Polymorfi

Sidst ændret: 07/19/2018 14:11:41
Opgaver

I dette kapitel vil vi se nærmere på mulighederne for at mindske koblingen mellem objekter, så de har et mindre konkret kendskab til hinanden. De første afsnit behandler den tekniske side af sagen, mens vi senere i kapitlet vender os mod spørgsmålet: Hvorfor?

Flere-former

Poly-morfi betyder *flere-former*.

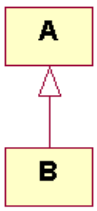
Hvad er det, der kan have flere "former"? Det er objekter!

1. Abstrakt kendskab til objekter

Ved vi hvad vi har?

Når vi har en reference til et objekt, ved vi hvad der er i den anden ende af referencen - hvad den refererer til. Hvis referencen er af typen **A**, ved vi at den refererer til en instans af klassen **A**; hvis den da ikke er **null**. Sådan har det været hidtil, men det er ikke nødvendigvis sådan.

Hvis vi f.eks. har flg. klasser:



Figur 1:
B *nedarver fra* **A**

Og har en reference erklæret ved:

```
A ref;
```

Så kan denne reference ikke alene referere til instanser af **A**. Den kan også referere til instanser af **B**.

F.eks.:

Source 1:
A-*reference til en instans af* **B**

```
public class A {
  public String toString() {
    return "[instans af A]";
  }
}
```

```
public class B extends A {
  public String toString() {
    return "[instans af B]";
  }
}
```

```
public class Main {

  public static void main( String[] argv ) {
    A ref;

    ref = new B();
    System.out.println( ref );
```

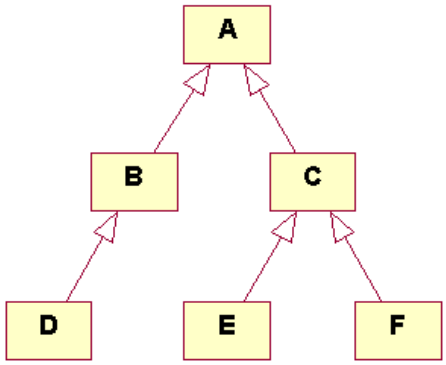
```
  }
}
```

```
[instans af B]
```

Derimod vil en reference af klassen **B** kun kunne referere til en instans af **B**.

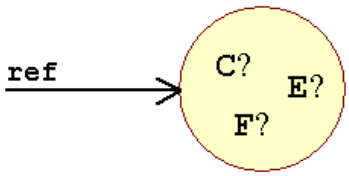
Det skyldes at en reference af en klasse **X**, *kun* kan referere til instanser af klassen **X** eller en subklasse til **X**. Bemærk, at dette gælder *alle* subklasser af **X**, ikke kun dem der *direkte* nedarver fra **X**.

Hvis vi f.eks. har følgende klasse-hierarki:



Figur 2:
Nedarvnings-hierarki med seks klasser

Så kan en reference af klassen **C**, kun referere til en instans af **C**, **E** eller **F**. Lad os antage at vi har en reference **ref** af klassen **C**, der referer til et objekt.



Figur 3:
Ukendt instans

Abstrakt

Da vi med en reference af klassen `C`, ikke kan vide præcist hvilken klasse vores objekt er en instans af, har vi fået et mere **abstrakt** forhold til instansen.

Konkret

Vi ved at den har de egenskaber vi kan forvente af en instans af `C`, men om den har de egenskaber der suppleres med i `E` eller `F`, ved vi ikke. Det vil kræve et mere **konkret** viden om hvilken klasse objektet er en instans af.

Polymorfi

Disse to egenskaber: **abstrakt** og **konkret**, er centrale i vores beskrivelse af polymorfi. Polymorfien opstår ved, at vi har et abstrakt forhold til instansen, idet vi ikke kender dens konkrete klasse. De metoder der er erklæret i `C`, kan antage forskellige "former" alt efter hvilken konkret klasse objektet er en instans af. Forskellige subklasser kan implementere de samme metoder på forskellig måde, og vi kan ikke vide præcist hvad der sker når vi kalder dem.

Eftersom vi ikke ved om referencen `ref` refererer til en instans af `C`, `E` eller `F`, kan vi kun sende requests til den, som er implementeret i alle tre klasser, nemlig dem der er erklæret i `C`.

2. Casting

Hvis vi vender tilbage til vores lidt enklere eksempel med de to klasse `A` og `B` (fra Source 1), og udbygger `B` med en ekstra metode `kvadrat`, kan vi få følgende problem:

Source 2:
Forsøg på at kalde **kvadrat** *via reference af klassen* **A**

```
public class A {
  public String toString() {
    return "[instans af A]";
  }
}
```

```
public class B extends A {
  public int kvadrat( int x ) {
    return x*x;
  }

  public String toString() {
```

```
    return "[instans af B]";
  }
}
```

```
public class Main {

  public static void main( String[] argv ) {
    A ref = new B();

    int femIAnden = ref.kvadrat( 5 );
    System.out.println( femIAnden );
  }
}
```

```
--------------------------- Compiler Output ---------------------------
test.java:21: Method kvadrat(int) not found in class A.
    int femIAnden = Aref.kvadrat( 5 );
                                  ^
1 error
```

For abstrakt et kendskab

Her laver vi en instans af **B**, men bruger en reference af klassen **A** til at holde fast i den. Det betyder at vi via referencen **ref** kun kan kalde metoder der er erklæret i **A**. Dermed er det ikke muligt at kalde **kvadrat**, selvom objektet faktisk har denne metode. Vi har ganske enkelt et for abstrakt kendskab til instansen, til at vi kan foretage et sådant kald. Kompileren ser fejlen og vil ikke oversætte metodekaldet.

Vi ved det

Hvad gør vi, hvis vi som programmører godt ved der er tale om en instans af **B** og at vi af praktiske grunde har fat i objektet med en **A**-reference? Hvordan kan vi overbevise kompileren om at det er i orden at kalde **kvadrat**-metoden. Hvordan kan vi gøre kendskabet mere konkret? Det kan vi med **casting**!

Berolige kompileren

Her er casting ikke det samme som vi så i kapitlet om typer. Dér var casting en konvertering mellem primitive typer, her er det kun et spørgsmål om at berolige kompileren, så den oversætter metodekaldet. Hvis det under programudførelsen skulle vise sig ikke at holde stik, kommer der et runtime error.

Selvom vores eksempel med **A** og **B** er meget simpelt, kan vi bruge det til at illustrere teknikken med casting af referencer. F.eks.:

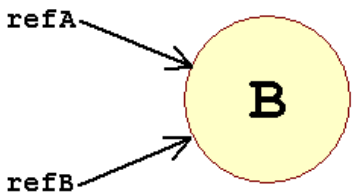
Source 3:
Casting fra **A** *til* **B**

```
public class Main {

  public static void main( String[] argv ) {
    A refA = new B();

    B refB = (B) refA;
    int femIAnden = refB.kvadrat( 5 );

    System.out.println( femIAnden );
  }
}
```

```
25
```

Her får vi to referencer til samme objekt:

Figur 4:
To forskellige slags referencer til instans af ***B***



Hvis vi sender en request via **refA** kan vi kun bruge de metoder som er erklæret i **A**, men bruger vi i stedet **refB** kan vi kalde de metoder der er erklæret i **B** (eller nedarvet til **B**).

Lad os prøve at snyde - at caste i en situation hvor det ikke passer:

Source 4:
Brudte løfter

```
public class Main {

  public static void main( String[] argv ) {
    A refA = new A();

    B refB = (B) refA;
    int femIAnden = refB.kvadrat( 5 );
```

```
    System.out.println( femIAnden );
  }
}
```

```
java.lang.ClassCastException: A
      at test.main(test.java:6)
```

Vi får en fejl i linie 6, under programudførelsen. Fejlen skyldes at vi ikke kan foretage den casting vi havde lovet kompileren, da vi placerede **`(B)`** foran **`refA`** i assignment.

instanceof

Kan vi sikre os mod, at denne fejl opstår? Er der en måde at checke hvilken klasse et objekt er en instans af? Ja, med operatoren **`instanceof`**:

Source 5:
Brug af ***instanceof***

```
public class Main {

  public static void main( String[] argv ) {
    A refA = new A();

    if ( refA instanceof B ) {
      B refB = (B) refA;

      int femIAnden = refB.kvadrat( 5 );
      System.out.println( femIAnden );
    } else
      System.out.println( "Det var ikke en instans af B" );
  }
}
```

```
Det var ikke en instans af B
```

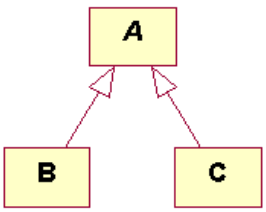
Klassen eller subklasser

Her er betingelsen ikke opfyldt - **refA** refererer ikke til en instans af **B**. Operatoren opererer på to operander; hvoraf den første er en reference og den anden er en klasse. Udtrykket evalueres til sandt, ikke alene hvis referencen refererer til en instans af klassen, men også hvis den pågældende klasse er en instans af en subklasse til klassen. Hvis **B** i dette tilfælde havde haft subklassen **D** fra figur 2., ville udtrykket også være evalueret til sandt, hvis **refA** havde refereret til en instans af **D**.

3. Abstrakte klasser

Betragt følgende klasse-hierarki:

Figur 5:
Abstrakt super-klasse **A**



A er praktisk for B og C

Lad os antage at vi *ikke* har lavet **A** med henblik på instantiering. **A** er lavet for at **B** og **C** kan arve en del af deres interface, samt den del af implementationen som de har til fælles. Mao., det at **A** eksisterer er primært af praktiske grunde, og vi ønsker aldrig at lave instanser af den.

Skrå skrift

Klasser som vi ikke ønsker der skal kunne laves instanser af, kaldes **abstrakte klasser**. I klassediagrammer skriver vi klassenavnet med *skrå skrift* (*italic*) for at anføre denne egenskab ved klassen.

Denne beslutning, om at en klasse aldrig skal bruges til instantiering, er sprogligt understøttet i Java:

Source 6:
Abstrakt super-klasse

```
public abstract class A {
  ...
}
```

```
public class B extends A {
  ...
}
```

```
public class C extends A {
  ...
}
```

Kompileren håndhæver det

Her har vi placeret ordet **abstract** foran **class**, og kompileren vil nu håndhæve at der *ikke* kan laves instanser af **A**. Vi kan stadig have *referencer* af klassen **A**, men de vil kun kunne refererer til instanser af **B** og **C**, da dé er de eneste subklasser af **A**, der tillader instantiering. F.eks:

Source 7: *Forsøg på instantiering af abstrakt klasse*

```
public class Main {

  public static void main( String[] argv ) {
    A ref1 = new C();
    A ref2 = new B();
    A ref3 = new A();
  }
}
```

```
--------------------------- Compiler Output ---------------------------
test.java:6: class A is an abstract class. It can't be instantiated.
    A ref3 = new A();
             ^
1 error
```

Man ser her, at instantieringen af **A** ikke godtages af kompileren, men at de andre accepteres.

3.1. **final** klasser

Mens vi er ved ord, man kan placeres foran **class**, bør vi også nævne **final**.

Ingen nedarvning

final understøtter en anden beslutning om en klasse. Den, at man ikke vil acceptere, at der nedarves fra den. Umiddelbart en lidt usædvanlig beslutning, og den er da også sjældent brugt[1].

F.eks:

Source 8: *Forsøg på nedarvning fra* **final** *klasse*

```
public final class A {
  ...
}
```

```
public class B extends A {
  ...
}
```

```
--------------------------- Compiler Output ---------------------------
test.java:...: Can't subclass final classes: class A
public class B extends A {
                       ^
1 error
```

Her ser man, at kompileren ikke accepterer, at der nedarves fra **A**, da den er **final**.

4. Interfaces

4.1 Abstrakte metoder

Manglende implementation

Man kan gå skridtet endnu videre fra en abstrakt klasse til en klasse der helt mangler implementation. I første omgang kunne man forstille sig følgende abstrakte super-klasse:

Source 9: *Forsøg på ikke at implementere metode i super-klasse*

```
public abstract class A {
  public int kvadrat( int x ) {
  }
}
```

```
public class B extends A {
  public int kvadrat( int x ) {
    return x*x;
```

```
  }
}
```

```
--------------------------- Compiler Output ---------------------------
test.java:2: Return required at end of int kvadrat(int).
  public int kvadrat( int x ) {
             ^
1 error
```

Kræver returnering

Men, som man ser, får vi en fejl. Hvis metoden skal returnere noget, vil Java ikke acceptere, at vi lader en metode stå uden indhold, for senere i subklassen at implementere den.

Ikke implementere metode

Hvis man i en abstrakt klasse vil undlade at implementere visse af metoderne skal man erklære dem **abstract** og ikke anføre nogen metode-krop.

Source 10:
*Brug af **abstract** metode*

```
public abstract class A {
  public abstract int kvadrat( int x );
}
```

```
public class B extends A {
  public int kvadrat( int x ) {
    return x*x;
  }
}
```

Som nævnt kan man kun gøre dette i en abstrakt klasse.

Blanding

Man kan på denne måde lave en blanding af metoder, hvor man tilbyder en default-implementation (de metoder der ikke er **abstract**) og for andre *kræver* at subklasserne *skal* lave deres egen implementation (de metoder der er **abstract**).

4.2 **interface**

Kun abstrakte metoder

Hvis man, som i vores eksempel, *kun* har abstrakte metoder, kan man i stedet erklære klassen som et **interface**:

Source 11:
Brug af **interface**

```
public interface A {
  public int kvadrat( int x );
}
```

```
public class B implements A {
  public int kvadrat( int x ) {
    return x*x;
  }
}
```

implements

Her er de to forekomster af **abstract** fjernet. I stedet står der **interface**, som angiver, at der er tale om en klasse uden nogen form for implementation. Ved nedarvningen skriver man ikke længere **extends**, men **implements**.

Multipel nedarvning

Til forskel fra klasser, hvoraf man kun kan nedarve fra én, kan man implementere så mange interfaces det skal være. På den måde bliver en begrænset form for multipel nedarvning mulig, hvor man kan nedarve fra én klasse og implementere flere interfaces. F.eks.:

Source 12:
Implementation af flere interfaces

```
public class A {
  ...
}
```

```
public interface B {
  ...
}
```

```
public interface C {
  ...
}
```

```
public class D extends A implements B, C {
  ...
}
```

Her har vi en klasse **A**, to interfaces **B** og **C**, samt en klasse der arver fra **A** og implementerer **B** og **C**.

Lutter pligter

Set fra en klasses synspunkt, er det lutter pligter der følger af at skulle implementere et interface. Den påtager sig at implementere de metoder som er nævnt i interfacet, og dén får ikke noget til gengæld.

Når vi arbejder med instanser af **D**, får vi til gengæld flere muligheder, f.eks:

Source 13:
Tre slags referencer til samme objekt

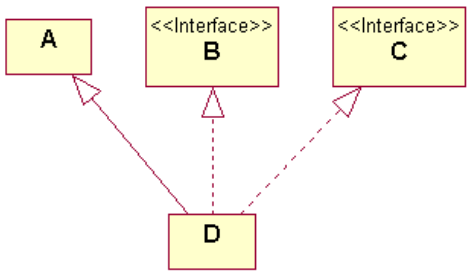
```
public class Main {

  public static void main( String[] argv ) {
    A refA = new D();
    B refB = new D();
    C refC = new D();
  }
}
```

Referencer af interfaces

Det er muligt at anvende referencer af interfaces til et objekt. Forskellen på de tre referencer ligger i hvilke requests vi kan sende. F.eks. kan vi via **refB** kun sende requests som er erklæret i interfacet **B**.

Når vi anfører et interface i et klasse-diagram skriver vi navnet "interface" mellem et dobbelt sæt vinklede paranteser før klassenavnet. Selvom et interface er en ultimativt abtrakt klasse, skrives navnet *ikke* med skrå skrift (italic). Følgende nedarvnings-hierarki illustrerer dette for vores eksempel.

Figur 6:
Nedarvning fra interfaces



5. `class Object`

`Object` er en speciel klasse i Java. Den er den ultimative super-klasse - alle super-klassers super-klasse.

`Object` befinder sig derfor øverst i ethvert klasse-hierarki, og en klasse der ikke nedarver fra nogen anden klasse, vil altid nedarve direkte fra **`Object`**.

Hvis vi f.eks. har følgende klasse:

```
public class A {
  ...
}
```

Vil den blive "læst" af compileren som:

```
public class A extends Object {
  ...
}
```

Dvs. uanset om vi vil det eller ej, så nedarver vi, direkte eller indirekte, fra **`Object`**. Det kan vi bruge til at lave generelle anvendelser af objekter. F.eks. kan vil lave en metode der kan tage et hvilket som helst objekt som parameter:

```
public ... f( Object obj ) {
  ...
}
```

Her kan metoden **f** tage en reference til et hvilket som helst objekt som parameter, idet en reference af klassen **Object** kan referere til en instans af **Object** eller en af dens subklasser - dvs. *alt*.

Hvad glæde har man af det? Først og fremmest kan man håndtere objekter generelt, uanset hvilke klasser de måtte være instanser af, men man kan også sende requests til dem.

toString

toString er den mest generelle af de metoder, der er erklæret i **Object**. Det er derfor den er understøttet af bla. **println** og klistreplus.

Man kan kalde **toString** på ethvert objekt og få en mere eller mindre beskrivende tekststreng returneret.

Implementationen i **Object** returnerer en tekststreng på følgende form:

<klassenavn>@<virtuel addresse>

Den virtuelle adresse[2] er en angivelse af hvor i lagret objektet befinder sig - normalt en oplysning af ringe værdi.

F.eks.:

```
public class Main {

  public static void main( String[] argv ) {
    Object obj = new Object();

    System.out.println( obj );
  }
}
```

```
java.lang.Object@713669d2
```

Fuld pakkesti

Man ser at klassenavnet angives med hvilken package klassen er placeret i - såkaldt: **fuld pakkesti**.

equals

equals er en anden metode, der er erklæret i **Object**. Her tager den et **Obejct** som parameter og returnerer om de to objekter er det samme objekt. Denne implementation er naturligvis af ringe værdi, og meningen er da også, at den skal overrides eller overloades, hvis den skal anvendes i praksis.

6. Abstraktion og kobling

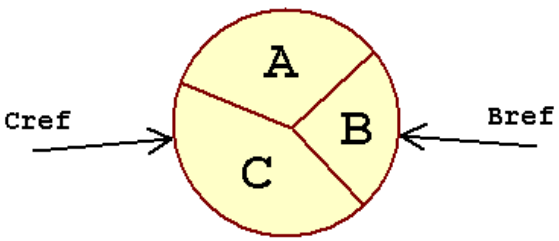
Hvorfor abstrakt

Hvorfor skulle vi ønske at have en reference der er abstrakt i forhold til det objekt vi har fat i? Hvorfor ikke have en reference der passer præcist til objektet, og dermed alle muligheder for at anvende dets metoder?

Fordi det kobler os stærkere til objektet. Vi vil kun være i stand til at holde objekter af netop den klasse, som passer til referencen, og vi vil kun kunne kommunikere med instanser af én bestemt klasse, selvom instanser af andre klasser (subklasser) også ville kunne forstå vores requests.

Kort sagt, vi bliver mindre fleksible.

Betragt følgende objekt:



Figur 7:
Opdelt interface

Objektet er en instans af **D**, fra **Source 12**. Vi har indtegnet de tre dele af dets interface som det har fra de tre forskellige klasser **A**, **B** og **C**; hvor **B** og **C** er **interface**'s. De to referencer kunne være erklæret og initialiseret ved:

```
B Bref = new D();
C Cref = (C) Bref;
```

Som det ses, ved vi her, at **`Bref`** refererer til et objekt der implementerer interfacet **`C`**, hvorfor vi kan caste referencen til **`C`**.

Når vi arbejder videre med disse referencer, kan vi via **`Cref`** kun kalde metoder fra interfacet **`C`** og via **`B`** kun kalde metoder fra interfacet **`B`**. Vi har på denne måde fået opdelt **`D`**-objektets interface. Hvad kan det bruges til?

Need to know

Det kan bruges til at realisere "need to know" associeringer mellem objekter. En associering er en kobling, og koblinger skal holdes på et minimum. Ved at opdele interfacet har vi mindsket den kobling vi har til **`D`**-objektet, da vi nu ikke har nogen kobling til den del af **`D`**-objektets interface som vi ikke bruger.

Som at sætte penge i banken

Under udviklingen af en applikation synes det måske overflødigt at lave disse justeringer af koblinger, men under vedligeholdelsen forhindre de folk i at begå dumheder. Den største del af den tid der arbejdes med en kildetekst er under vedligeholdelsen - ikke under udviklingen. Og samtidig er de dumheder man begår under udviklingen for intet at regne mod dem man laver i uvidenhed under vedligeholdelsen. Derfor: At være opmærksom på koblinger under udviklingen er som at sætte penge i banken, der betales tilbage med renter under vedligeholdelsen.

Vores idealistiske mål er derfor at have så abstrakt et kendskab som muligt. Som med mange andre idealistiske mål skal man naturligvis foretage en fornuftig afvejning af mulighederne i den konkrete situation. Fornuften skal altid råde.

6.1 Eksempel: **`interface java.lang.Comparable`**

Lad os se et eksempel på et interface som optræder i Java's standard packages, nemlig **`java.lang.Comparable`**.

Indeholder kun én metode

Interfacet indeholder kun én metode, og en klasse der implementerer det, forpligtiger sig derfor ikke til det store. Signaturen for metoden er:

```
int compareTo( Object )
```

Vi kender metoden fra kapitlet: "Klasser", og grunden til at vi valgte den samme signatur dér, var dette interfaces eksistens i Java's standard packages (I kapitlet "Klasser", brugte vi dog ikke **`Object`** som parameter-type).

Sammenligne objekter

Metoden bruges som bekendt til at sammenligne objekter. Dens implementation definerer en ordning på mængden af instanser af den pågældende klasse. Den primære anvendelse findes man i forbindelse med sortering (som vi senere skal se i kapitlet "Sortering" under "Datastrukturer").

Betragt følgende metode:

Source 14: *Sorterings-metode*

```
public void sort( Comparable[] tabel ) {
  // sorterer elementerne i 'tabel'

  ...
}
```

Kan kun sammenligne

Metoden sorterer elementerne i det array den modtager som parameter. Den anvender sin viden om, at alle elementerne i arrayet har implementeret interfacet **`Comparable`**, og kan derfor bruge metoden **`compareTo`** til at sammenligner elementerne med hinanden.

Metoden ved ikke andet om objekterne i arrayet, og vi undgår på den måde at lave andre koblinger, der ville være unødvendige.

7. Polymorfi og komposition

Hvad med selve polymorfien? Hvilken glæde har vi af den?

Lad os se et eksempel:

7.1 Eksempel: **`interface Log`**

I nogle programmer kan må få brug for at føre en log over hændelser.

Kaptajnens log

Man kender betegnelsen log, fra sømandslivet, hvor kaptajnen fører en log over hvad er sker hver dag; hvor skibet befinder sig; hvor det er på vej hen, osv. Det er en slags dagbog for skibet, der bruges til at registrere alle væsentlige hændelser.

Server log

I EDB-sammenhæng bruges en log f.eks. af en server, der registrerer hvem der logger på og af, hvornår og hvad de laver. Til denne brug kunne det være praktisk med et objekt som havde ansvaret for at føre denne log.

Hvad skal der konkret ske med log'en? Skal den skrives i en file på harddisken eller på skærmen eller noget andet?

`Log`-objektets problem, hvad der skal ske

Vi kan starte med at fastslå, at svaret på spørgsmålet kun kan vedrøre selve **`Log`**-objektet. Det er den der har ansvaret for at føre log'en, og det er dens problem om der skal skrives i en file eller på skærmen eller noget andet. Den eneste del af program, der ellers behøver at beskæftige sig med det, er den del der fortæller **`Log`**-objektet hvad den skal gøre med log'en. Vi vil derfor starte med at fokusere på klienternes abstrakte forhold til **`Log`**-objektet.

Kun kende én metode

Klienterne til **`Log`**-objektet, som sender det beskeder om hændelser det skal registrere, behøver kun kende en enkelt metode, vi kunne kalde den **`registrer`**, som registrerer en hændelse. Metoden kunne have signaturen:

```
void registrer( String )
```

Klienter kalder denne metode med en tekststreng, der beskriver hændelsen.

Interface

Da vi kun ønsker at koble klienterne til **`Log`**-objektet med denne ene metode, kan vi lave et interface som alle **`Log`**-klasser skal implementere:

```
public interface Log {
  public void registrer( String s );
}
```

Implementere interface

Vi kan nu lave en række klasser, der implementerer dette interface, som er forskellige mht. hvad de konkret gør ved log'en. Man kunne f.eks. lave følgende to klasser:

```
public class LogToFile implements Log {
  private String filename;

  public LogToFile( String fn ) {
    filename = fn;
  }

  public void registrer( String s ) {
    // skrive s i filen
    ...
  }
}

public class LogToScreen implements Log {

  public void registrer( String s ) {
    System.out.println( s );
  }
}
```

Den første klasse vil give instanser, der skriver til en file; hvis navn man giver med til konstruktoren (Vi implementerer ikke her file-operationerne, der skal realisere udskriften til filen, da det ligger udenfor vores emne). Den anden klasse implementerer **`registrer`** som en udskrift til skærmen.

Need to know

Vi har nu brugt vores viden om interfaces til at realisere det abstrakte "need to know" forhold som resten af programmet skal have til **`Log`**-objektet.

Skifte `Log`-objekt

Der er dog et design-problem som vi stadig mangler at behandle. Hvad skal der ske hvis vi skifter **`Log`**-objektet ud med et andet. Vi kunne f.eks. ønske at udskifte et **`Log`**-objekt, der udskriver til en file, med et der udskriver til skærmen. Vi skal fortælle det til alle objekter der bruger **`Log`**-objektet. De skal alle have besked om i stedet at bruge et andet **`Log`**-objekt. Det er naturligvis en kompliceret opgave at holde styr på alle objekter der bruger **`Log`**-objektet, og vi løser da også vores design-problem ved at anvende et design pattern vi allerede kender, nemlig **Handle Pattern**.

Handle Pattern

Vi indskyder et handle-objekt mellem de objekter, der bruger **`Log`**-objektet og selve **`Log`**-objektet. På den måde skal vi kun fortælle handle-objektet om skift af **`Log`**-objektet, og alle de andre objekter kan uforandret forsætte med at bruge handle-objektet som deres adgang til **`Log`**-objektet.

Vores handle-objekt kunne realiseres som en instans af følgende klasse:

```
public class LogHandle implements Log {
  private Log logger;

  public LogHandle( Log lgr ) {
    setLog( lgr );
  }
```

```
  public void setLog( Log lgr ) {
    logger = lgr;
  }

  public void registrer( String s ) {
    logger.registrer( s );
  }
}
```

Som ethvert andet handle-objekt sender den kaldet af registrer videre til det bagved liggende objekt. Handle-objektet har en set-metode så man kan skifte **`Log`**-objektet.

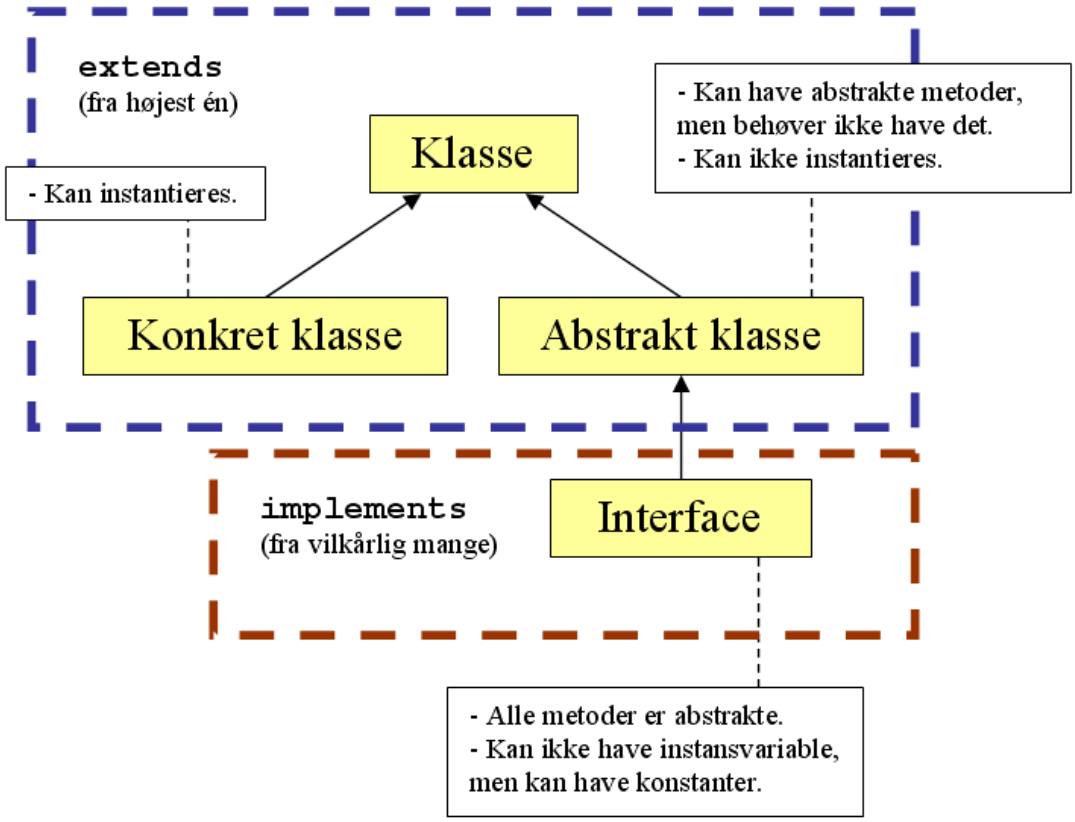
Bemærk at **`LogHandle`** implementerer **`Log`**-interfacet. Det betyder at klienter ikke vil være klar over, at de anvender et handle - det vil ikke kunne ses af deres kildetekst.

Løs kobling

Vi har her set, hvordan vi kan bruge polymorfi og et design pattern til at realisere en løs kobling, der giver et smidigt design der er let at arbejde med - når det først er lavet.

8. Relationer mellem forskellige slags klasser

Følgende figur opsummerer sammenhængen mellem de forskellige former for klasser:



Figur 8: *Relationer mellem forskellige slags klasser*

Som man ser er et **interface** en speciel **abstrakt klasse**, og en **abstrakt klasse** er en speciel **klasse**. På den måde beskriver man de forskellige former for klasser ved en nedarvningsrelation, idet man specialiserer klasse-begrebet når man bevæger sig nedad i foguren, mens man genereliserer når man bevæger sig opad.

---

fodnoter:

1. En af grundene til at anvende **final**, kan være at man fra et kommercielt synpunkt ikke ønsker at brugere af en package skal have mulighed for at specialisere klasserne. Hvis kunderne ønsker flere muligheder, kan de sende en mail til vores supportafdeling - vi vil dernæst forbedre vores produkt og naturligvis kræve betaling for en opdatering.

2. Tallet efter **@** er ikke *rigtig* den virtuelle adresse - det er objektets **hashværdi**. Ifølge kommentarerne i kildeteksten til **java.lang.Object.hashCode()** anbefales det, mere eller mindre, at man omformer **objektets virtuelle adresse** til en integer og anvender dette som hashværdi. Jeg antager derfor, at det er det Sun har gjort i forbindelse med deres

egen virtuelle maskine, og at hashværdien derfor er lig den virtuelle adresse - forudsat at det virtuelle adresserum kan adresseres med en integer.

Repetitionsspørgsmål

1. Hvad betyder ordet "polymorfi"?
2. Hvad vil det sige at have et abstrakt kendskab til et objekt?
3. Hvilke objekter kan en reference referere til?
4. Hvad er casting af referencer i forhold til casting af primitive typer?
5. Hvordan kan man teste om et objekt er af en given klasse?
6. Hvad er en abstrakt klasse?
7. Hvordan markerer man, at en klasse er abstrakt i et klassediagram?
8. Hvad er en **`final`** klasse?
9. Hvad er en abstrakt metode?
10. Hvad er et **`interface`** i Java?
11. Hvordan markerer man, at en klasse er et interface i et klassediagram?
12. Hvad specielt er der ved **`class Object`**?
13. Hvilke velkendte metoder er erklæret i **`class Object`**?
14. Hvorfor vil vi gerne have abstrakte kendskaber?
15. Hvordan kan man opdele et interface?
16. Hvorfor vil vi gerne have "need to know"-associeringer?
17. Hvori består polymorfien i **`Log`**-eksemplet?

Svar på repetitionsspørgsmål

1. "Flere-former".
2. At man ikke ved præcist hvilken klasse objektet er en instans af.
3. Instanser af referencens klasse, eller subklasser til referencens klasse.
4. I modsætning til casting af primitive typer, hvor der sker en konvertering, er det her kun et spørgsmål om at berolige compileren så den vil oversætte det.
5. Vha. **`instanceof`**-operatoren.
6. En klasse der ikke kan bruges til at instantiere objekter.
7. Ved at skrive navnet med *skrå skrift* (*italic*).
8. En klasse der ikke kan nedarves fra.
9. En metode, der ikke er implementeret i klassen, men skal implementeres i subklasser.

   En abstrakt klasse, hvor alle metoder er abstrakte.

10

11 Man skriver **`<<interface>>`** foran klassenavnet. Selv om et interface er en ultimativ abstrakt klasse, skrives klassenavnet *ikke* med skrå skrift (italic)

12 Den er direkte eller indirekte superklasse for alle andre klasser, og har ikke selv nogen superklasse.

13 **`equals`** og **`toString`**.

14 Ellers kan vi kun referere til objekter af en bestemt klasse. Med et abstrakt kendskab kan man referere til instanser af forskellige klasser, blot de har en fælles superklasse.

15 Ved at lave **`interface`**'s for hver del af det samlede interface, og implementere dem i den samme klasse.

16 Så fristes man ikke til at lave unødvendige koblinger.

17 Den består i, at man kalder den samme metode **`registrer`**, men det er forskelligt hvad der sker alt efter hvilken konkret klasse **`Log`**-objektet er en instans er af.