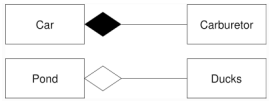
Föreläsning 3
Arvsmekanismen
Variabler och typer
Typer och subtyper

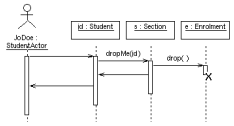
Grundbegreppen i objektorienterad design

Encapsulation Abstraction & Information Hiding

Composition Nested Objects

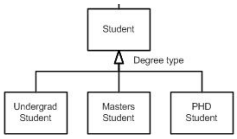


Distribution of Responsibility *Separation of concerns*



Message Passing *Delegating responsibility*

Inheritance *Conceptual hierarchy, polymorphism and reuse*



Vad är UML?

Unified Modeling Language (UML) är att objektorienterat modellerings-språk.

Språket är standardiserat av OMG (Object Management Group).

UML används för att:

- visualisera
- specificera
- konstruera
- dokumentera

UML omfattar:

- syntax: hur symboler får se ut och kombineras.
- semantik: symbolernas betydelse.

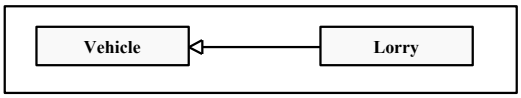
Relationer i UML

Det finns fem sorters relationer mellan klasser:

- En *generalisering* beskriver att en klass är en subklass till en annan klass
- En *realisering* beskriver att en klass implementerar ett interface.
- En *association* beskriver att en klass har attribut av en annan klass eller vice versa.
- En *inkapsling* beskriver att en klass är deklarerad inuti en annan klass för att göra denna klass osynlig för andra klasser.
- Ett *beroende* beskriver en relation mellan två klasser, av annat slag än ovanstående, som medför att den beroende klassen kan behöva modifieras om den andra klassen förändras.

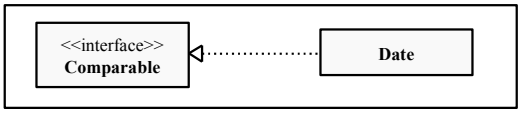
Relationer i UML-diagram

Generalisering:



Generalisering används då en klass utvidgar en annan klass, d.v.s. att en subklass skapas från en superklass. Ritas som en ofylld triangelpil mot superklassen.

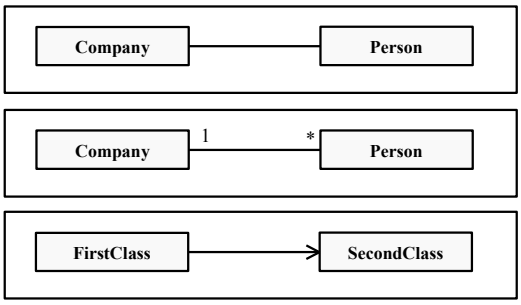
Realisering:



Realisering används då en klass implementerar ett interface. Ritas som en streckad ofylld triangelpil mot interfacet.

Relationer i UML-diagram

Association:



När en klass innehåller attribut av en annan klass kallas detta association. Ritas med ett heldraget streck.

Man kan för en association ange multiplicitet, d.v.s hur många instanser av den ena klassen som finns i den andra klassen.

För att visa vilken klass i associationen som har attribut av den andra klassen används navigationspilar.

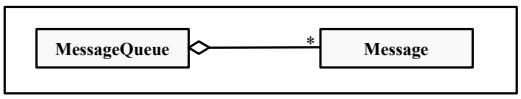
Relationer i UML-diagram

Det finns två olika typer av associationer; aggregation och komposition.

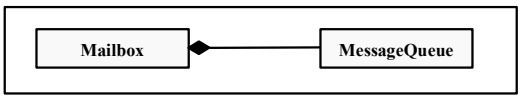
Vid aggregation kan en instans vara associerad med flera ägarobjekt. Försvinner ägarobjektet kommer den associerade instansen att leva vidare.

Vid komposition kan en instans endast vara associerad med ett ägarobjekt. Försvinner ägarobjektet så försvinner också den associerade instansen.

Aggregation:

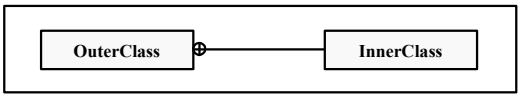


Komposition:



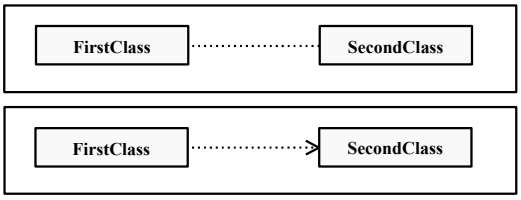
Relationer i UML-diagram

Inkapsling:



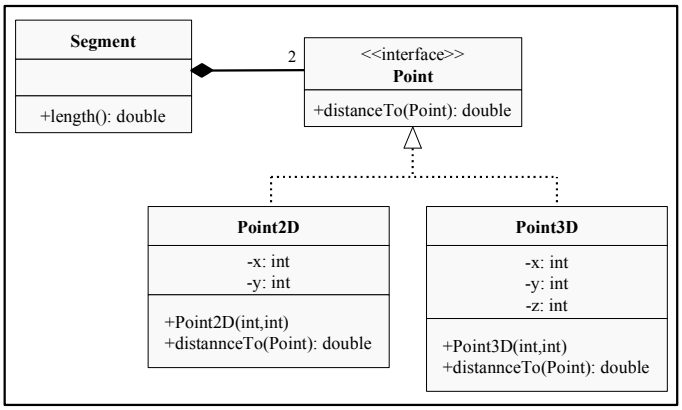
Anger att en klass är inuti en annan klass.

Beroende:



Anger att en klass har en metod där någon parameter eller returvärdet är av annan klass. Ritas med en streckad linje. För att ange att ett beroende är riktat används navigationspilar.

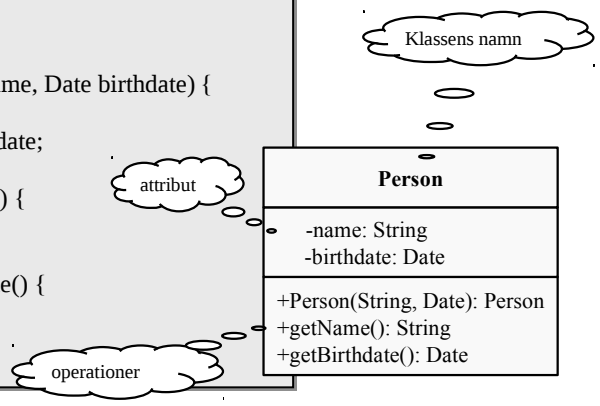
Ett enkelt klassdiagram



Klasser

Något förenklat uttryckt, är en klass en mall från vilken man kan skapa objekt.

```
import java.util.Date;
public class Person {
   private String name;
   private Date birthdate;
   public Person(String name, Date birthdate) {
      this.name = name;
      this.birthdate = birthdate;
   }//constructor
   public String getName() {
      return name;
   }//getName
   public Date getBirthdate() {
      return birthdate;
   }//getBirthdate
}//Person
```



Begreppet arv

Begreppet arv (*inheritance*) är ett av grundkoncepten inom objekt-orientering. Arv är det koncept som möjliggör polymorfism och är en viktig mekanism för återanvändning av kod.

Arv betyder att en klass erhåller vissa egenskaper genom att överta de egenskaper som en annan klass eller ett interface har.

Arv innebär att man kan relatera olika kategorier av objekt enligt en *arvshierarki*.

Arv beskriver en *”är”-relation*:

en bil *är* ett fordon
en sportbil *är* en bil
en mobiltelefon *är* en telefon
en klockradio *är* en radio och en klockradio *är* en klocka

I det sista exemplet har vi *multipelt arv*, en klockradio är både en klocka och en radio.

Arv i Java

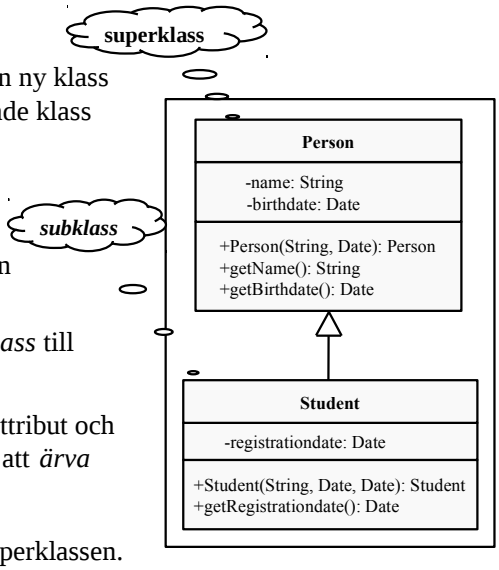
När en klass övertar egenskaperna från en annan klass kallas det för *implementationsarv*. Vid implementationsarv ärvs (implementationen för) alla instansvariabler och klassvariabler (även privata), och alla icke-privata metoder. Konstruktorer ärvs inte.

När en klass övertar egenskaperna från ett interface kallas det för *specifikationsarv*.

Java tillåter inte multipelt implementationsarv, varför multipelt arv i Java alltid involverar specifikationsarv.

Implementationsarv

Vid implementationsarv skapar man en ny klass genom att utgå från en redan existerande klass och lägger till nya egenskaper.



Den nya klassen är en *subklass* till den existerande klassen.

Den existerande klassen är en *superklass* till subklassen.

Subklassen får samtliga egenskaper (attribut och operationer) från superklassen genom att *ärva koden* från superklassen.

En subklass är en *specialisering* av superklassen.

Implementation av `Student`

extends anger att klassen är en subklass

```
import java.util.Date;
public class Student extends Person {
   private Date registrationdate;
   public Student(String name, Date birthdate, Date registrationdate) {
      super(name, birthdate);
      this.registrationdate = registrationdate;
   }//constructor
   public Date getRegistrationdate() {
      return registrationdate;
   }//getRegistrationdate
}//Student
```

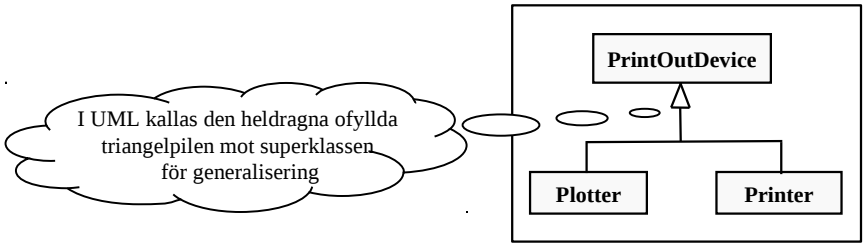
Har inte direkt access till name och birthdate i superklassen Person då dessa är **private**

Klassen `Student` är en *specialisering* av klassen `Person`.

Klassen `Person` är en *generalisering* av klassen `Student`.

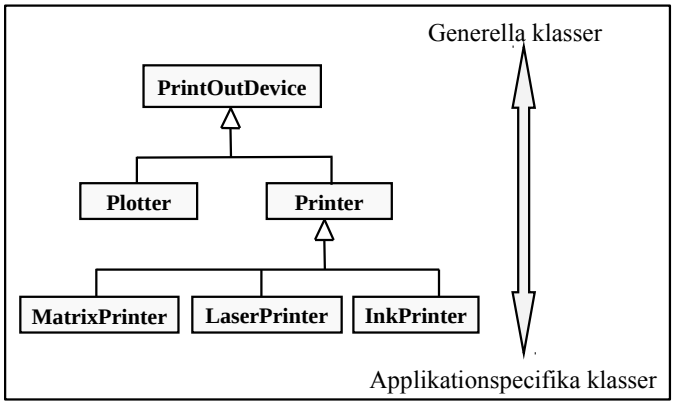
Implementationsarv

En superklass får ha ett godtyckligt antal subklasser.



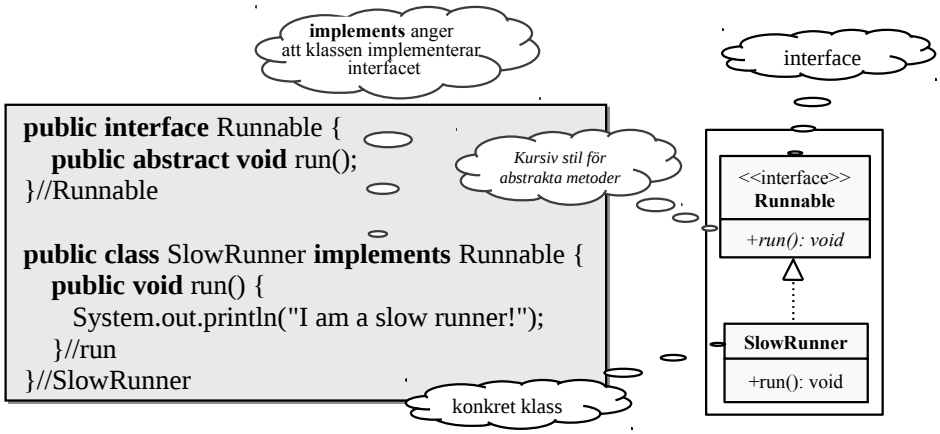
Arv

Arv innebär att man kan relatera olika kategorier av objekt enligt en *arvshierarki*.



Specifikationsarv

Vid *specifikationsarv* skapas en klass genom att klassen *implementerar* de beteenden (=instansmetoder) som specificeras av ett interface. Metoderna i ett interface saknar implementationer, de är *abstrakta*.



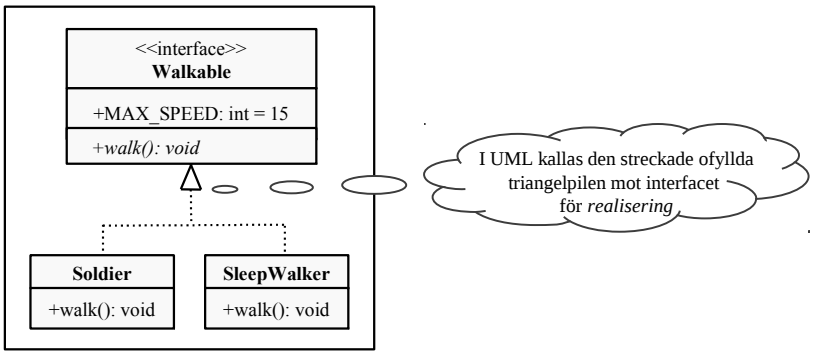

```
public interface Runnable {
   public abstract void run();
}//Runnable

public class SlowRunner implements Runnable {
   public void run() {
      System.out.println("I am a slow runner!");
   }//run
}//SlowRunner
```

Specifikationsarv

Flera klasser kan implementera samma interface.

Förutom abstrakta instansmetoder kan ett interface innehålla statiska konstanter.



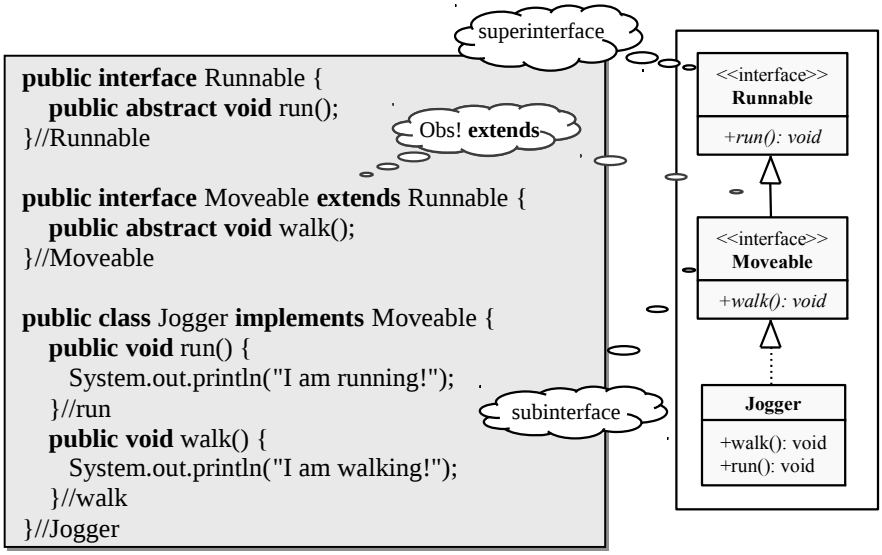
Interface

Ett interface kan utöka ett eller flera andra interface.

```
public interface Runnable {
   public abstract void run();
}//Runnable

public interface Moveable extends Runnable {
   public abstract void walk();
}//Moveable

public class Jogger implements Moveable {
   public void run() {
      System.out.println("I am running!");
   }//run
   public void walk() {
      System.out.println("I am walking!");
   }//walk
}//Jogger
```



Abstrakta klasser

En *abstrakt klass* är en klassen där en eller flera metoder kan sakna implementation. I Java anges att en klass är abstrakt med det reserverade ordet **abstract**.

En metod som saknar implementation kallas för en *abstrakt metod*. I Java anges att en metod är abstrakt med det resererade ordet **abstract**.

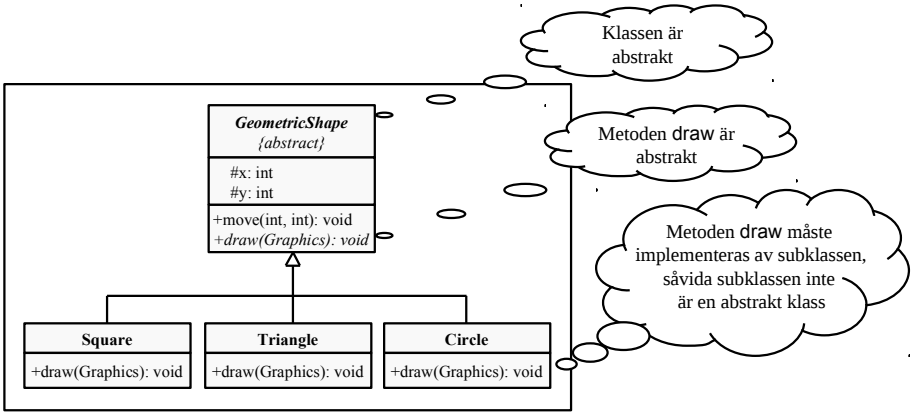
En klass måste deklareras som abstrakt om klassen innehåller en eller flera abstrakta metoder.

En abstrakt klass beter sig exakt som en vanlig klass med ett enda undantag:

- *man kan inte skapa instanser av en abstrakt klass*

Det är subklassernas uppgift att implementera de abstrakta metoderna.

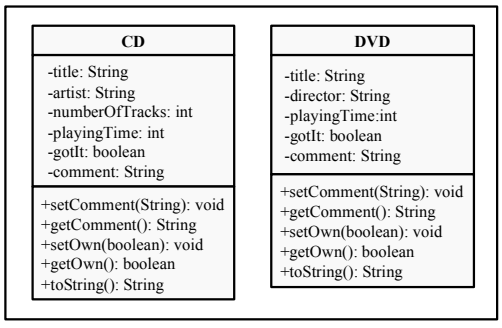
Abstrakta klasser: Exempel



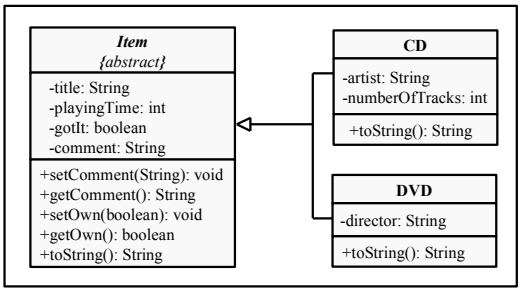
När en subklass implementerar en abstrakt klass *överskuggar* (*override*) subklassen de abstrakta instansmetoderna i superklassen.

Användning av abstrakta klasser: Generalisering

Om vi har flera klasser i koden som har stora likheter och innehåller duplicerad kod, är det lämpligt att införa en abstrakt klass för att städa upp i koden.



Användning av abstrakta klasser: Generalisering



I den abstrakta superklassen läggs alla gemensamma dataattribut och alla gemensamma metoder. Subklasserna innehåller de dataattribut och metoder som är unika för den specifika subklassen.

> *Klassen `Item` innehåller inga abstrakta metoder! Varför har vi då gjort klassen Item till en abstrakt klass?*
>
> *Varför är metoden toString överskuggad?*

Implementationsarv: super

En subklass kan referera till sin ***direkta superklass*** med **super**

- används i subklassens konstruktorer för att anropa superklassens konstruktorer

  super(*parameterlist*);

  Om subklassens konstruktor gör anrop till någon av superklassens konstruktorer skall detta anrop ligga som ***första sats*** i konstruktorn.

  Anropas inte någon av superklassens konstruktorer, sker ett ***automatiskt anrop*** till superklassens default-konstruktor (konstruktorn utan parametrar):

  - saknar superklassen helt konstruktorer propagerar anropet uppåt i klasshiearkin
  - saknar superklassen default-konstruktor men har någon annan konstruktor uppkommer ett kompileringsfel.

super i konstruktorer

```
public class Item {
    private String title;
    private int playingTime;
    private boolean gotIt
    private String comment;
    public Item(String title, int playingTime, boolean gotIt) {
        this.title = title;
        this.playingTime = playingTime;
        this.gotIt = gotIt;
    }//constructor
    . . .
}//Item
```

```
public class CVD extends Item {
    private String director;
    public Item(String title, int time, boolean gotIt, String director) {
        super(title, time, gotIt);
        this.director = director;
    }//constructor
    . . .
}//CD
```

super-anrop i metoder

När en subklass överskuggar (*override*) en instansmetod eller döljer (*hiding*) klassmetoder i superklassen blir dessa metoder inte direkt åtkomliga för subklassen.

Eftersom en subklass utvidgar egenskaperna i superklassen är ofta en överskuggad metod en utvidgning av en del av metoden som implementeras i subklassen.

En subklass kommer åt en överskuggad metod `overridedMethod(...)` respektive en dold metod `hiddenMethod(...)` med konstruktionen:

super.overridedMethod(...)
super.hiddenMethod(...)

Exempel:

```
public class CD extends Item {
    . . .
    @Override
    public String toString() {
        return super.toString() +  "\nartist: " + artist +
                "\ntracks: " + numberOfTracks + " \n");
    }//toString
}//CD
```

Interface vs abstrakta klasser

Abstrakta klasser har en fördel jämfört med interface typer:

- de kan implementera gemensamma beteenden

Men de har också en allvarlig nackdel:

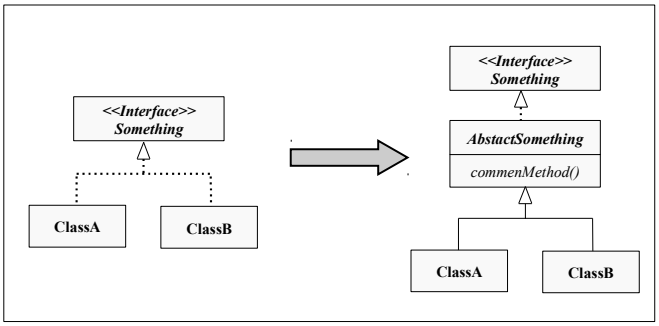
- en klass kan endast göra ”extend” på en klass, men kan implementera många olika interface.

Slutsats:

- Använd aldrig abstrakta klasser där alla metoder saknar implementation, använd istället interface.
- Använd abstrakta klasser för att faktorisera ut gemensam kod.

Abstrakta klasser: eliminering av duplicerad kod

Abstrakta klasser skall användas för att eliminera duplicerad kod och/eller implementera *default* beteenden



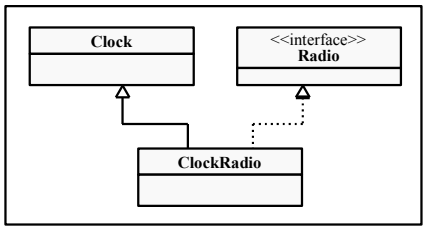
Används mycket flitigt i Java's API.

Multipelt arv i Java

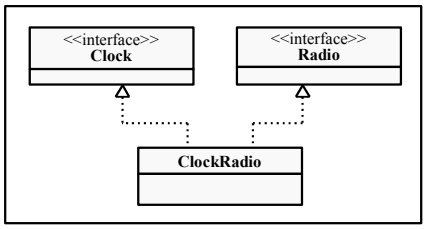
I Java kan en klass ärva från

- endast en klass
- ett godtyckligt antal interface.

I Java involverar *multipelt arv* därför alltid specifikationsarv.



ClockRadio **extends** Clock **implements** Radio



ClockRadio **implements** Clock, Radio

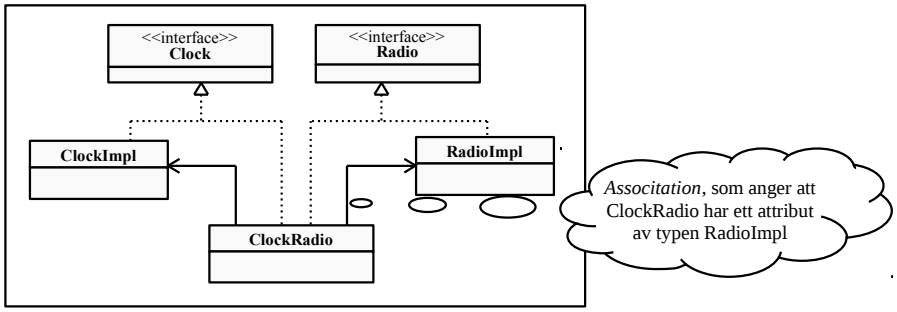
Varför har man i Java förbjudit multipelt implementationsarv, vilket t.ex. är tillåtet i C++?

Multipelt arv och delegering

Två viktig designprincip, som vi kommer att diskutera under kursen är

- *Programmera mot gränssnitt, inte mot konkreta klasser*
- *Föredra delegering framför implementationsarv*

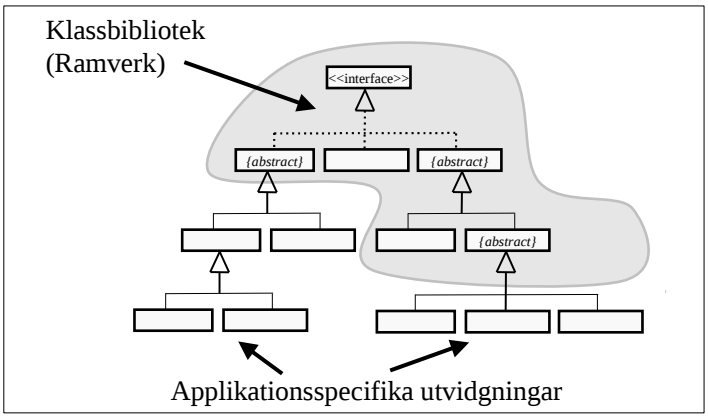
Multipelt arv bör därför implementeras enligt följande:



Vilka fördelar erhålls med denna design?

Javas API

Javas API innehåller en stor uppsättning interface, abstrakta klasser och konkreta klasser som kan ärvas och utvidgas för applikationsspecifika behov.



Utvidgning av API-klassen `Rectangle`

När vi skall utveckla en klass är det alltid lämpligt att titta i API:n om det finns någon klass som uppfyller våra behov, eller om det finns någon lämplig klass som vi kan utvidga för att uppfylla de behov vi har.

Exempel:

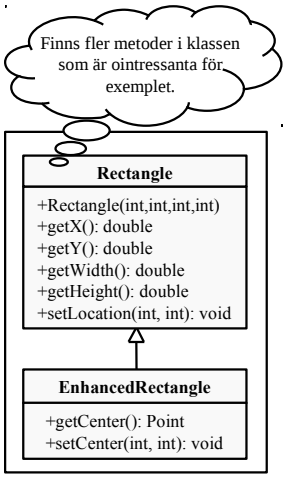
I ett ritprogram behöver vi kunna hantera rektanglar.

Klassen `java.awt.Rectangle` uppfyller nästan våra behov. Vi behöver dock metoderna

- `getCenter()`, för att avläsa centrum
- `setCenter(int x, int y)`, för att sätta centrum

som klassen `java.awt.Rectangle` saknar.

En lösning är att skapa en subklass till klassen `java.awt.Rectangle` och utöka subklassen med metoderna `getCenter` och `setCenter`.*



*Dock, som vi senare kommer att se, inte alltid den bästa lösningen!

Implementation av `EnhancedRectangle`

```
import java.awt.Rectangle;
import java.awt.Point;
public class EnhancedRectangle extends Rectangle {

   public EnhancedRectangle(int x, int y, int w, int h) {
      super(x, y, w, h);
   }//constructor

   public Point getCenter() {
      return new Point((int) (getX() + getWidth()/2), (int) (getY() + getHeight()/2));
   }//getCenter

   public void setCenter(int x, int y) {
      setLocation(x -(int) getWidth()/2, y - (int) getHeight()/2);
   }//setCenter
}//EnhancedRectangle
```

Klassen `EnhancedRectangle` är en *specialisering* av klassen `Rectangle`.

Objekt i Java

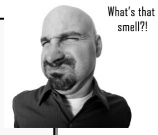
Java är ett objektorienterat språk:

- program skapar objekt
  obj = **new** SomeClass();
- program anropar metoder i objekt
  obj.someMethod();
- program hämtar värden i objekt-variabler
  x = obj.someField;
- program lägger nya värden i objekt-variabler
  obj.sameField = 5;

Om nu Java är objektorienterat, var har vi objekten i programmet?

Exempel:

Vilka uttryck i detta program beräknas till objekt?

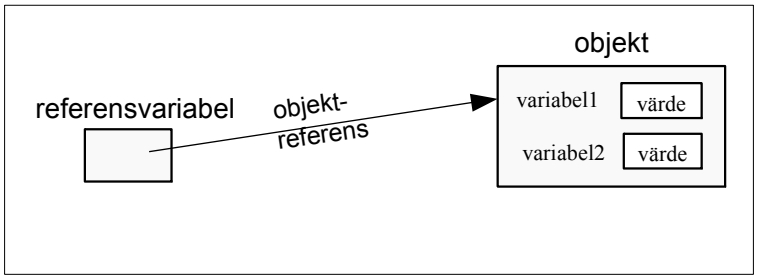



```
//-- Bad code, only used for illustration --//
public class NumberList {
   int num;
   NumberList next;
   public NumberList(int i, NumberList nl) {
      num = i;
      next = nl;
   }//constructor
}//NumberList
```

```
public class TestNumberList {
   . . .
   public void someMethod() {
      . . .
      NumberList nl_1 = new NumberList(52, null);
      NumberList nl_2 = new NumberList(71, nl_1);
      NumberList nl_3 = nl_2.next;
      nl_2.next.num = 22;
      int x = nl_3.num;
   }//someMethod
}//TestNumberList
```

Referensvariabler

Det finns inga Java-uttryck som är objekt. Istället hanterar Java-program *referenser till* objekt.



Åter till exemplet

```
1)  NumberList nl_1 = new NumberList(52, null);
2)  NumberList nl_2 = new NumberList(71, nl_1);
3)  NumberList nl_3 = nl_2.next;
4)  nl_2.next.num = 22;
5)  x = nl_3.num;
```

Uttryck i satserna ovan som beräknas till primitiva värden:

1) 52
2) 71
4) 22
5) nl_3.num

Uttryck i satserna ovan som beräknas till objekt-referenser:

1) **new** NumberList(52, **null**) och **null**
2) **new** NumberList(71, nl_1) och nl_1
3) nl_2.next

Värden av uttryck är primitiva värden eller objekt-referenser, aldrig objekt.

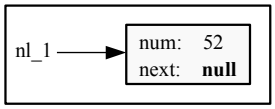
Vad spelar det för roll?

Skillnaden mellan ett objekt och dess referens är mycket viktig, då det gäller:

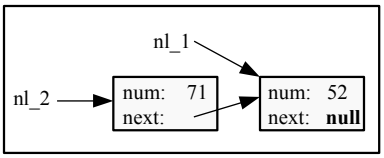
- exekvering
  - frågan om olika alias
  - frågan om dynamisk bindning
- typer
  - frågan om olika typer för objekt och deras referenser (skillnaden mellan *statisk typ* och *dynamisk typ*)

Beräkningarna i vårt exempel

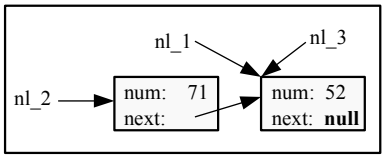
NumberList nl_1 = **new** NumberList(52, **null**);



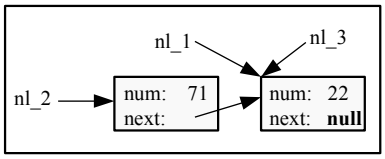
NumberList nl_2 = **new** NumberList(71, nl_1);



NumberList nl_3 = nl_2.next;

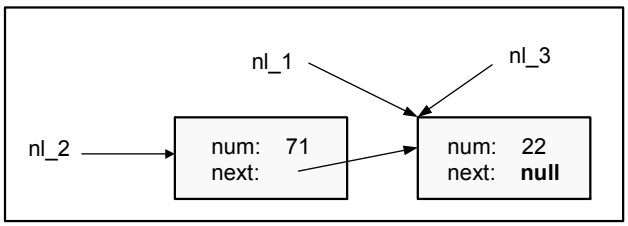


nl_2.next.num = 22;



Alias

Två eller flera referenser till samma objekt kallas *alias*.



Allt som görs via en referens påverkar alla alias till denna referens.

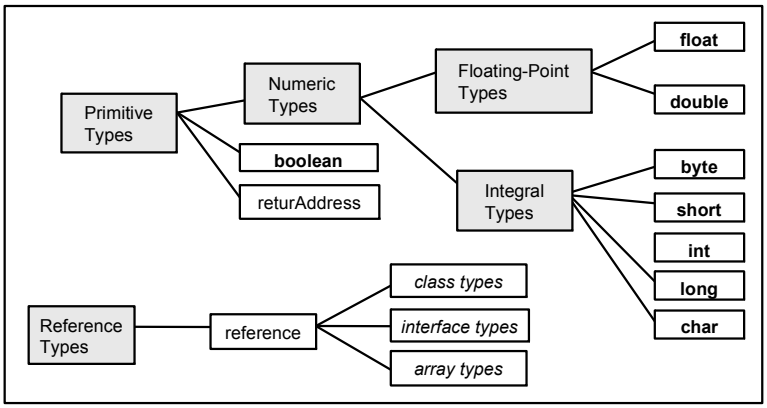
Risken att göra fel är stor.

Man måste därför vara mycket noggrann när man använder alias.

Typer i Java

Java har två olika sorters typer:

- Primitiva typer, dessa är inbyggda. Vi kan inte skapa nya primitiva typer.
- Referenstyper. Vi kan skapa nya referenstyper.



Värde eller referens?

Att det finns två olika sorters typer får följder för betydelsen av språkliga konstruktioner.

Värdesemantik (*by value*)

- inget delat tillstånd, dvs kopior saknas
- används för primitiva datatyper

Referenssemantik (*by referencs*).

- tillståndet kan vara delat (d.v.s. det kan finnas alias)
- används för referenstyper

Återkommande frågeställning:

Har vi värde- eller referenssemantik?

Definiera nya typer

Det finns två standardsätt i Java att definiera nya typer:

- genom att skapa nya interface

  public interface AnInterface { . . . }

  public interface ASecondInterface **extends** AnInterface { . . . }

- genom att skapa nya klasser

  public class AFirstClass { . . . }
  public class ASecondClass **extends** AClass{ . . . }
  public class AThirdClass **implements** AnInterface { . . . }

En typ definierar en värdemängd och de operationer som kan göras på denna värdemängd.

Definiera nya typer via klasser

Varje klass `C` introducerar en typ `C`.

- Uppsättningen *publika metoder* som klassen `C` tillhandahåller utgör mängden operationer för typen `C`.
- Alla instanser av klassen `C` och alla instanser av subklasser till `C` utgör värdemängden för typen `C`.

Definiera nya typer via klasser

Klassen `BankAccount` definierar en typ `BankAccount`.

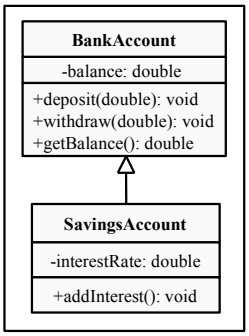
Klassen `SavingsAccount` definierar en typ `SavingsAccount`.

Typen `BankAccount` har operationerna `deposit`, `withdraw` och `getBalance`.

Typen `SavingsAccount` har operationerna `deposit`, `withdraw`, `getBalance` och `addInterest`.

Värdemängden för typen `BankAccount` utgörs av alla instanser av klassen `BankAccount` och alla instanser av klassen `SavingsAccount`.

Värdemängden för typen `SavingsAccount` utgörs av alla instanser av klassen `SavingsAccount`.



Typen `SavingsAccount` är en *subtyp* till typen `BankAccount`. Objekten i typen `SavingsAccount` är en delmängd av objekten i typen `BankAccount`.

Definiera nya typer via interface

Varje interface `I` definierar en typ `I`.

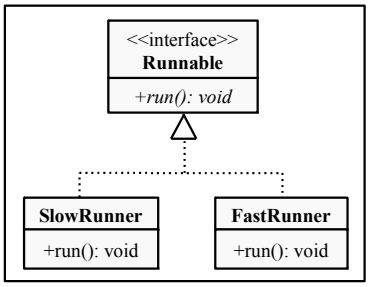
- Uppsättningen metoder som interfacet `I` definierar utgör mängden operationer för typen `I`.
- Värdemängden för typen `I` utgörs av
  - alla instanser av alla klasser C_i som implementerar interfacet I eller implementerar något subinterface till `I`, samt
  - alla instanser av alla subklasser till varje klass C_i.

Definiera nya typer via interface

Interfacet `Runnable` definierar en typ `Runnable`.

Klassen `SlowRunner` definierar en typ `SlowRunner`.

Klassen `FastRunner` definierar en typ `FastRunner`.



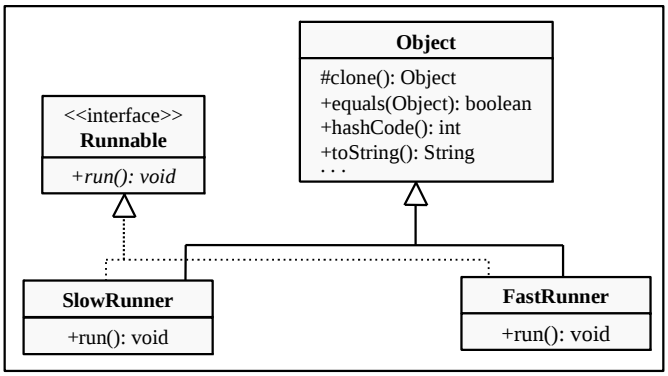
Typen `Runnable` har endast operationen `run`.

Mängden av objekt i typen `Runnable` utgörs av alla instanser av klassen `SlowRunner` och alla instanser av klassen `FastRunner`.

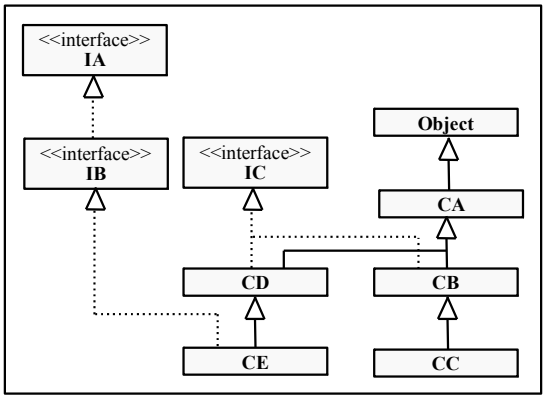
Typerna `SlowRunner` och `FastRunner` är subtyper till typen `Runnable`.

Klassen Object

Eftersom alla klasser är subklasser till klassen `Object` tillhör alltså ett objekt av typen `SlowRunner` eller typen `FastRunner` även typerna `Runnable` och `Object`.



Typer och subtyper



Vilka typer tillhör instanser av klassen CC?
Vilka typer tillhör instanser av klassen CE?
Vilka objekt är av typen IC?

Typer i Java

- Java är ett starkt typat språk.
- De operationer man skall kunna utföra på data i programmet skall bara vara de som anges av den typ som datat har.
- Om programmet kompilerar skall man inte under körning kunna få några fel som beror på typer. Som vi kommer att se senare i kursen uppfyller inte Java detta fullt ut.
- Förklaringen till varför saker är som de är bottnar ofta i typsäkerheten.

Typkompatibilitet

Typsystemet behöver inte vara hur strikt som helst, att t.ex. använda en **long** istället för en **int** innebär aldrig någon fara.

Det finns ett stort antal regler för vilka typer som får användas istället för en given typ, kallas *typkompatibilitet*.

Kompilatorn känner till dessa regler och utför om möjligt implicita typomvandlingar vid behov.

Typkompatibilitet

Exempel på godtagbara omvandlingar på primitiva typer är mindre till större (i bytes) och heltal till flyttal (kallas *widening*):

short -> **int**
int -> **float**
float -> **double**
long -> **float**

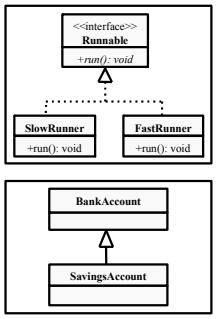
Följande tilldelningssatser är alltså korrekta:

```
int small = 10;
long large = small;
```

eftersom typen **int** är *kompatibel* med typen typen **long**.

Typkompatibilitet

För referenstyper gäller att *en subtyp är kompatibel med sina supertyper*, detta kallas för *polymorfism.*



Det är alltså tillåtet att skriva

```
//widening reference types
Runnable theWinner = new FastRunner();
Runnable theLoser = new SlowRunner();
BankAccount account = new SavingsAccount()
```

eftersom typerna `FastRunner` och `SlowRunner` är subtyper till typen `Runnable`, och typen `SavingsAccount` är subtyp till typen `BankAccount`.

Observera att vi har olika typer på objekten och på referenserna till objekten.

Som vi snart skall se är både typen av objektet och typen av dess referens signifikanta på olika sätt.

Innebär: Viktigt att förstå typer i Java.

Explicit typomvandling (casting)

Det är möjligt att säga åt kompilatorn att åsidosätta typkontrollen genom *explicit typomvandling*:

```
Runneable runner = . . .;
SlowRunner slow = (SlowRunner) runner;
```

Eftersom vi säger att `slow` är av typen `SlowRunner`, kontrolleras inte detta av kompilatorn. Ansvaret är vårt! Förhoppningsvis stämmer det, om `runner` är av typen `FastRunner` erhålls ett exekveringsfel.

Observera att explicit typomvandling inte förändrar ett objekt från en typ till en annan typ! Det är endast en indikation till kompilatorn att en referensvariabel *förutsätts* referera till ett objekt av angiven typ. I detta fall att referensvariabeln `slow` refererar till ett objekt av typen `SlowRunner`.

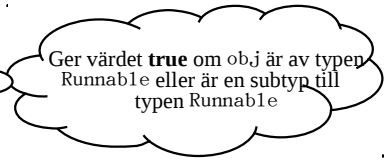
Explicit typomvandling kallas även *narrowing* och *downcasting*.

> Explicita typomvandlingar innebär alltid en risk!
> Typkontrollen sätts ur spel. Undvik!

Explicit typomvandling

För att kontrollera att tilldelning och explicit typomvandling är möjlig (dvs. bibehåller typsäkerheten), kan man för referenstyper använda **instanceof**-operatorn.

```
Object obj = . . .;
if (obj instanceof Runnable)
    Runnable r = (Runnable) obj;
```



Men även detta bör undvikas.

Typomvandlingar mellan helt orelaterade typer (då det inte finns något supertyp/subtyp förhållande) är inte tillåtet.

Återkommande frågeställning:

Är typerna kompatibla? Går det att omvandla A till B utan att typsäkerheten bryter samman.

Typer och subtyper: Sammanfattning

Det finns en speciell relation mellan typen av en superklass och typerna av dess subklasser, samt mellan typen av ett interface och typerna för klasser som implementerar interfacet:

- en subklass till en klass definierar en subtyp till superklassens typ
- en klass som implementerar ett interface definierar en subtyp till interfacets typ.

Om S är en subtyp till T, så är mängden av objekt i typen S en delmängd till mängden av objekt i typen T, och mängden operationer i typen T är en delmängd till mängden operationer i typen S.