# Lösningsförslag: Övning vecka 2. *Revision 2017-11-09*

## Uppgift 1

Utskriften blir:

a: Vector, <x = 2, y = 2, z = 3>
b: Vector, <x = 0, y = 2, z = 0>
c: Vector, <x = 2, y = 2, z = 3>
x: 1 y: 2 z: 3

För detaljer se bilaga 1.

Kommentarer:

- a och c är alias, d.v.s. *referensvariablerna refererar till samma objekt*; samma objekt förändras alltså vid anrop av t.ex. metoden add() oavsett om a eller c angavs som referens.
- Det sker skuggning av variabler i metoden silly(). Därför krävs **this**.x för att identifiera instansvariabeln x. (Genom att bara skriva x identifieras *metodparametern* x.)
- ++x inkrementerar variabeln x *innan* exekveringen av satsen som x ingår i, medan x++ inkrementerar x *efter* exekveringen av satsen som x ingår i.
- **this**.z += z har samma semantik som **this**.z = **this**.z + z.

## Uppgift 2

Utskriften blir:

a= 1 b= 2  // har inte bytt plats
c= 1 d= 2  // har inte bytt plats
a= 2 b= 1  // har bytt plats
a= 2 b= 1  // har inte bytt plats

För detaljer se bilaga 2.

Kommentarer: ***Reviderad***

- simpleSwap fungerar inte, oavsett om primitiva typer (för variablerna a och b) eller referenstyper (för variablerna c och d) ges i metodanropet. När metoden anropas läggs kopior av dessa i parametervariablerna. Metoden behandlar sedan bara dessa parametervariabler, och efter att temp kopierats till y terminerar metoden vilket gör att parametervariabeln y kastas bort.
- valueHolderSwap fungerar. Parametervariablerna innehåller visserligen (som alltid) kopior av variabelvärdena som angavs i metodanropet. Men bytet fungerar eftersom metoden sedan arbetar på innehållet (value) i objekten som dessa parametervariabler refererar till. Parametervariablerna vh1 och vh2 är alias till de lokala variablerna v1 respektive v2 från main-metoden.
- valueHolderSwap2 fungerar inte eftersom endast parametervariablerna skrivs över och dessa kastas när metoden terminerar (jämför med simpleSwap).

**Uppgift 3**

Utskriften blir -440487.

Orsaken är att varje enskild heltalsliteral är av typen **int** om inget annat anges explicit. Likaså är resultatet av en multiplikation av två **int**s också en **int**. Om ett värde eller beräkningsresultat inte passar i en **int** "slår det runt". Minvärdet och maxvärdet för en **int** är $-2^{31}$ =-2147483648 respektive $2^{31} - 1$ = 2147483647. Dessa värden nås för övrigt även via konstanterna Integer.MAX_VALUE respektive Integer.MIN_VALUE.

Det gäller alltså att uttrycket 2147483647 + 1 blir -2147483648 och uttrycket -2147483648 - 1 blir 2147483647.

För övriga primitiva typer, följer här ett citat ur "The Java Language specification, Java SE 7 edition", delkapitel 4.2:

> Primitive values do not share state with other primitive values.
>
> The numeric types are the integral types and the floating-point types.
>
> The integral types are byte, short, int, and long, whose values are 8-bit, 16-bit, 32-bit and 64-bit signed two's-complement integers, respectively, and char, whose values are 16-bit unsigned integers representing UTF-16 code units ($3.1).
>
> The floating-point types are float, whose values include the 32-bit IEEE 754 floating-point numbers, and double, whose values include the 64-bit IEEE 754 floating-point numbers.
>
> The boolean type has exactly two values: true and false.
>
> The values of the integral types are integers in the following ranges:
>
> - For byte, from -128 to 127, inclusive
> - For short, from -32768 to 32767, inclusive
> - For int, from -2147483648 to 2147483647, inclusive
> - For long, from -9223372036854775808 to 9223372036854775807, inclusive
> - For char, from 'nu0000' to 'nu' inclusive, that is, from 0 to 65535

För att ge en annan typ till literaler kan ett suffix läggas till värdet. För att få rätt resultat på uträkningen i uppgiften skall literalerna göras till typen **long** t.ex. genom att ett "L" läggs till efter det numeriska värdet:

```
final long PARSEC = 30587L * 1000000000 * 1000;
```

Observera att det endast behöver göras på en av literalerna eftersom den andra ingående operanden (till multiplikationen) då automatiskt kommer att konverteras till **long**.

Ett program som ger det önskade värdet får alltså följande utseende:

```
public class Casting {
    public static void main(String[ ] args) {
        final long PARSEC = 30587L * 1000000000 * 1000; // eller (long) 30587 * 1000000000 * 1000
        final long M_PER_KM = 1000;
        System.out.println(PARSEC / M_PER_KM);
    }
}
```

**Uppgift 4**

a) Utskriften blir

That was a Base, not a strict subclass of Base.: I am a Base!
This was a SubA: I am a SubA!
This was a SubB: I am a SubB!
That was a Base, subclassed by class SubA: I am a SubA!
This was a SubA: I am a SubB!

Kommentarer:

- Variabeln spa har statisk typ Base och dynamisk typ Base.
- Variabeln apa har statisk typ SubA och dynamisk typ SubA.
- Variabeln bepa har statisk typ SubB och dynamisk typ SubB.
- Variabeln apalt har statisk typ Base men dynamisk typ SubA.
- Variabeln bepalt har statisk typ SubA men dynamisk typ SubB.

Generellt gäller att den statiska typen (d.v.s. den deklarerade typen hos referensvariabeln) avgör vilken metod som anropas:

*överlagring*: de statiska typerna hos parametrarna i metodanropet bestämmer vilken metod som anropas.

*hiding* (vilket är när en subklass deklarerat om en statisk metod som finns deklarerad i en superklass): Den statiska typen på referensvariabeln till det objekt som gör anropet bestämmer vilken metod som anropas.

Vid överskuggning (omdefinition) av instansmetoder avgör den dynamiska typen (typen på det objekt som refereras) vilken metod som anropas.

b) Utskrift

That was a Base, subclassed by class SubA: I am a SubA!
That was a Base, subclassed by class SubB: I am a SubB!
That was a Base, not a strict subclass of Base.: I am a Base!
That was a Base, subclassed by class SubA: I am a SubA!
That was a Base, subclassed by class SubB: I am a SubB!
This was a SubA: I am a SubA!
This was a SubA: I am a SubB!

Kommentarer:

Observera att det inuti varje **for**-loop görs en tilldelning till en temporär referensvariabel (o). Det är denna variabels deklarerade typ som utgör den statiska typen i varje anrop till printValue().

Första **for**-loopen har statisk typ Base för alla objekt och dynamiska typer SubA, SubB och Base (för respektive objekt i arrayen). printValue() är överlagrad och går därför på argumentets statiska typ, därmed väljs alltid metoden printValue(Base v). Väl inne i denna metod kontrolleras den dynamiska typen med hjälp av metoden getClass(), och lämplig utskrift görs beroende på om den var Base eller inte. Slutligen anropas toString() som är överskuggad, vilket gör att valet beror å den dynamiska typen.

I andra **for**-loopen har återigen o statisk typ Base vid alla anrop av printValue(),vilket medför samma anropsordning som i föregående punkt.

I tredje **for**-loopen har däremot o statisk typ SubA vilket medför att metoden printValue(SubA v) anropas varje gång.

**Uppgift 5** ***Reviderad***

Det kommer att bli svårt att lägga till nya varelser i spelet och att ändra beteendet hos någon varelse. Strukturen bryter mot *The Open-Closed Principle*.

Problemet beror på en stark koppling mellan klasserna Creature och Gang; funktionaliteten hos en Creature ligger dessutom till viss mån i Gang (dess skadeverkan). Hög koppling brukar leda till att ändringar som i princip borde kunna göras på ett enkelt sätt måste göras på onödigt många ställen. Det blir lätt att missa detaljer och det kan passera obemärkt både vid kompilering och exekvering.

Ponera till exempel att en ny varelse "Wolf" läggs till, men att programmeraren glömmer att införa ett fall för hur mycket skada en varg gör (i metoden damageSum). Detta kommer förmodligen inte att märkas om antalet varelser i ett gäng är stort. Även tillägg till olika varelsers speciella egenskaper blir onödigt komplicerat trots att bara en klass (Creature) behöver förändras. Om till exempel en orm skall kunna ha en viss mängd gift kommer denna variabel (och än värre tillhörande metoder) att existera även för troll och spindlar.

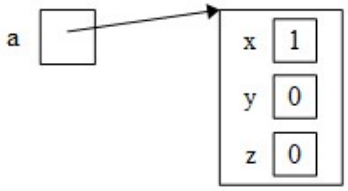
Lösningen på problemet (se javakoden) låter Creature vara en abstrakt klass med olika subklasser för varje enskild typ av varelse. När alla varelser har en gemensam supertyp kan ett gäng (Gang) innehålla en lista av varelser (Creature:s) och anropa gemensamma metoder för denna typ. En varelses speciella funktionalitet kan nu läggas i respektive subklass och göras oberoende av varandra (se metoden damage()).

```
public abstract class Creature {
    private int energy = 100;
    private String name;
    public Creature(String name) {
        this.name = name;
    }
    public int getEnergy() {
        return energy;
    }
    public void setEnergy(int energy) {
        this.energy = energy;
    }
    public String getName() {
        return name;
    }
    public abstract int damage();
}


public class Goblin extends Creature {
    public Goblin(String name) {
        super(name);
    }
    public int damage() {
        return 4 * getEnergy() * getEnergy();
    }
}


public class Snake extends Creature {
    public Snake(String name) {
        super(name);
    }
    public int damage() {
        return 10 * getEnergy();
    }
}
```

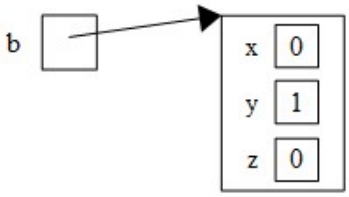
*forts.*

```
public class Spider extends Creature {
   public Spider(String name) {
      super(name);
   }
   public int damage() {
      if (getEnergy() > 5) {
         return 100;
      } else {
         return 0;
      }
   }
}

import java.util.List;
import java.util.ArrayList;
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 int damageSum() {
      int total = 0;
      for (Creature c : members) {
         if (c != null) {
            total += c.damage();
         }
      }
      return total;
   }
}

public class Main {
   public static void main(String[] args) {
      Snake snake = new Snake("Sour Serpent");
      Goblin goblin = new Goblin("Greasy Goblin");
      Spider spider = new Spider("Spicy Spider");
      Gang gang = new Gang(3);
      gang.add(snake);
      gang.add(spider);
      gang.add(goblin);
      System.out.println("Collective damage: " + gang.damageSum());
   }
}
```

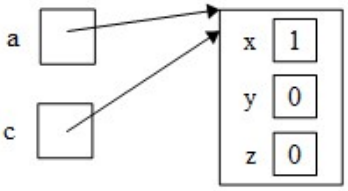
1) `Vector a = new Vector(1, 0, 0);`



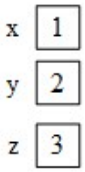

2) `Vector b = new Vector(0, 1, 0);`



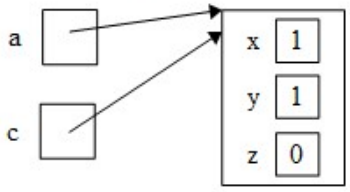

3) `Vector c = a;`



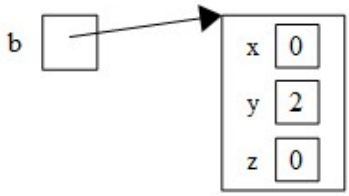

4)
```
int x = 1;
int y = 2;
int z = 3;
```



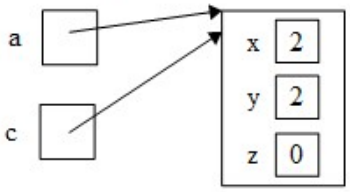

5) `a.add(b);`



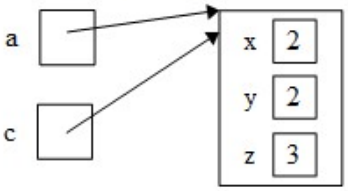

6) `b.add(b)`

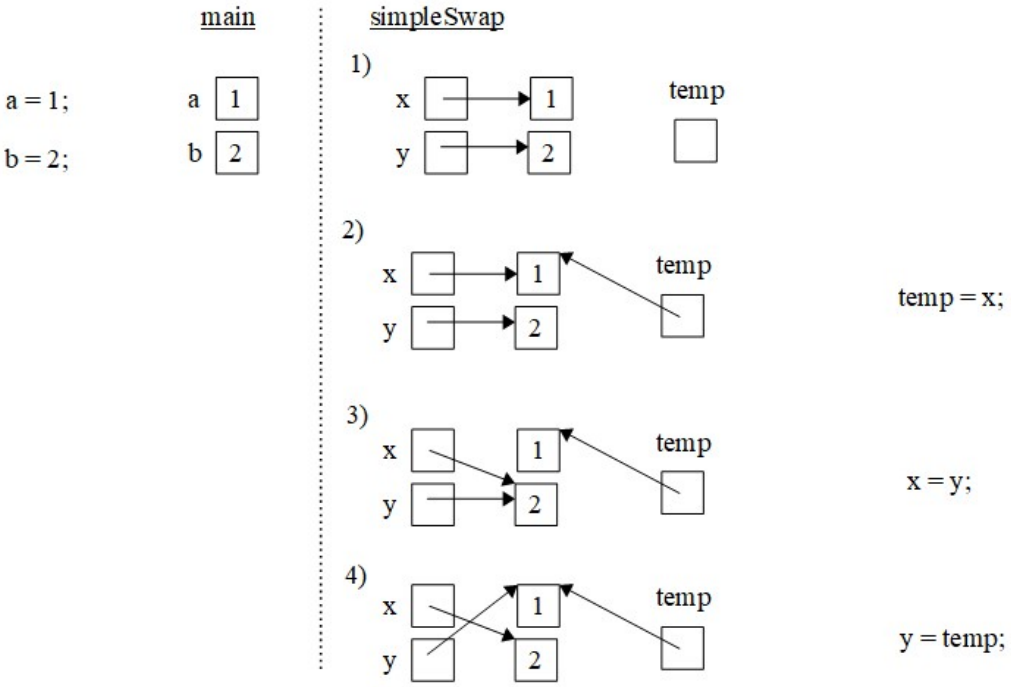


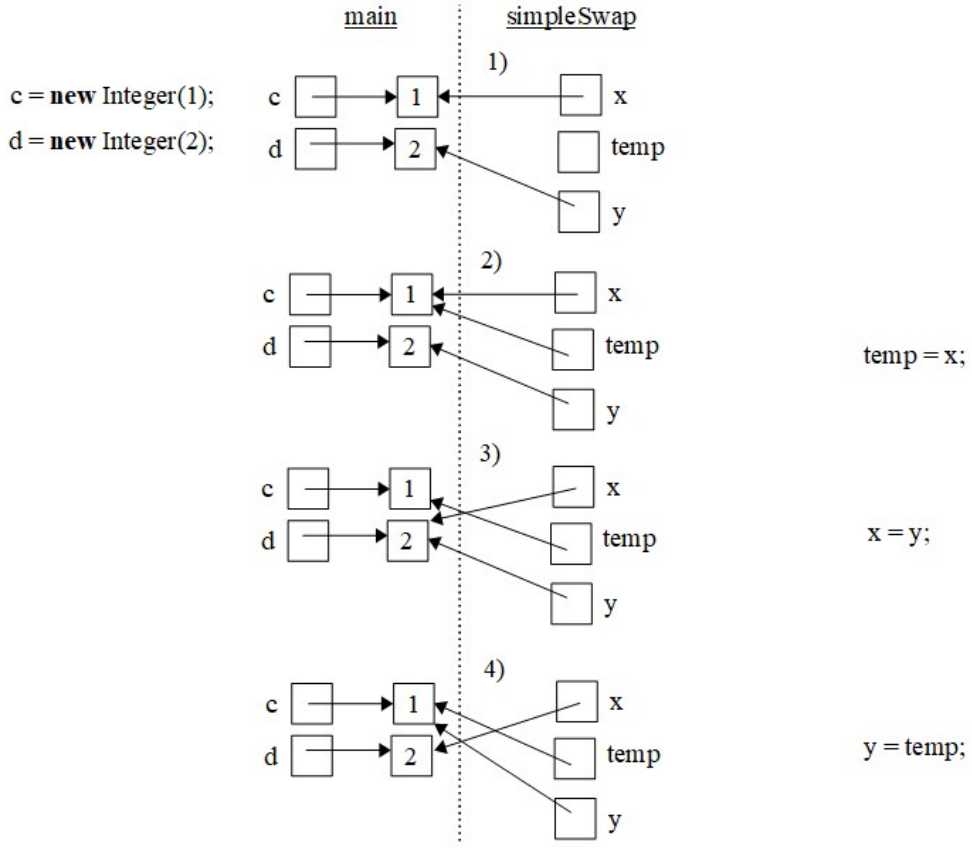

7) `c.add(c);`




8) `c.silly(x, y, z);`

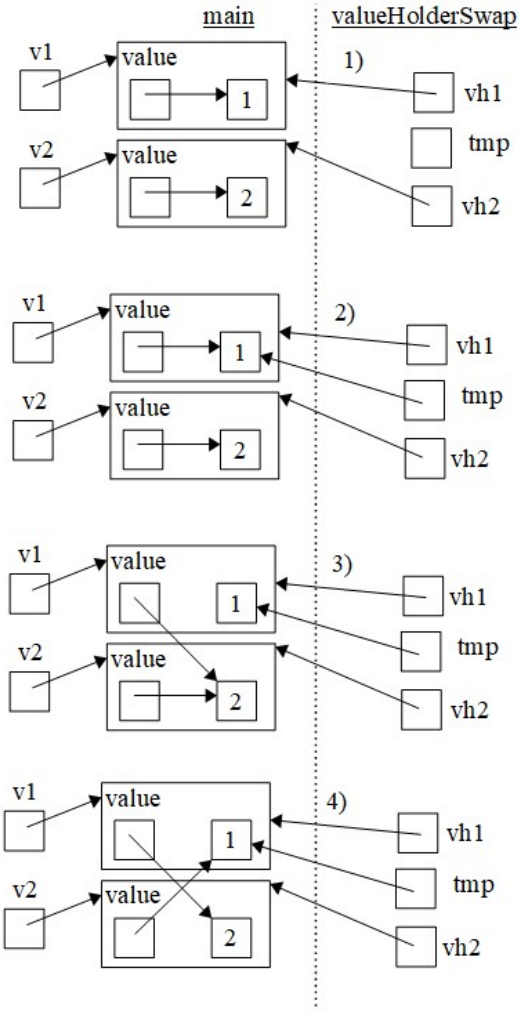


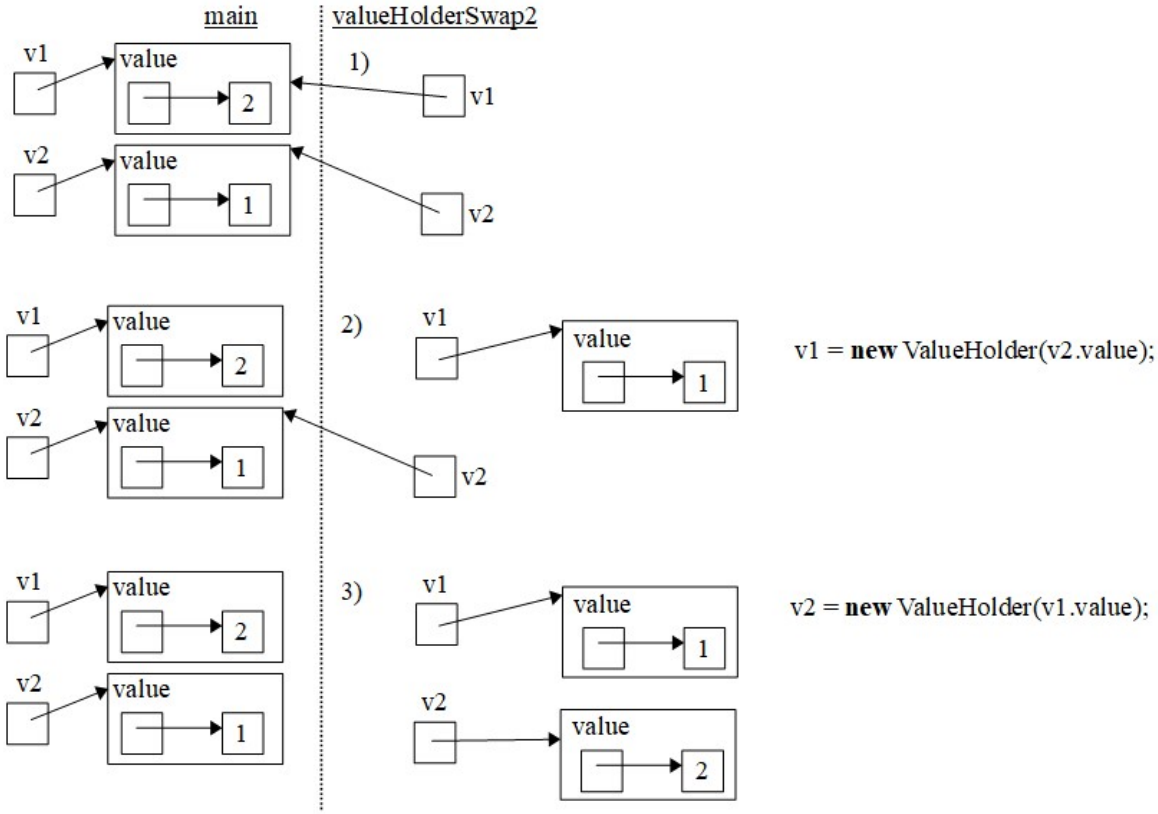

main
simpleSwap
a = 1;
b = 2;
a
b
1)
x
y
temp
2)
temp = x;
3)
x = y;
4)
y = temp;


main
simpleSwap
c = new Integer(1);
d = new Integer(2);
c
d
1)
x
temp
y
2)
temp = x;
3)
x = y;
4)
y = temp;

v1 = **new** ValueHolder(1);

v2 = **new** ValueHolder(2);




main
valueHolderSwap2
v1
value
2
v2
value
1
1)
v1
v2
2)
v1 = new ValueHolder(v2.value);
3)
v2 = new ValueHolder(v1.value);