Object oriented programming – supervision 2

James Wood

December 1, 2014

a

Dynamic polymorphism is a process by which the method run on an object is decided based on the runtime type of the object. The methods to be chosen from have the same name, where some override others due to being implemented in subclasses. It is useful because it allows the more relevant methods of subclasses to be used without the compiler needing to infer the type of each object.

For example, consider the following code:

```
package code;

import java.util.ArrayList;

class A {
    public void write() {
        System.out.println("Hello␣from␣A");
    }
}

class B extends A {
    @Override
    public void write() {
        System.out.println("Hello␣from␣B");
    }
}

public class S2a {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
```

```
        ArrayList<A> arr = new ArrayList<>();
        arr.add(a);
        arr.add(b);
        for (A e : arr) {
            e.write();
        }
    }
}
```

When calling `write()` from inside the `for` loop, the compiler can only infer that each element of `arr` is of type `A`, and can't infer which of them (if any) are of type `B`. Due to dynamic polymorphism, however, the runtime can see that the second element is actually of type `B`, so the `write` defined for class `B` is called. Output:

```
Hello from A
Hello from B
```

b

A class is a template for an object, and can be extended to make other classes (unless it is final). It can define what type of state and what behaviour its instances will have. An abstract class is a class that cannot be instantiated. It can define type of state and default behaviours for its subclasses. An interface also cannot have instances, but cannot define state either. An interface is implemented, rather than extended, and classes can implement any number of interfaces (as opposed to only being able to extend exactly one class).

c

`List` is an interface defining what behaviours an ordered mutable list should have. `AbstractList` is an abstract class that gives a minimal implementation of the methods in `List`. `Vector`, `LinkedList` and `ArrayList` implement `List` and are non-abstract. `Vector` and `ArrayList` extend `AbstractList`. `Vector` is considered deprecated due to poorly-designed concurrency features, whilst `ArrayList` is still used. Both `Vector` and `ArrayList` are implemented using arrays whose length is updated when the list needs more or less capacity. `LinkedList` is implemented as a doubly-linked list.

The interface `Set` describes what behaviours a collection containing strictly distinct elements (according to the elements' `equals` methods) should have.

`TreeSet` and `HashSet` implement it concretely. `TreeSet` stores its elements in a `TreeMap`, ordered either by their natural ordering or a specified `Comparator` object. `HashSet` stores its elements in a `HashMap`, which offers constant-time operations (as opposed to the typically logarithmic-time operations of `TreeSet`, which uses a red-black tree).

`HashMap` and `LinkedHashMap` implement `Map`, which means that they both store a table of key-value pairs. `LinkedHashMap` is implemented as a doubly-linked list, meaning that most actions run in linear time. It guarantees that its elements will be kept in the order in which their key was added to the map. `HashMap` acieves constant time compexity by storing values in an array with their position being determined by the hash of their key. There is a possibility for key collision for both types if the number of elements approaches the number of available keys.

d

Mutability allows an object to be modified without it being moved to a new part of memory. This is much more efficient if the object is often modified. However, mutability means that the object could be modified by any function it is passed to. This makes reasoning about mutable objects difficult. Reasoning about mutable objects is even more difficult when concurrency is involved.

A Java class can be made immutable by making all member fields private and not providing setter methods. There should also be suitable constructors to allow the fields to be populated.

e

Shadowing occurs when two different entities are given the same name, and both could be available in a single scope. Generally, the more local name will shadow the less local name. For example, a method variable will shadow a member variable.

Example:

```
package code;

class A {
    public int a = 2;
    public void f() {
        System.out.print("A");
```

```
    }
}

class B extends A {
    public int a = 3;
    public void f() {
        System.out.print("B");
    }
}

public class S2e {
    public static void main(String[] args) {
        A x = new A();
        A y = new B();
        B z = new B();
        System.out.print(x.a);
        System.out.print(y.a);
        System.out.print(z.a);
        x.f();
        y.f();
        z.f();
        System.out.println("");
    }
}
```

This outputs `223ABB` because, in Java, field shadowing is decided at compile time but method shadowing is decided at runtime. `y` is an `A` according to the compiler, but a `B` at runtime, which leads to this descrepancy.

f

`public void add(Vector2D v)` would be used with a mutable implementation of `Vector2D`. The statement `a.add(b);` would add the value of `b` onto `a`, modifying `a`.

`public Vector2D add(Vector2D v)` would do the same thing, but also return the new value of `a`. This would make the method work like other assignment operators, like `+=`. Alternatively, it may simply return the result of adding `v` to the base vector without modifying anything.

`public Vector2D add(Vector2D v1, Vector2D v2)` is like the above method, but adds two vectors to the original vector. It doesn't seem that useful.

`public static Vector2D add(Vector2D v1, Vector2D v2)` is analo-

gous to a usual binary operator, but is written as a prefix function. `Vector2D.add(a, b)` would return the sum of `a` and `b` without mutating either of them.

g

	`public`	`protected`	`private`	unspecified
a	free access	free access	compile error	free access
b	free access	free access	compile error	compile error
c	free access	free access	compile error	free access
d	free access	compile error	compile error	compile error

h

```
package code;

interface INinja {
    public void hide();
}

class Student {
    private boolean mSleeping = false;

    public void sleep() {
        mSleeping = true;
        System.out.println("zzz");
    }

    public void wake() {
        try {
            Thread.sleep(10000);
        } catch (InterruptedException ignore) { }

        mSleeping = false;
        System.out.println("yawn");
    }

    public Student(boolean sleeping) {
        mSleeping = sleeping;
    }

    public Student() {
        mSleeping = true;
```

```
    }
}

class Ninja implements INinja {
    @Override
    public void hide() {
        System.out.println("...");
    }
}

class NinjaStudent extends Student implements INinja {
    private Ninja mNinja = new Ninja();

    @Override public void hide() {
        mNinja.hide();
    }
}

public class S2h {
    public static void main(String[] args) {
        NinjaStudent ns = new NinjaStudent();
        ns.wake();
        ns.hide();
    }
}
```

i

Bytecode (comments moved to new line):

```
Compiled from "S2i.java"
public class code.S2i {
  public code.S2i();
    Code:
       0: aload_0
       1: invokespecial #1
          // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: iconst_0
       1: istore_1
       2: iconst_0
```

```
       3: istore_2
       4: iload_2
       5: bipush        10
       7: if_icmpgt     20
      10: iload_1
      11: iload_2
      12: iadd
      13: istore_1
      14: iinc          2, 1
      17: goto          4
      20: getstatic     #2
          // Field java/lang/System.out:Ljava/io/PrintStream;
      23: iload_1
      24: invokevirtual #3
          // Method java/io/PrintStream.println:(I)V
      27: return
}
```

The bytecode for `main` can be split into three sections – initialisation (ending before line 4), loop (ending before line 20) and output (line 20 onward). `istore_n` stores the value at the top of the stack to virtual register `n`. `total` is stored in address `1` and `i` is stored in address `2`. `iload_n` pushes the value from virtual register `n` onto the stack. `iadd` pops two values from the stack, adds them together and pushes the result. `iinc` increments the value at the top of the stack.

j

Subclassing is a special case of subtyping in which the subtype extends the supertype, and both are classes. For a type to subtype another, the only restriction is that the subtype must be able to be cast to the supertype. Subclassing is covered by this, but so are types where the parameters are in a subtype relationship. For example, `ArrayList<B>` is a subtype of `ArrayList<A>` if `B` is a subtype of `A`.

k

Java 7 introduces:

- multi-catch exceptions: in the `catch` line of a `try` block, multiple exception types can be supplied as options. The types are separated by

a `|` and the identifier follows. This allows different types of exception to share the same handling logic without copying code.

- rethrowing exceptions: the exception caught in a `catch` block can be thrown for code further down the call stack to handle. Previously, the exception had to be wrapped in a new exception, and the other code had to be modified to catch this new type of exception (`RuntimeException`, usually), rather than the exception that was actually thrown.

- try-with: a `try` block can be combined with a block that would be called `with` in Python or `using` in C#. A variable of a type implementing `AutoCloseable` can be assigned to in round brackets after the `try` keyword, and it will be usable in the rest of the `try` block. Importantly, a `finally` block will be added, which uses the methods described by `AutoCloseable` to clean up the object. For example, when a file has finished being read, it should be closed to allow other programs to access it. If there were no `finally` block and an error occurred before the file was closed, the file would never be closed.

l

```
package code;

import java.util.TreeMap;
import java.util.ArrayList;
import java.util.Map;
import java.util.Collections;

public class Students {
    TreeMap<String,Double> mStudents = new TreeMap<>();

    public ArrayList<String> alphabeticList() {
        return new ArrayList<>(mStudents.keySet());
    }

    // arr is in descending order on values
    private ArrayList<Map.Entry<String,Double>> insert(
            ArrayList<Map.Entry<String,Double>> arr,
            Map.Entry<String,Double> e) {
        for (int i = arr.size() - 1; i >= 0; i--) {
            if (e.getValue() <= arr.get(i).getValue()) {
```

```
                arr.add(i + 1, e);
                return arr;
            }
        }
        arr.add(0, e);
        return arr;
    }

    public ArrayList<String> topStudents(double percentage) {
        int n = (int)Math.round(
                    (double)mStudents.size() * (percentage / 100));
        ArrayList<Map.Entry<String,Double>> acc = new ArrayList<>(n
            + 1);
        for (Map.Entry<String,Double> e : mStudents.entrySet()) {
            acc = insert(acc, e);
            if (acc.size() == n + 1)
                acc.remove(n);
        }

        ArrayList<String> ret = new ArrayList<>();
        for (Map.Entry<String,Double> e : acc)
            ret.add(e.getKey());
        return ret;
    }

    public Double median() {
        ArrayList<Double> sortedScores =
            new ArrayList<Double>(mStudents.values());
        Collections.sort(sortedScores);
        int l = sortedScores.size();
        if (l % 2 == 0)
            return (sortedScores.get(l / 2 - 1) + sortedScores.get(l
                 / 2))
                / 2;
        else
            return sortedScores.get(l / 2);
    }

    public Students(TreeMap<String,Double> students) {
        mStudents = students;
    }
}
```

m

Stack.java:

```
package s2m;

import java.util.EmptyStackException;
import java.util.List;

public class Stack<T> implements IStack<T> {
    private T mTop;
    private Stack<T> mRest;

    public Stack(Stack<T> s) {
        mTop = s.mTop;
        mRest = s.mRest;
    }

    public Stack(T top, Stack<T> rest) {
        mTop = top;
        mRest = rest;
    }

    public Stack(List<T> elements) {
        if (elements.size() == 0) {
            mTop = null;
            mRest = null;
        }
        else {
            mTop = elements.get(0);
            mRest = new Stack<T>(elements.subList(1,elements.size())
                );
        }
    }

    public Stack() {
        mTop = null;
        mRest = null;
    }

    @Override
    public T top() throws EmptyStackException {
        if (mTop == null)
            throw new EmptyStackException();
        else
```

```
            return mTop;
    }

    @Override
    public void push(T o) {
        mRest = new Stack<T>(this);
        mTop = o;
    }

    @Override
    public T pop() throws EmptyStackException {
        if (mTop == null)
            throw new EmptyStackException();
        else {
            T ret = mTop;
            mTop = mRest.mTop;
            mRest = mRest.mRest;
            return ret;
        }
    }

    @Override
    public int size() {
        return mTop == null ? 0 : 1 + mRest.size();
    }
}
```

StackTest.java:

```
package s2m;

import static org.junit.Assert.*;
import java.util.EmptyStackException;
import org.junit.Before;
import org.junit.Test;

public class StackTest {
    private IStack<String> s;

    @Before
    public void setUp() {
        this.s = new Stack<String>();
    }

    @Test(expected = EmptyStackException.class)
    public void testPopWhenEmpty() {
```

```
        s.pop();
    }

    @Test
    public void testPush() throws Exception {
        s.push("0");
        assertEquals(1, s.size());
        assertEquals("0", s.top());
        s.push("1");
        assertEquals(2, s.size());
        assertEquals("1", s.top());
        s.push("2");
        assertEquals(3, s.size());
        assertEquals("2", s.top());
    }

    @Test(expected = EmptyStackException.class)
    public void testPop() throws Exception {
        s.pop();
        assertEquals(2, s.size());
        assertEquals("1", s.top());
        s.pop();
        assertEquals(1, s.size());
        assertEquals("0", s.top());
        s.pop();
        assertEquals(0, s.size());
        s.pop();
    }
}
```

Test results:

```
<?xml version="1.0" encoding="UTF-8"?><testrun duration="6"
    footerText="Generated␣by␣IntelliJ␣IDEA␣on␣01/12/14␣00:05" name="
    StackTest">
    <count name="total" value="3"/>
    <count name="passed" value="3"/>
    <suite duration="6" status="passed" name="StackTest">
        <test duration="5" status="passed" name="testPop"/>
        <test duration="1" status="passed" name="testPopWhenEmpty"/>
        <test duration="0" status="passed" name="testPush"/>
    </suite>
</testrun>
```