tree represents an element containing data.
  - A node may have zero or more **child** nodes. The children nodes are called also *descendants*. Each node may have a **parent node**
  - A tree consists of one root node, plus zero or more children sub-trees

Example

- A is the root of the tree
  - B is root of one sub tree of A



Binary trees

- In a binary tree, a node can have at most 2 children
  - Left and right
- A **leaf node** is a node without children
- A **root node** is a node without parents
  - There is only one root node
- Each node in the tree is itself a sub-tree
  - A leaf node is a sub-tree with one single node

How to implement a tree

- One basic data structure is the Node, as in the List data structure
  - In the List structure, a Node had two links, `next` and `prev` (see `./examples/10.java-examples/oosd/containers/QueueLis`
- A possible implementation for a Tree Node is the following:

```
class Node<E> {
  E elem;
  Node left;
  Node right;
}
```

- Optionally, it could contain a link to the parent node
- If one of the links is equal to `null` then the corresponding sub-tree is null

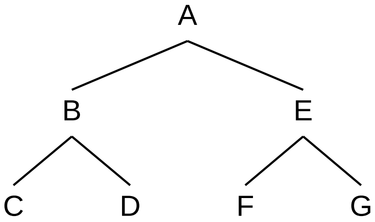
Interface for a Tree

- We must be able to:
  - Create single-node trees
  - Link a sub-tree to single-node tree (to the left or to the right)
  - Get the left (right) sub-tree
- Also, we would like to print the contents of the tree
  - To do this, we need to establish an order of printing

Visiting a tree

- There are two ways of listing the contents of a tree
- **Depth-first**
  - Pre-order: first the root node is visited, then the left sub-tree, then the right sub-tree
  - Post-order: first the left sub-tree is visited, then the right sub-tree, then the root node
  - In-order: first the left sub-tree is visited, then the root node, then the right sub-tree
- **Breadth first**
  - First the root node is visited; then all the children; then all the children of the children; and so on

Example



- Breadth first: A B E C D F G
- Pre-order: A B C D E F G
- Post-order: C D B E F G E A
- In-order: C B D A F E G

Implementation of a Binary Tree

- Let’s start with the node

BTree.java

```
public class BTree<E>  {
    private class Node {
        E elem;
        Node l;
        Node r;

        void addLeft(Node n) {
            l = n;
        }
        void addRight(Node n) {
            r = n;
        }
        Node(E elem) { this.elem = elem; }
    }

    private Node root = null;

    private BTree(Node n) {
        root = n;
    }
```

Adding nodes

BTree.java

```
    public BTree(E elem) {
        root = new Node(elem);
    }

    public BTree<E> addLeft(BTree<E> t) {
        root.addLeft(t.root);
        return this;
    }

    public BTree<E> addRight(BTree<E> t) {
        root.addRight(t.root);
        return this;
    }

    public BTree<E> linkToLeft(BTree<E> t) {
        t.root.addLeft(root);
        return this;
    }

    public BTree<E> linkToRight(BTree<E> t) {
        t.root.addRight(root);
        return this;
    }
```

BTree continued

BTree.java

```
    public BTree<E> getLeftSubtree() {
        if (root == null) return null;
        else return new BTree<E>(root.l);
    }

    public BTree<E> getRightSubtree() {
        if (root == null) return null;
        else return new BTree<E>(root.r);
    }

    void printPostOrder() {
        if (root == null) return;
        else {
            getLeftSubtree().printPostOrder();
            getRightSubtree().printPostOrder();
            System.out.print(root.elem);
            System.out.print(" ");
        }
    }
```

BTree continued

BTree.java

```
    void printPreOrder() {
        if (root == null) return;
        else {
            System.out.print(root.elem);
            System.out.print(" ");
            getLeftSubtree().printPreOrder();
            getRightSubtree().printPreOrder();
        }
    }

    void printInOrder() {
        if (root == null) return;
        else {
            getLeftSubtree().printInOrder();
            System.out.print(root.elem);
            System.out.print(" ");
            getRightSubtree().printInOrder();
        }
    }
```

BTree usage

BTree.java

```
    public static void main(String args[]) {
        BTree<String> mytree = new BTree<String>("*");

        BTree<String> ll = new BTree<String>("+");

        BTree<String> rr = new BTree<String>("-");

        ll.addLeft(new BTree<String>("2")).
            addRight(new BTree<String>("3")).
            linkToLeft(mytree);
        rr.addLeft(new BTree<String>("6")).
            addRight(new BTree<String>("4")).
            linkToRight(mytree);

        System.out.println("Post Order: ");
        mytree.printPostOrder();
        System.out.println("\nPre Order: ");
        mytree.printPreOrder();
        System.out.println("\nIn Order: ");
        mytree.printInOrder();
        System.out.println("\n");
    }
```

A tree Iterator

- In reality, we would like to make the visiting operation more abstract
  - In fact, while visiting we may want to perform other operations than printing
  - For example, evaluating an expression (!)
- Therefore, we need to generalise the algorithm for visiting the tree, and make it independent of the specific operation
  - To do so, we have to modify the structure of the algorithm
  - In the previous program, we have used a simple recursive algorithm
  - Now we need to implement an iterative algorithm, through an iterator
  - The implementation is slightly complex, so pay attention!

The node

BTreeIt.java

```
public class BTreeIt<E>  {
    private class Node {
        E elem;
        Node l;
        Node r;
        Node p;

        void addLeft(Node n) {
            l = n;
            n.p = this;
        }
        void addRight(Node n) {
            r = n;
            n.p = this;
        }
        Node(E elem) { this.elem = elem; }
    }
```

- We now use also the parent link p, because we will need to go up in the hierarchy

Iterator

BTreeIt.java

```
    private class PostOrderIterator implements Iterator<E> {
        Node next;
        Node last;

        PostOrderIterator() {
            next = root;
            last = null;
            moveToLeftMostLeaf();
        }

        private void moveToLeftMostLeaf() {
            do {
                // go down left
                while (next.l != null) next = next.l;
                // maybe there is a node with no left but some right...
                // then go down rigth
                if (next.r != null) next = next.r;
            } while (next.l != null || next.r != null);
            // exit when both left and right are null
        }
```

The next method

BTreeIt.java

```
        public E next() throws NoSuchElementException {
            if (next == null) throw new NoSuchElementException();
            E key = next.elem;
            // I already visited left and right,
            // so I have to go up (and maybe right)
            last = next;
            next = next.p;
            if (next != null && last == next.l) {
                next = next.r;
                moveToLeftMostLeaf();
            }

            return key;
        }
    }

    /* ---------------------------------*/
    /* INTERFACE                        */
    /* ---------------------------------*/

    public BTreeIt(E elem) {
        root = new Node(elem);
    }

    public BTreeIt<E> addLeft(BTreeIt<E> t) {
        root.addLeft(t.root);
```

The tree class

- Just the same, except for the method to return the iterator:

```
    Iterator<E> postOrderIterator() {
        return new PostOrderIterator();
    }
```

- Notice that we do not need the `printXXX()` functions, because we can use the iterator

BTreeIt usage

BTreeIt.java

```
            addRight(new BTreeIt<String>("*").
                     addLeft(new BTreeIt<String>("2")).
                     addRight(new BTreeIt<String>("2"))
                     ).
            linkToRight(mytree);

        System.out.println("Post Order: ");
        Iterator<String> it = mytree.postOrderIterator();
        while (it.hasNext()) System.out.print(it.next() + " ");
        System.out.println("");

    }
}
```

Exercises

- Write the pre-order and the in-order iterators for class **BTreeIt**