Innehåll

Samspel mellan olika klasser

- Icke-triviala program delas upp i olika klasser
- Relaterade klasser samlas i ***paket***, t ex `java.awt`
- Nya klasser kan definieras m h a befintliga klasser genom att använda sig av deras metoder
- Kräver att klasserna är "synliga" i samma kontext (giltiga i ***samma scope***)
  - ➨ Finns definierade i samma scope (katalog eller `package`)
  - ➨ Måste importeras från en annan scope (`import`)

Skuggning av namn

- **OBS!** Nya deklarationer skuggar giltiga identifierare
- ➨ Man kommer inte längre åt det skuggade namnet

- Se `Rektangel` vs `Rectangle` uppgiften

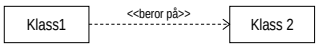
- Använd ***full kvalificering*** (kontext + namn) för att komma åt skuggade namn
- Exempel (se `shapes` exemplet):

```
import java.awt.*;
public class Rectangle
{
      canvas.fill(new java.awt.Rectangle(…);
}
```


Klassrelationer 1

- Klasserna kan använda sig av varandra på olika sätt
- Användningen innebär att klasserna blir beroende av varandra
- Ändringar i en klass kan påverka funktionaliteten i den andra

- Notation för allmänt beroende (***dependency***):



Klassrelationer 2

- Det finns tre speciella kategorier av beroenden

- Anropa metoder
- ➨ ***Association***

- Ha objekt som attribut
- ➨ ***Komposition***

- Dela på attribut och metoder
- ➨ ***Arv***

Ju starkare relation desto högre påverkan

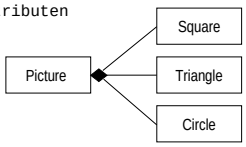
Association

```
import java.awt.Polygon;

public class Triangle
{
   …
   private void draw()
   {
      DrawingPaper paper = DrawingPaper .getPaper();
      …
      paper.fill(new Polygon(xpoints, ypoints, 3));
      …
   }
}
```

Komposition

```
public class Picture
{
   private Square wall;
   private Square window;
   private Triangle roof;
   private Circle sun;

   public Picture()
   {
      // initialisera attributen
   }
   …
}
```

Finns i exempelmappen



Arv: Problemet

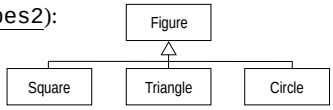
- Klasserna `Square`, `Triangle`, `Circle` och `Rektangel` i `shapes` exemplet ganska lika
  - Nästan samma attribut
  - Nästan samma metoder
  - Mycket kod "dubbelt" (klipp och klistra)

➨ Onödig arbete
➨ Onödig mycket kod
➨ Fel måste fixas på många ställen
➨ Ändringar/ tillägg måste göras på många ställen
➨ **Ej flexibelt**

Arv: Lösningen

- Definiera gemensamma attribut och metoder en enda gång i en *superklass*
- Ärver dessa attribut och metoder till nya *subklasser*
- Lägg till specifika attribut och metoder i subklasserna
- Subklassen är en mer specifik version av superklassen
- Varje superklass kan ha många subklasser

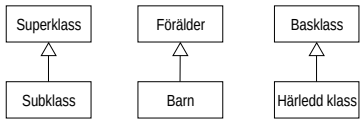
- Exempel (`shapes2`):



Härleda subklasser i Java

- Det reserverade ordet `extends` används för att åstadkomma arvet
- I Java finns bara ***enkelt arv*** för klasser
- Varje subklass kan ha ***högst en superklass***
- Alla klasser ärver från klassen `Object`

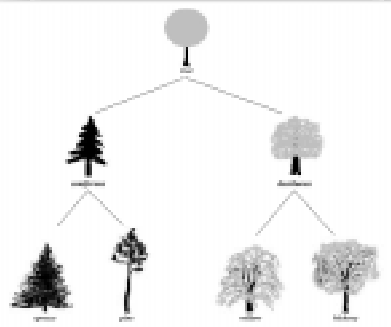
- Begrepp:



`Triangle` extends `Figure` 1

```
public class Triangle extends Figure
{
   // Attribut för Triangle utöver de som redan finns
   // för Figure (de har ju ärvts)

   // Konstruktor(er)

   // Metoder för Triangle utöver de som redan finns
   // för Figure (de har ju ärvts)
}
```


Arv och abstraktion



Arv och modifierare

- Modifierarna avgör vad som ärvs
- Attribut och metoder som är `public` ***ärvs***
- … medan `private` ***inte ärvs***
- **OBS!** Attribut bör inte vara `public`

- Allt som ärvs kan refereras direkt i subklassen, som om det var deklarerat där
- …men, även en metod eller ett attribut som inte ärvs, är definierad för subklassen
- …och kan refereras indirekt genom superklassens metoder
  - ➨ `Get`/ `set` metoder (***accessors***)

Arv och konstruktorer

- Konstruktorer ***ärvs inte***, trots att de (oftast) är deklarerade `public`

- Vi behöver dock tillgång till den överordnade klassens konstruktor för att initialisera de ärvda "delarna"

- Reserverade ordet `super` gör det möjligt att referera till superklassen
  - Måste vara första sats i metoden
  - `Super()` anropas automatiskt om programmeraren inte gör det
  - Om man vill skicka med parameter måste anropet finnas

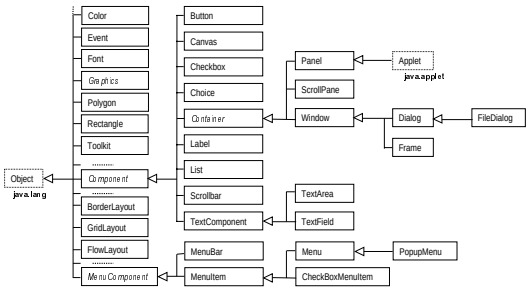
`Triangle` extends `Figure` 2

```
public class Triangle extends Figure
{
   private int height;
   private int width;

   /**
    * Create a new triangle with default height & width.
    */
   public Square()
   {
      super();   <----------------------------  Sköter om "konstruktionen" och
      height = 30;                              initialiseringen av den ärvda delen
      width = 40;
      draw();
   }
   …
}
```


Klasshierarkier

- Subklasser kan i sin tur ha subklasser
- God klassdesign placerar gemensamma egenskaper så högt som möjligt i hierarkin
- God klassdesign lägger bara till ett fåtal nya attribut och metoder åt gången
- Klasshierarkier måste ofta modifieras och utökas
- Ingen generell design som duger överallt

En klasshierarki: `AWT`



Klassen `Object`

- Alla klasser härleds från klassen `Object`
- Om inget arv anges, är klassen automatiskt subklass till `Object`
- Klassen `Object` är alltså rot i alla hierarkier

Object
… // `==´ och `=´ för objekt + boolean equals (Object obj) + Object clone ( ) + String toString ( ) + Class getClass ( ) # wait (long timeout) …

Sammanfattning

- Arv tillåter att speciella klasser skapas på ett enkelt sätt
- Tre typer av beroenden:
  - Arv hårt bundet
  - Komposition
  - Association löst bundet

- Se `Database` klassen i `people2` exemplet