1 LSP

1.1 Specgen I

Kommentarer

Se javakod.

- Då en klass **Sub** ärver av en klass **Super** och överskuggning förekommer (även för klassmetoder (static)), krävs att returtyperna är utbytbara (return-type-substitutable). Om returtypen till metoden i subklass **Sub** är r_{sub} och returtypen till superklassen **Super** är r_{super}, måste antingen $r_{\text{sub}} = r_{\text{super}}$ eller r_{sub} vara en subtyp till r_{super}. I övningen innebär det att klassen **IntegerClass** kan ärva av **ObjectClass**, men inte tvärtom, eftersom **Integer** är subtyp till **Object**, men inte tvärtom.

1.2 Specgen II

Kommentarer

- En subklass måste (enligt *Liskovs substitutionsprincip* (LSP)) uppträda (allra minst) som dess superklass. I detta fall handlar det om synligheten, en subklass måste tillåta *minst lika stor* synlighet för en metod den överskuggar som dess superklass gör. Eftersom **getValue()** är **public** i klassen **SuperClass**, måste den vara minst **public** i klassen **SubClass**.

- *Överlagring* gäller två eller flera metoder som har samma namn men olika parameterlistor (olika parametertyper och/eller antal parametrar). *Överskuggning* uppstår mellan två eller flera metoder med samma namn *och* samma parameterlistor i klasser med arvsförhållande. Men för att överskuggningen ska vara giltig (programmet kompilerar) måste följande vara uppfyllt för metoderna:

  - De får (naturligtvis) inte befinna sig i samma klass.
  - Subklassmetodens returtyp måste vara utbytbar mot superklassmetodens returtyp (se ovan).
  - Subklassmetodens synlighet måste vara minst lika stor som superklassmetodens.
  - Subklassmetoden får inte kasta några exceptions som inte superklassmetoden gör (men den behöver inte kasta dem! Subklassmetoden får alltså inte göra ”mer fel” än superklassmetoden).

  Som generell minnesregel: Vid överskuggning agerar subklassmetoden i superklassmetodens ställe, och måste därför uppfylla superklassmetodens kontrakt (den ska uppträda minst ”lika bra”).

2 Polymorfism och dynamisk typ

- Generellt gäller att den statiska typen (d.v.s. typen hos referensvariabeln) avgör vilken metod som anropas vid
  - Överlagring (statiska typerna hos parametrarna i metodanropet bestämmer vilken metod som anropas).
  - Överskuggning av klassmetoder (static) (typen på referensvariabeln till det objekt vars metod anropas, bestämmer vilken metod som anropas).

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

2.1 Dynamisk kontra statisk bindning I

Utskrift

```
That was an S, not a strict subclass of S.: I am an S!
This was an A: I am an A!
This was a B: I am a B!
That was an S, subclassed by class binding.A: I am an A!
This was an A: I am a B!
```

Kommentarer

- Variabeln `spa` har statisk typ **S** och dynamisk typ **S**.
- Variabeln `apa` har statisk typ **A** och dynamisk typ **A**.
- Variabeln `bepa` har statisk typ **B** och dynamisk typ **B**.
- Variabeln `apalt` har statisk typ **S** men *dynamisk typ **A***.
- Variabeln `bepalt` har statisk typ **A** men *dynamisk typ **B***.

2.2 Dynamisk kontra statisk bindning II

Utskrift

```
That was an S, subclassed by class binding.A: I am an A!
That was an S, subclassed by class binding.B: I am a B!
That was an S, not a strict subclass of S.: I am an S!
That was an S, subclassed by class binding.A: I am an A!
That was an S, subclassed by class binding.B: I am a B!
This was an A: I am an A!
This was an A: I am a B!
```

Kommentarer

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

    1. Första for-loopen har statisk typ **S** för alla objekt och dynamiska typer **A**, **B** och **S** (för respektive objekt i arrayen). `printValue()` är överlagrad och går därför på argumentets *statiska* typ, därmed väljs alltid `printValue(S v)`. Väl inne i denna metod kontrolleras den dynamiska typen med hjälp av metoden `getClass()`, och lämplig utskrift görs beroende på om den var **S** eller inte. Slutligen anropas `toString()` som är överskuggad, vilket gör att valet beror av den *dynamiska* typen.
    2. I andra for-loopen har återigen `o` statisk typ **S** vid alla anrop av `printValue()`, vilket medför samma anropsordning som i föregående punkt.
    3. I tredje for-loopen har däremot `o` statisk typ **A** vilket medför att `printValue(A v)` anropas alla gånger. Denna metod utför endast en enkel utskrift.

3 Klassinvarianter

Kommentarer

- De två invarianterna

  ```
  @invariant getStart() consistently returns the same value
  after object creation
  @invariant getEnd() consistently returns the same value
  after object creation
  ```

  anger att klassen ska vara icke-muterbar.

- Programmeraren har gjort två fel (se javakod för lösningar):

  – Typerna hos konstruktorns två parametrar är inte primitiva eller icke-muterbara, vilket gör att kopior av objekten måste tas. Annars kan den som skapar ett **Period**-objekt behålla referenser till *start* och *end*. När konstruktorn har kört färdigt och kontrollerat att *start* är mindre eller lika med *end*, kan den utomstående då förändra detta förhållande. Kopior behöver därför tas av objekten innan dessa tilldelas instansvariablerna *start* och *end*.

  – I metoderna **getStart()** och **getEnd()** returneras kopior av instansvariablerna *start* och *end*, vilka refererar till periodens interna objekt. En utomstående får då möjlighet att förändra innehållet i dessa objekt. Istället måste kopior göras av *objekten* och referenser till dessa nya objekt returneras.

4 Equals

4.1 Relationsdrama I

Kommentarer

- Vid anropet till

  ```
  cis.equals(str)
  ```

  där `cis` har typen **CaseInsensitiveString** fungerar jämförelsen både då `str` har typen **CaseInsensitiveString** och då den är av typen **MyString**. Jämförelsen är alltså alltid oberoende av stora och små bokstäver. Men i uttrycket

  ```
  s.equals(str)
  ```

  där `s` är av typen **MyString** utförs istället en jämförelse som är känslig för stora och små bokstäver. Detta medför att vi har ett brott mot symmetrin. Ta t.ex. fallet då en **MyString** `"Bo"` jämförs med en **CaseInsensitiveString** som innehåller `"bo"`. Följande kommer inträffa:

```
"bo" equals "Bo" --> true
"Bo" equals "bo" --> false
```

- Det finns ingen annan lösning på problemet än att endast utföra jämförelser mellan objekt av samma typ (se javakod).

4.2 Relationsdrama II

Kommentarer

- Här har man infört en värdekomponent genom instansvariabeln *name*. För att åtgärda symmetriproblemet testas i **NamedPoint**s **equals()** om det objekt (*o*) som jämförs med är av typen **Point**, i sådana fall anropas dennes **equals()**. Men det blir ändå problem, i detta fall med transitiviteten. Här följer ett motexempel som visar ett sådant brott. En **Point** representeras som en tvåtupel med sina x- och y-koordinater, (x,y), och en **NamedPoint** som en tretupel med x- och y-koordinater följt av namnet, (x,y,namn).

  ```
  p1 = (0,0,"centrum")
  p2 = (0,0)
  p3 = (0,0,"origo")

  p1 equals p2 --> true
  p2 equals p3 --> true
  p1 equals p3 --> false
  ```

  De två första jämförelserna var ok, då testades endast koordinaterna eftersom en **Point** var del av jämförelsen. Men det hela fallerar i tredje jämförelsen när namnen tas med.

- Återigen finns det ingen fullständig lösning på problemet. Även i detta fall (då en värdekomponent införts) är enda sättet att endast utföra jämförelser mellan objekt av samma typ[1] (se javakod).

- Observera att **instanceof**-operatorn returnerar sant för subtyper och därmed kommer jämförelser med subtyper att tillåtas. Dessa kan innehålla införda värdekomponenter, vilket gör att equals-kontraktet bryts. För att undvika detta kan metoden **getClass()** användas istället för **instanceof**. Men att inte tillåta jämförelser med subklasser medför ett brott mot LSP, vilket kan ställa till det, t.ex. om objekt av subtyper ska kunna lagras i samlingar. Det måste övervägas om equals-kontraktet eller LSP ska uppfyllas.

[1] För att slippa duplicera kod kan ett objekt av typen **Point** inneslutas som en komponent i ett **NamedPoint**-objekt.