1 Referenser

1.1 Vektor

Utskrift

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

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 `dumt()`. 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 `x` ingår i, medan `x++` inkrementerar `x` efter körningen av satsen.
- `(this.z += z)` har samma semantik som `(this.z = this.z + z)`.

1.2 Samma kontra lika

Utskrift

```
s3 == s4 -> false s3.equals(s4) -> true
s5 == s6 -> true s5.equals(s6) -> true
s7 == s8 -> false s7.equals(s8) -> true
i >= j -> true
i == j -> false
i == 2 -> true
```

Kommentarer

- Java gör s.k. *interning* av String-objekt d.v.s. under huven bevarar den tidigare skapade *strängkonstanter* i en samling. När programmeraren skapar en ny **String** (observera att **String** är icke-muterbar!), tittar Java först i sin pool om motsvarande textsträng redan existerar där. Om så är fallet, skapas inte något nytt objekt utan en referens till det existerande **String**-objektet returneras. Men som framgår av övningen är Java begränsat i sin härledningsförmåga, därför kan ett nytt objekt i vissa fall ändå skapas trots att ett med samma värde redan existerar. Observera att detta poolningsförfarande inte gäller alla

typer av objekt i Java, speciellt inte de vars klasser du skapar själv! Se exempel 3.10.5-1 i "The Java Language specification, Java SE 7 edition" för ytterligare exempel och resonemang.

- Det existerar omslagstyper (wrapper types) till alla primitiva typer i Java. Dessa är referenstyper som i sitt objekt lagrar samma typ av värde som den motsvarande primitiva typen. "**int**" har en motsvarande klass "**Integer**", "**boolean**" har "**Boolean**" o.s.v. En omslagstyp kan i viss omfattning användas på samma plats som sin motsvarande primitiva typ och tvärtom. När en omslagstyp används på platsen för en primitiv typ kallas konverteringen "unboxing" (det inneboende primitiva värdet packas upp ur lådan) och det omvända fallet kallas "boxing". I fallet med operatorn "==" sker dock inte någon "unboxing", eftersom Java prioriterar jämförelse mellan objektens referenser (vilket är vad som sker för alla sorters referenstyper [inte bara omslagstyper]).

1.3 Swappers

Utskrift

```
a= 1 b= 2
c= 1 d= 2
a= 2 b= 1
a= 2 b= 1
```

Kommentarer

- **SimpleSwapper** 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` avslutas metoden vilket gör att parametervariabeln `y` kastas bort.

- **ValueHolderSwapper** fungerar. Parametervariablerna innehåller visserligen (som alltid) kopior av variabelvärdena som angavs i metodanropet. Men bytet fungerar eftersom metoden sedan arbetar på innehållet (`i`) i objekten som dessa parametervariabler refererar till. Parametervariablerna `v1` och `v2` är alias till de lokala variablerna `v1` respektive `v2` från main-metoden.

- **ValueHolderSwapper2** fungerar inte eftersom endast parametervariablerna skrivs över och dessa kastas när metoden kört färdigt (jämför med **SimpleSwapper**).

2 Implicit typomvandling

Utskrift

```
-440487
```

Kommentarer

- 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** (som ju kan anta negativa värden) är $-2^{31} = -2147483648$ respektive $2^{31} - 1 = 2147483647$[1] vilka alltid bör användas istället för int-literalerna.

- 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 '\u0000' to '\uffff' inclusive, that is, from 0 to 65535

[1] Dessa värden nås för övrigt även via konstanterna `Integer.MAX_VALUE` respektive `Integer.MIN_VALUE`

- För att ge en annan typ till literaler kan ett suffix läggas till värdet. För att få rätt resultat på uträkningen i uppgiften kan literalerna t.ex. göras till typen **long** 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 kommer konverteras till **long**.

3 Snygg design?

Kommentarer

- *Problemet* beror på en stark koppling mellan klasserna **Creature** och **Gang**; funktionaliteten hos en **Creature** ligger dessutom till viss mån i **Gang** (dess skadeverkan). Förändringar som inte genomförs behöver heller inte leda till något fel vid kompilering och dessutom vara svåra att upptäcka under körning. 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 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 ska 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()`).