TDA550 – Objektorienterad programvaruutveckling IT, forts. kurs
Övning vecka 2

Pelle Evensen, Daniel Wetterbro*

20 oktober 2011

Sammanfattning

Denna vecka ska vi titta på skillnader mellan primitiva typer och referenstyper, typomvandling mellan primitiva typer samt referenser kontra kopior av referenser samt "aliasing". Vi ska som hastigast också identifiera problem i samt bygga om lite bedrövligt dålig programkod.
Övningarna är graderade (något subjektivt) från lätt (*) till svår (***). Svårighetsgraden hos en övning behöver inte ha någonting med lösningens storlek att göra.

1 Referenser

1.1 Anropssemantik & alias *

Vad borde utdata bli när man kör `main` i klassen **Vektor**[1] på sidan 2?

Skilj på primitiva typer, referenstyper och alias.

Tänk på alias och om ni har *call-by-value* semantik eller (nästan) *call-by-reference*[2]. I Java sker anrop med call-by-value för *primitiva* typer. För *referenstyper* är det alltid call-by-sharing, d.v.s. värdet som skickas är en *kopia av referensen* till ett objekt.

*Med lån från övningar av Joachim von Hacht.

[1]Mer eller mindre tagen från tenta för Objektorienterad Programmering IT LP 1 2009

[2]Java har *egentligen inte* call-by-reference-semantik alls; Det som förekommer är *call by sharing*. Se gärna `http://en.wikipedia.org/wiki/Call_by_value#Call_by_sharing`. Call by sharing kan man också kalla "call by *value of* reference".

```
public class Vektor {
    private int x;
    private int y;
    private int z;

    public Vektor(int x, int y, int z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    public double norm() {
        return Math.sqrt(x * x + y * y + z * z);
    }

    public void add(Vektor v) {
        x += v.x;
        y += v.y;
        z += v.z;
    }

    public void dumt(int x, int y, int z) {
        this.x = ++x;
        this.y = y + 1;
        this.z += z;
    }

    public int getX() {  return x; }
    public int getY() {  return y; }
    public int getZ() {  return z; }

    public String toString() {
        return "Vektor, <x = " + x + ", y = " + y + ", z = " + z + ">";
    }

    public static void main(String[] args) {
        Vektor a = new Vektor(1, 0, 0);
        Vektor b = new Vektor(0, 1, 0);
        Vektor c = a;
        int x = 1;
        int y = 2;
        int z = 3;
        a.add(b);
        b.add(b);
        c.add(c);
        c.dumt(x, y, z);
        System.out.println("a: " + a);
        System.out.println("b: " + b);
        System.out.println("c: " + c);
        System.out.println("x: " + x + "\ty: " + y + "\tz: " + z);
    }

}
```

Figur 1: Klassen **Vektor**.

1.2 Samma kontra lika **

Avgör för klassen **Referenser1** vad utskriften blir. Kör sedan programmet och se om svaret blev vad du trodde det skulle bli. Förklara varför det blev eller inte blev vad du trodde.

För att få en tydlig(?) bild av vad svaren borde bli kan man slå i [GJSB05][3]. Specifikationen är bitvis ganska svårläst men den enda källa man i slutändan egentligen kan lita på.

```
public class Referenser1 {
    public static void main(String[] args) {
        stringRefs();
        integerRefs();
    }

    private static void print(String exp, boolean b) {
        System.out.println(exp + " -> " + b);
    }

    private static void printIdEq(Object a, Object b, String aName, String bName) {
        System.out.println(aName + " == " + bName + " -> " + (a == b) + "\t"
                + aName + ".equals(" + bName + ") -> " + a.equals(b));
    }

    private static void integerRefs() {
        Integer i = new Integer(2);
        Integer j = new Integer(2);
        print("i >= j", i >= j);
        print("i == j", i == j); // Hmmm...
        print("i == 2", i == 2);
    }

    private static void stringRefs() {
        String s3 = new String("False");
        String s4 = new String("False");
        printIdEq(s3, s4, "s3", "s4");

        String s5 = "True";
        String s6 = "Tr" + "ue";
        printIdEq(s5, s6, "s5", "s6");

        String s7 = "False";
        String sx = "F";
        String s8 = sx + "alse";
        printIdEq(s7, s8, "s7", "s8");
    }
}
```

Figur 2: Klassen **Referenser1**.

[3] Finns gratis här: `http://java.sun.com/docs/books/jls/`

1.3 Referens eller kopia? **

Betrakta klasserna **SimpleSwapper**, **ValueHolder**, **ValueHolderSwapper2** och **ValueHolderSwapper**. Avgör vad utskriften blir då `main` körs i klassen **RefvalMain**.

```
class SimpleSwapper {
    public void swap(Integer x,
            Integer y) {
        Integer temp = x;
        x = y;
        y = temp;
    }
}

public class ValueHolder {
    public Integer i;
    public ValueHolder(Integer i) {
        this.i = i;
    }
}
```

Figur 3: **SimpleSwapper** & **ValueHolder**.

```
class ValueHolderSwapper {
    public void swap(ValueHolder v1,
            ValueHolder v2) {
        Integer tmp = v1.i;
        v1.i = v2.i;
        v2.i = tmp;
    }
}

class ValueHolderSwapper2 {
    public void swap(ValueHolder v1,
            ValueHolder v2) {
        v1 = new ValueHolder(v2.i);
        v2 = new ValueHolder(v1.i);
    }
}
```

Figur 4: **ValueHolderSwapper** & **ValueHolderSwapper2**.

```
public class RefvalMain {
    public static void main(String[] args) {
        SimpleSwapper ss = new SimpleSwapper();
        ValueHolderSwapper vhs = new ValueHolderSwapper();
        ValueHolderSwapper2 vhs2 = new ValueHolderSwapper2();

        int a = 1;
        int b = 2;
        ss.swap(a, b);
        System.out.println("a= " + a + " b= " + b);

        Integer c = new Integer(1);
        Integer d = new Integer(2);
        ss.swap(c, d);
        System.out.println("c= " + c + " d= " + d);

        ValueHolder v1 = new ValueHolder(a);
        ValueHolder v2 = new ValueHolder(b);
        vhs.swap(v1, v2);
        System.out.println("a= " + v1.i + " b= " + v2.i);

        vhs2.swap(v1, v2);
        System.out.println("a= " + v1.i + " b= " + v2.i);
    }
}
```

Figur 5: **RefvalMain**.

2 Implicit typomvandling

Vid körning av `main` i klassen **Casting1**[4] skulle man kunna tro att utskriften blir

```
30587000000000
```

I själva verket blir det något helt annat[5]. Exakt vad är inte så intressant men genom att härleda vilken typ varje deluttryck har kan man förstå varför. Antag att a, b, c är av samma typ, att $a + b$ är av typen `int` och att $a + b + c$ är samma sak som $(a + b) + c$ vilket bör få samma typ som $a + b \ldots$

Vilken precision har (de primitiva) typerna[6] i Java? Vilka intervall kan de representera?

```
public class Casting1 {
    public static void main(String[] args) {
        final long PARSEC = 30587 * 1000000000 * 1000;
        final long M_PER_KM = 1000;

        System.out.println(PARSEC / M_PER_KM);
    }
}
```

Figur 6: Klassen **Casting1**.

3 Snygg design? *

Jan[7] har byggt ett fint spel där man ska döda varelser. Som tur var insåg han tidigt i utvecklingen av spelet att strukturen nedan skulle vara mindre lyckad då man kan tänkas vilja lägga till fler varelser och beteenden.

Diskutera vilka nackdelar som finns med strukturen i klasserna **Gang** (fig. 7) och **Creature** (fig. 8).

Skriv om koden så att man blir av med de problem ni identifierat. Kan abstrakta klasser eller interface vara till någon hjälp? I så fall, vilket borde man välja? Vill eller kan man kombinera dem? Har polymorfism något med saken att göra?

Referenser

[BG05] Joshua Bloch and Neal Gafter. *Java puzzlers: traps, pitfalls, and corner cases.* Addison-Wesley, Boston, Mass., 2005.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition.* The Java Series. Addison-Wesley, Boston, Mass., 2005.

[4] Lånat från [BG05] och något modifierat.
[5] –440487
[6] Se 4.2.1 i `http://java.sun.com/docs/books/jls/third_edition/html/typesValues.html`
[7] Problemet lånat från tenta av Jan Skansholm.

```
public class Gang {
    private List<Creature> members;

    public Gang(int size) { members = new ArrayList<Creature>(size); }

    public void add(Creature m) { members.add(m); }

    public boolean remove(Creature m) { return members.remove(m); }

    public int damageSum() {
        int total = 0;

        for (Creature c : members) {
            if (c != null) {
                int energy = c.getEnergy();
                switch (c.getType()) {
                    case SNAKE: total += 10 * energy; break;
                    case GOBLIN: total += 4 * energy * energy; break;
                    case SPIDER: if (energy > 5) total += 100; break;
                }
            }
        }

        return total;
    }

    public static void main(String[] args) {
        Gang gang = new Gang(3);
        gang.add(new Creature("Sour Serpent", Creature.Type.SNAKE));
        gang.add(new Creature("Greasy Goblin", Creature.Type.GOBLIN));
        gang.add(new Creature("Spicy Spider", Creature.Type.SPIDER));
        System.out.println("Collective damage: " + gang.damageSum());
    }
}
```

Figur 7: Klassen **Gang**.

```
public class Creature {
    public enum Type { SNAKE, GOBLIN, SPIDER };

    private Type type;
    private int energy = 100;
    private String name;

    public Creature(String n, Type t) {
        this.name = n;
        this.type = t;
    }

    public Type getType() { return type; }
    public int getEnergy() { return energy; }
    public void setEnergy(int e) { this.energy = e; }
    public String getName() { return name; }
}
```

Figur 8: Klassen **Creature**.