Polymorfism, skuggning och överlagring av metoder

Tema: Du ska fördjupa dina kunskaper om polymorfism, skuggning och överlagring av metoder.

Begreppen typ, subtyp och supertyp

När vi deklarerar variabler anger vi vilken *typ* de har. Typen anger vilka värden som kan tilldelas variabeln. För referensvariabler används klassnamn (eller interfacenamn) för att ange typen:

```
Person p;   // Person är en klass
Set set;    // Set är ett interface
```

Begreppen *supertyp* och *subtyp* är viktiga att känna till. Supertyper till en viss klass `C` är alla de klasser från vilka `C` ärver och alla de interface klassen implementerar. Supertyper till ett interface `I` är alla de interface från vilka `I` ärver. En klass `C` eller ett interface `I` är subtyp till alla sina supertyper.

Varje objekt tillhör någon klass, den klass som används när objektet skapades. Om en referensvariabel deklareras ha en viss typ (klass eller interface) så får den referera till objekt av denna typ eller dess subtyper. Om ett interfacenamn har använts i deklarationen innebär detta speciellt att referensvariablen får referera till objekt av någon klass som implementerar detta interface. För variablerna `p` och `s`, deklarerade ovan, gäller därför att `p` kan referera till objekt av klasserna `Person`, `Student` och `FacultyMember`. Referensvariablen `set` kan referera till objekt av t ex klasserna `TreeSet` och `HashSet` som finns i `java.util` och som båda implementerar interfacet `Set`. Det är alltså tillåtet att skriva:

```
set = new TreeSet();
set = new HashSet();
```

U 1. Rita ett klassdiagram i UML och ange alla subtyp–supertyp-relationer som gäller när följande deklarationer gjorts:

```
class A {..}
interface I {..}
class B extends A implements I {..}
interface J extends I {..}
class C extends B implements J {..}
```

U 2. Antag att följande deklarationer gjorts:

```
interface IA {..};
interface IB extends IA {..}
class C implements IA {..}
```

Rita ett klassdiagram i UML. Vilka av tilldelningssatserna i följande kod är då korrekta?

```
IA a = new C();
IB b = new C();
a = b;
b = a;
C c = new C();
c = a;
c = b;
```

Metodsignaturer och metodanrop

Antag att vi har en metod, deklarerad enligt följande mönster:

```
public void m(Type1 param1, Type2 param2,...);
```

där $\texttt{Type}_i$ anger en klass eller ett interface. Metodens *signatur* är dess namn tillsammans med antalet parametrar och deras typer. Två metoder har samma signatur om de har samma namn, samma antal parametrar och om den i:e parametern i den första metoden har samma typ som den i:e parametern i den andra metoden för alla i.

För att ett anrop, $\texttt{m(act}_1\texttt{, act}_2\texttt{,..);}$, ska vara korrekt krävs att varje aktuell parameter $\texttt{act}_i$ har en sådan deklarerad typ att tilldelningssatsen $\texttt{param}_i\texttt{ = act}_i$ är korrekt. Detta innebär att den *deklarerade* typen för $\texttt{act}_i$ måste vara $\texttt{Type}_i$ eller en subtyp till denna typ.

U 3. Ange, för var och en av metodanropen i det följande om de är korrekta eller felaktiga:

```
public class C {
    public void m(Person p, Student s) {...}
}
..
Person p1 = new Person(...);
Student  s1 = new Student(...);
C c = new C();
c.m(p1, s1);
c.m(p1, p1);
c.m(s1, p1);
c.m(s1, s1);
p1 = new Student(...);
c.m(p1, p1);
```

Metodanrop, skuggning och överlagring

Vi kan inte deklarera två metoder med samma signatur i en klass. Däremot kan det finnas metoder med samma signatur i en klass och i någon av dess superklasser eller subklasser. Det är detta som kallas att skugga metoder (eng: override). Om det i en klass `C` och i en subklass `D` till `C` finns metoder med samma signatur så säger vi att metoden i `D` skuggar (eng:overrides) metoden i klassen `C`.

Metoder som har samma namn men olika signatur (t. ex olika antal parametrar eller olika typer på sina parametrar) sägs *överlagra* (eng: overload) varandra. Det är tillåtet att ha flera överlagrade metoder i samma klass. Överlagring inträffar också om en klass har en metod med samma namn men olika signatur som någon metod i en super- eller subklass.

Då man gör ett metodanrop `x.m(a,b)` kommer någon metod med namnet `m` och med formella parametrar av en sådan typ att anropet är korrekt med hänsyn till typerna hos de aktuella parametrarna `a` och `b` att exekveras. I allmänhet är det inte svårt att avgöra vilken metod det blir. Ofta har vi bara en metod med rätt namn och rätt antal parametrar. Vid vissa typer av överlagring kan dock tveksamhet uppstå. Betrakta följande exempel:

```
class BaseClass {
    public void foo(Person p) {...}
    public void foo(Student s) {...}
}
...
BaseClass bc = new BaseClass();
Person p = new Student(...);
bc.foo(p);
```

Vilken av `foo`-metoderna kommer att exekveras? De deklarerade typerna för `bc` respektive `p` används av kompilatorn för att avgöra detta. `bc` är av typen `BaseClass` och `p` är av typen `Person`. Kompilatorn undersöker därför om klassen `BaseClass` har en metod som heter `foo` och som har en formell parameter av en sådan typ att den kan anropas med en aktuell parameter av typen `Person`. Det finns bara en sådan, nämligen den första foo-metoden.

`bc` skulle under exekvering kunna referera till ett objekt av en subklass till `BaseClass` om sådan finns. Exempel:

```
class DerivedClass extends BaseClass {
    public void foo(Person p) {...}
}
...
BaseClass bc = new DerivedClass();
Person p = new Student(...);
bc.foo(p);
```

Nu blir det metoden foo i klassen `DerivedClass` som exekveras. Kompilatorn gör samma beslut som i föregående exempel och fastlägger *signaturen* på metoden till att vara namnet `foo` och en parameter av typen `Person`. Under exekvering kommer klassen för det objekt till vilket `bc` då refererar att utgöra den virtuella maskinens startpunkt för sökandet efter metod med denna signatur. Finns ingen sådan metod där fortsätter sökandet i superklassen etc. I detta fall startar alltså sökandet i `DerivedClass` där det finns en metod med rätt signatur.

Ett ytterligare exempel där överlagrade metoder är definierade i olika klasser i en klasshierarki:

```
class BaseClass {
    public void foo(Person p) {...}
}
class DerivedClass extends BaseClass {
    public void foo(Student s) {...}
}
..
DerivedClass dc = new DerivedClass();
Person p = new Student(...);
dc.foo(p);
```

`dc` har typen `DerivedClass` som har två överlagrade metoder med namnet `foo`. Men bara en av dessa, den som är definierad i superklassen `BaseClass` passar för anropet eftersom dess parameter är av typen `Person`. Metoden i `DerivedClass` har ju formell parameter av typen `Student` och denna kan inte tilldelas en aktuell parameter av typen `Person`. Det blir alltså metoden `foo` i `BaseClass` som exekveras.

Sammanfattning

Nedan sammanfattas vad som sker vid kompilering respektive exekvering av ett metodanrop `x.m(a,b);`

1. Den deklarerade typen för `x` används av kompilatorn för att bestämma i vilken klass sökandet efter metoder som har rätt signatur för anropet ska starta. Kalla denna klass `C1`.

2. I `C1` och dess eventuella superklasser lokaliseras alla metoder som matchar metodanropet, dvs. har rätt namn och har sådana typer på sina formella parametrar att de kan tilldelas variabler av respektive aktuell parameters deklarerade typ. Om ingen sådan metod finns, genereras ett kompileringsfel. Om det bara finns en sådan metod, fortsätt med steg 3. I komplicerade fall med många överlagrade metoder kan det finnas flera som passar in. I så

fall försöker kompilatorn få fram den metod som är "mest specifik" enligt vissa regler, som vi inte går in på här. Slutar denna process med att det finns mer än en möjlig metod får vi ett kompileringsfel (ambiguous method call).

3. En metod är nu utvald och det är en metod av dess exakta signatur som kommer att exekveras. Vilken det blir beror på klassen för det objekt `x` refererar till under exekvering. Kalla denna klass `C2`. Den virtuella maskinen startar sökandet efter metod att exekvera i `C2` och fortsätter eventuellt i superklasser tills man hittar en med rätt signatur. I normala fall är det säkert att vi har en metod med rätt signatur någonstans i denna kedja, annars hade kompileringsfel genererats i steg 2 ovan. Bara om vi ändrat någonting som har med inblandade metoder att göra och glömt kompilera om vissa klasser kan sökandet misslyckas. I så fall får man ett exekveringsfel.

Anmärkning till punkt 1: Metodanropet kan göras inifrån en annan metod och x kan vara en av de formella parametrarna till denna enligt följande:

```
public void p(SomeClass x) {
    x.m(a,b);
    ...
}
```

Den deklarerade typen för `x` är då `SomeClass`.

Anmärkningar till punkt 2: Det räcker här inte med att en metod har rätt namn och rätt typ på sina parametrar. Den måste också vara möjlig att anropa med hänsyn till sin skyddsnivå (`private, public` etc.). Den som vill veta mera om hur det går till att bestämma mest specifika metod när man har flera kandidater kan gå till avsnitt 15.12 i Javas språkdefinition som finns på adressen `http://docs.oracle.com/javase/specs/`

Anmärkningar till punkt 3: Att man först under exekvering bestämmer exakt vilken metod som exekveras brukar kallas *dynamisk bindning* (eng: dynamic binding eller late binding). Ibland kan man dock redan under kompileringen bestämma vilken metod som ska exekveras. Metoder som är deklarerade `final` får inte omdefinieras i subklasser. Om steg 2 slutar med att man hittat en metod och denna är `final` så vet man alltså att det måste bli denna som ska exekveras.

U 4. Vad ingår i en metods signatur?

U 5. Vad menas med överskuggning? Ge exempel.

U 6. Vad menas med överlagring? Ge exempel.