



Objects First With Java
A Practical Introduction Using BlueJ

Abstraheringsteknik - fortsättningskurs

Abstrakta klasser
och
gränssnitt (interfaces)



Fler viktiga begrepp

- Abstrakta klasser
- Gränssnitt (Interfaces)
- Multipel ärvning

Simuleringar

- Aktiviteter som ibland simuleras av dataprogram:
 - stadstrafik
 - vädret
 - processer i atomkärnor
 - aktiemarknaden
 - miljöförändringar



Simuleringar

- Oftast bara partiella simuleringar.
- Innebär att man infört förenklingar:
 - Fler detaljer ger oftast bättre noggrannhet.
 - Fler detaljer kräver oftast mer resurser:
 - Processortid.
 - Minne.





Fördelar med simulering

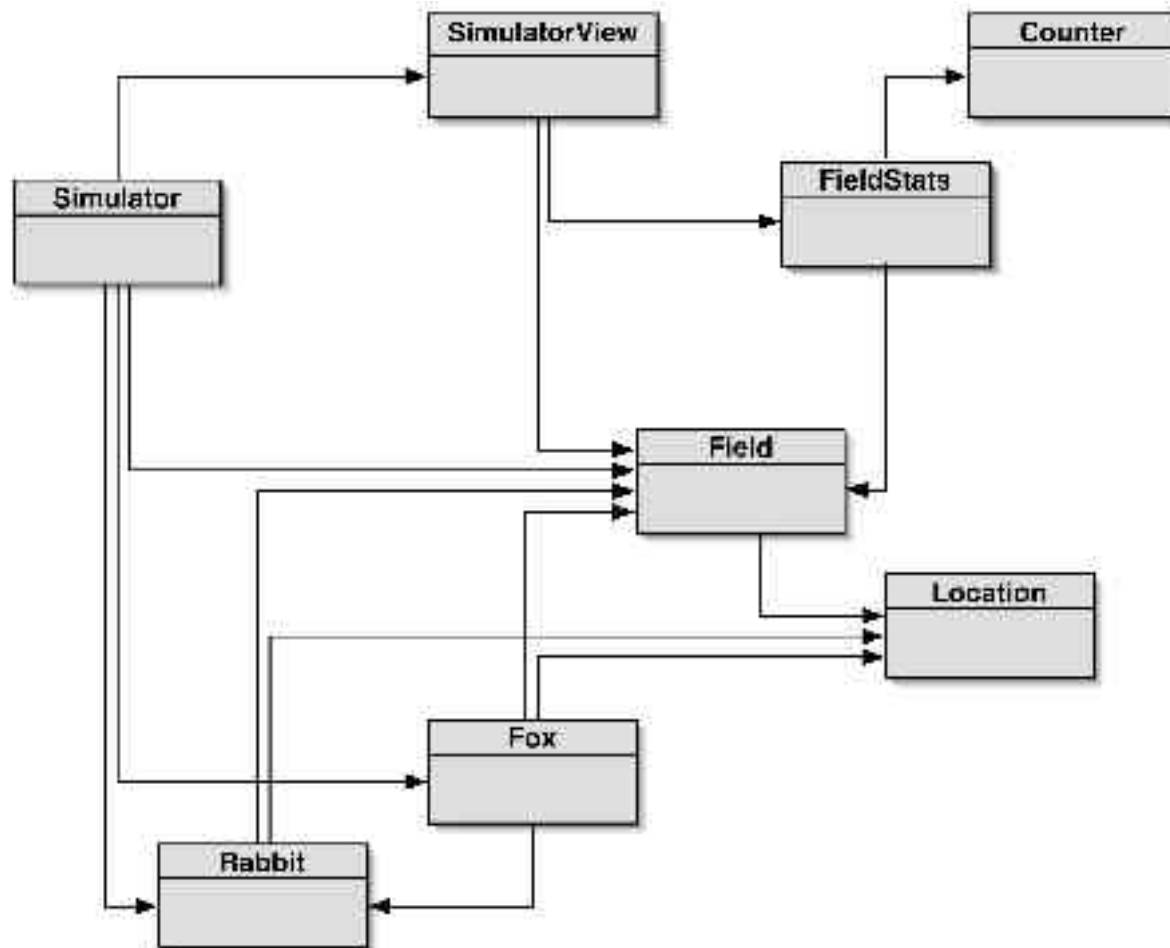
- Ger ofta underlag för bra prediktioner.
 - Vädret ..?
- Tillåter att man experimenterar.
 - säkrare, billigare, snabbare.
- Exempel:
 - ‘Hur kommer djurlivet att påverkas om vi anlägger en motorväg genom nationalparken?’



Simulering av rovdjurs jakt på bytesdjur

- I naturen finns det ofta en delikat balans mellan olika arter.
 - Få rovdjur innebär att antalet bytesdjur ökar.
 - Många bytesdjur innebär mer mat för rovdjuren.
 - Mer mat för rovdjuren innebär att antalet rovdjur ökar.
 - Många rovdjur innebär färre bytesdjur.
 - Färre bytesdjur innebär mindre mat för rovdjuren.
 - Mindre mat för rovdjuren innebär färre rovdjur.
 - Få rovdjur innebär ...

Projekt: Rävar och kaniner





Intressanta klasser

- `Fox`
 - Enkel modell av ett rovdjur.
- `Rabbit`
 - Enkel modell av ett bytesdjur.
- `Simulator`
 - Hanterar den övergripande simuleringen.
 - Hanterar en kollektion av rävar och kaniner.

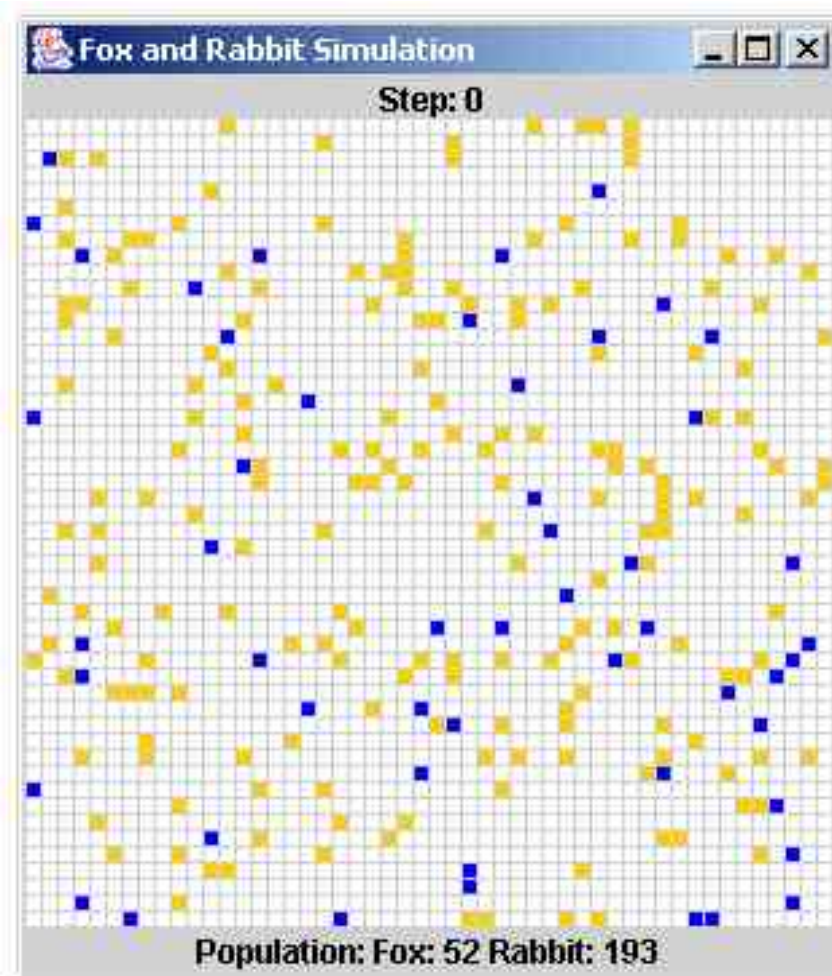


Återstående klasser

- `Field`
 - Representerar ett fält i 2D.
- `Location`
 - Representerar en position på fältet.
- `SimulatorView`, `FieldStats`, `Counter`
 - Räknar ut statistik och presenterar ett fönster som visar fältet.



Ett fält med rävar och kaniner



En kanins tillstånd

```
public class Rabbit
{
    Static fields omitted.

    // Individual characteristics (instance fields).

    // The rabbit's age.
    private int age;
    // Whether the rabbit is alive or not.
    private boolean alive;
    // The rabbit's position
    private Location location;

    Method omitted.
}
```

En kanins beteende

- Hanteras av `run i Rabbit` .
- Kaninens ålder ökas i varje simuleringssteg.
 - Kan innebära att kaninen dör av ålderdom.
- Kaniner som är tillräckligt gamla kan få ungar i varje simuleringssteg.
 - Det innebär att nya kaniner tillkommer varje simuleringssteg.



Förenklingar - Kaniner

- Kaniner har inte olika kön:
 - Alla är honor.
- En kanin kan få ungar varje simuleringssteg.
- Alla kaniner dör vid samma ålder.
- Andra?



En rävs tillstånd

```
public class Fox
{
    Static fields omitted

    // The fox's age.
    private int age;
    // Whether the fox is alive or not.
    private boolean alive;
    // The fox's position
    private Location location;
    // The fox's food level, which is increased
    // by eating rabbits.
    private int foodLevel;

    Methods omitted.
}
```

En rävs beteende

- Hanteras av hunt i Fox.
- Rävar åldras och dör av ålderdom.
- De blir hungriga.
- De jagar de kaniner som befinner sig nära.

Förenklingar - rävar

- Motsvarande förenklingar som för kaniner.
- Hur en räv jagar och äter kan modelleras på många olika sätt.
 - Skall man räkna antalet kaniner en räv äter?
 - Kommer en hungrig räv att jaga oftare?
- Viktig fråga: Vilka förenklingar påverkar det man vill ha ut av en simulering och vilka gör det inte?



Simulatorklassen

- Tre nyckelkomponenter:
 - Initieringarna i konstruktorn.
 - Metoden `populate`.
 - Varje djur får en slumpvis startålder.
 - Metoden `simulateOneStep`.
 - Itererar över hela populationen.
 - Två `Field`-objekt används: `field` och `updatedField`.



| simulateOneStep

```
if (animal instanceof Rabbit) {  
    Rabbit rabbit = (Rabbit) animal;  
    rabbit.run(updatedField, newAnimals);  
}  
else if (animal instanceof Fox) {  
    Fox fox = (Fox) animal;  
    fox.hunt(field, updatedField, newAnimals);  
}
```



Utrymme för förbättringar

- `Fox` och `Rabbit` liknar varandra mycket men har inte en gemensam superklass ...
- Klassen `Simulator` är tätt sammankopplad med andra klasser. Den känner t.ex. till rävar och kaniner. Borde vara mer abstrakt.





Superklassen Animal

- Gemensamma fält i `Rabbit` och `Fox` deklarerars i `Animal`:
 - `age`, `alive`, `location`
- Vi ändrar namn på metoder för att kapsla in information:
 - `run` och `hunt` blir `act`.
- `Simulator`-klassen blir nu mindre sammankopplad än tidigare.

Modifiering av iterationen

```
for(Iterator iter = animals.iterator();  
    iter.hasNext(); ) {  
    Animal animal = (Animal)iter.next();  
    animal.act(field, updatedField, newAnimals);  
}
```

**Simulatorklassen hanterar nu
'Animal' i stället för 'Fox' och 'Rabbit'**

Metoden act i Animal

- För att den statiska typkontrollen skall fungera behövs en act-metod i Animal.
- Men - om man tittar på koden i hunt och run så finns det inte någon gemensam kod som bör skrivas i act.
- Vi definierar act som abstract:

```
abstract public void act(Field currentField,  
                          Field updatedField,  
                          List newAnimals);
```

Abstrakta klasser och metoder

- Abstrakta metoder skall deklarerars `abstract`.
- Abstrakta metoder har inget kodblock.
- En abstrakt metod gör att klassen blir abstrakt.
- Abstrakta klasser kan inte instansieras.
- Konkreta subklasser behövs för att deras implementering skall bli fullständig.

Abstrakta klasser

Användning av abstrakta klasser är ett designbeslut.

- Vi samlar gemensamma egenskaper i en klass som inte kan instansieras. I stället måste subklasser implementera de abstrakta metoderna.

Abstrakta klasser och metoder

Vi har:

```
abstract class Cow
{
    ...
}
```

Fråga: Kan vi i en annan klass skriva:

```
Cow majGull = new Cow("MajGull");
```

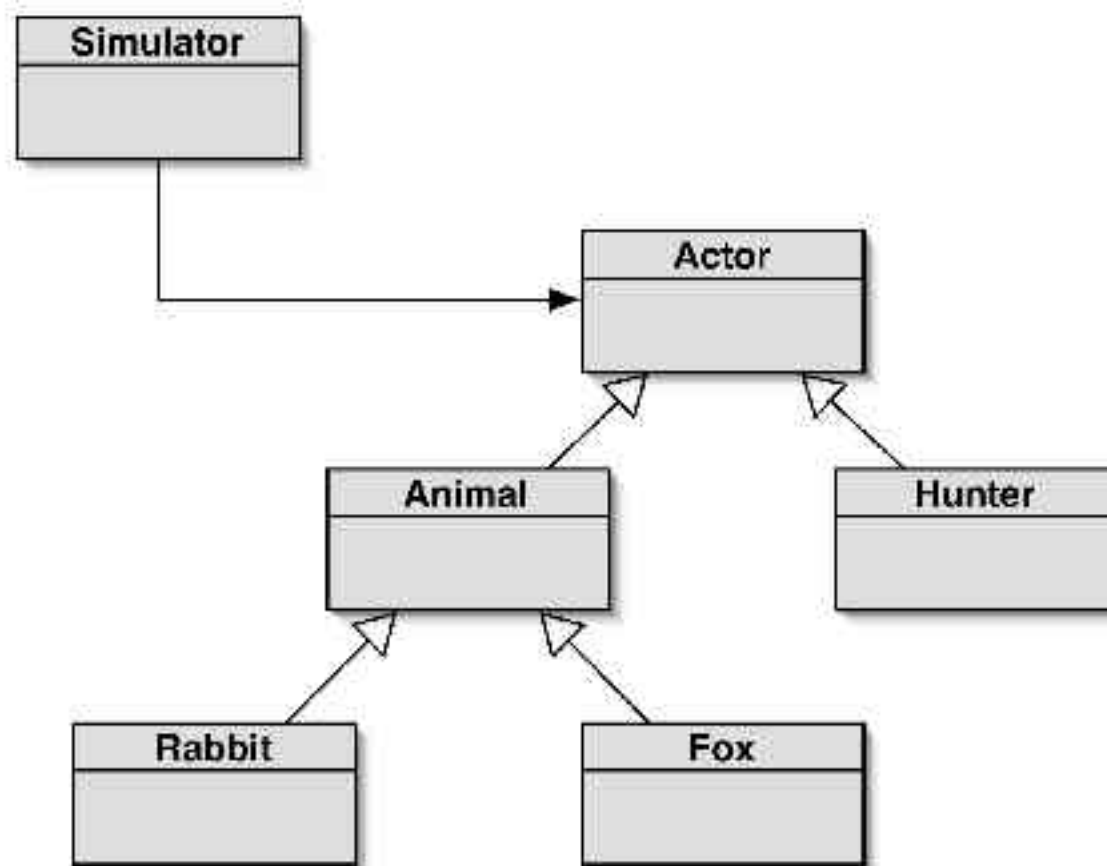
Klassen Animal

```
public abstract class Animal
{
    fields omitted

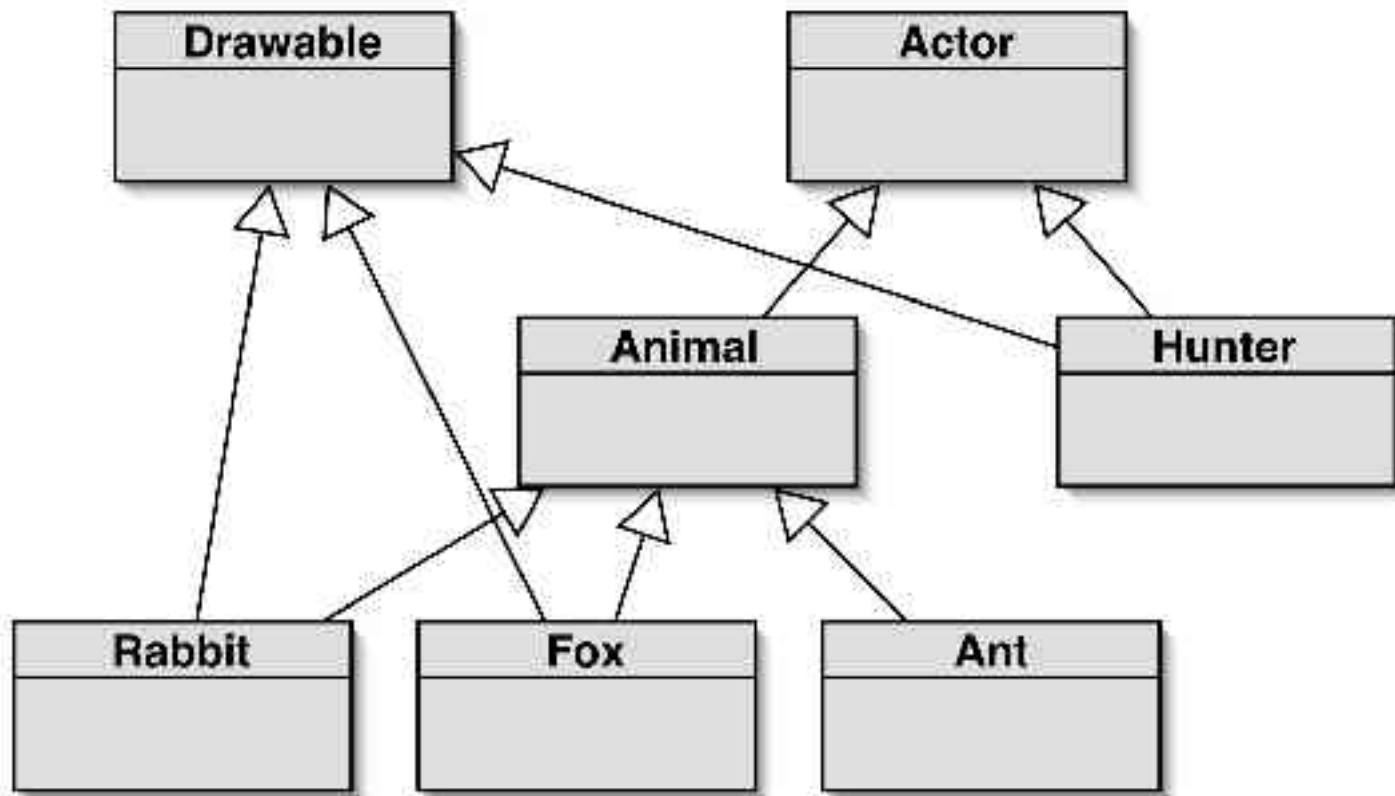
    /**
     * Make this animal act - that is: make it do
     * whatever it wants/needs to do.
     */
    abstract public void act(Field currentField,
                             Field updatedField,
                             List newAnimals);

    other methods omitted
}
```

Mer abstraktion



Vi vill kunna rita djuren (multipelt arv?)



Multipelt arv

- Innebär att en klass ärver från flera superklasser.
- Varje OOP-språk har sina egna regler.
 - Kraftfullt - men innebär tekniska och tolkningsmässiga problem.
- Java förbjuder multipelt arv för klasser.
- Java tillåter multipelt arv för gränssnitt (interfaces).
 - Innebär att vi har en kompromiss av reglerna för multipelt arv som undviker problemen.

Gränssnittet för Actor

```
public interface Actor
{
    /**
     * Perform the actor's daily behavior.
     * Transfer the actor to updatedField if it is
     * to participate in further steps of the simulation.
     * @param currentField The current state of the field.
     * @param location The actor's location in the field.
     * @param updatedField The updated state of the field.
     */
    void act(Field currentField, Location location,
             Field updatedField);
}
```

Klasser implementerar ett (eller flera) gränssnitt

```
public class Fox extends Animal implements Drawable
{
    ...
}
```

```
public class Hunter implements Actor, Drawable
{
    ...
}
```



Ett gränssnitt definierar också en typ

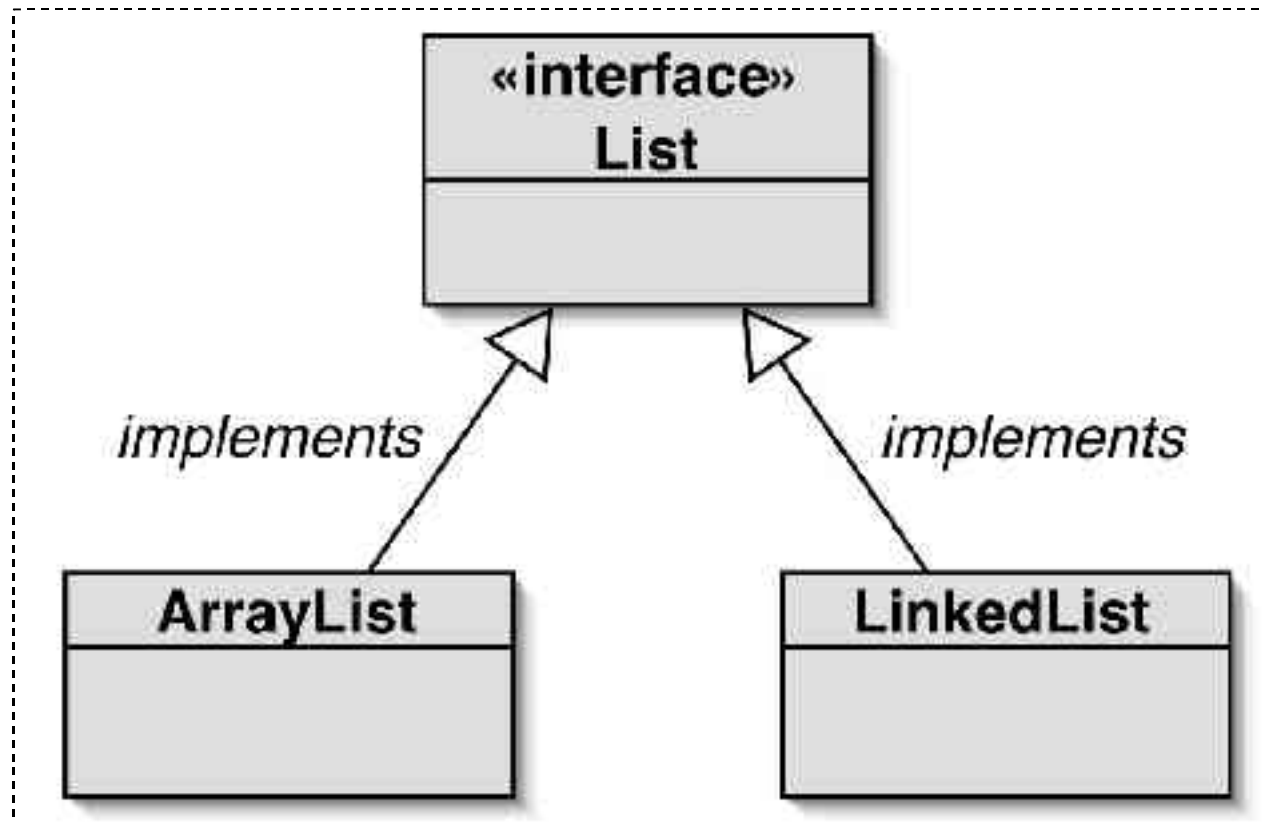
- Implementerande klasser ärver inte någon kod, men ...
- ... implementerande klasser blir subtyper till det gränssnitt de implementerar.
- Alltså, polymorfism fungerar med gränssnitt liksom med klasser.



Ett gränssnitt fungerar också som en specifikation

- Separerar funktionaliteten från implementationen.
 - Det som bestäms av gränssnittet är: Namn, parametrar och returtyp.
- Innebär att klienter kan använda ett gränssnitt oberoende av vilken klass som implementerat det.
 - Innebär att en klient (en klass som använder ett gränssnitt) kan välja mellan olika implementeringar.

Alternativa implementeringar



Summaring

- Arv i en super- subklass hierarki innebär att en implementation på en nivå kan användas på en lägre nivå
 - Abstrakta och 'konkreta' klasser.
- Arv innebär också att vi får hierarkier av super- och subtyper.

Abstrakta metoder/klasser

- Abstrakta metoder tillåter kontroll av statiska typer utan att något behöver implementeras.
- Abstrakta klasser ger stöd för polymorfism:
 - En abstrakt klass är supertyp för sina subklasser.
- Abstrakta klasser kan uppfattas som ofullständiga superklasser.
 - Tillåter inte instanser.
 - Tvingar subklasser att implementera de abstrakta metoderna.



Gränssnitt (interface)

- Ett gränssnitt ger en specifikation utan implementation.
 - Gränssnitt är helt abstrakta.
- Java tillåter multipelt arv när det gäller gränssnitt.
- Gränssnitt ger stöd för polymorfism:
 - Gränssnittets typ är supertyp för alla klasser som implementerar det.

