Repitisjonskurs

Arv, Subklasser og Grensesnitt

Subklasser

- Klasser i OO-programmering representerer typer av objekter som deler et sett med egenskaper.
- En subklasse har egenskapene til en klasse + ett sett med ekstra, mer spesialiserte, egenskaper.
- Vi sier at subklassen **arver** egenskapene til superklassen.
- Nøkkelord: **extends**

```
class Bil {
    ...
}
```

```
class Personbil extends Bil {
    ...
}
```

Subklasser

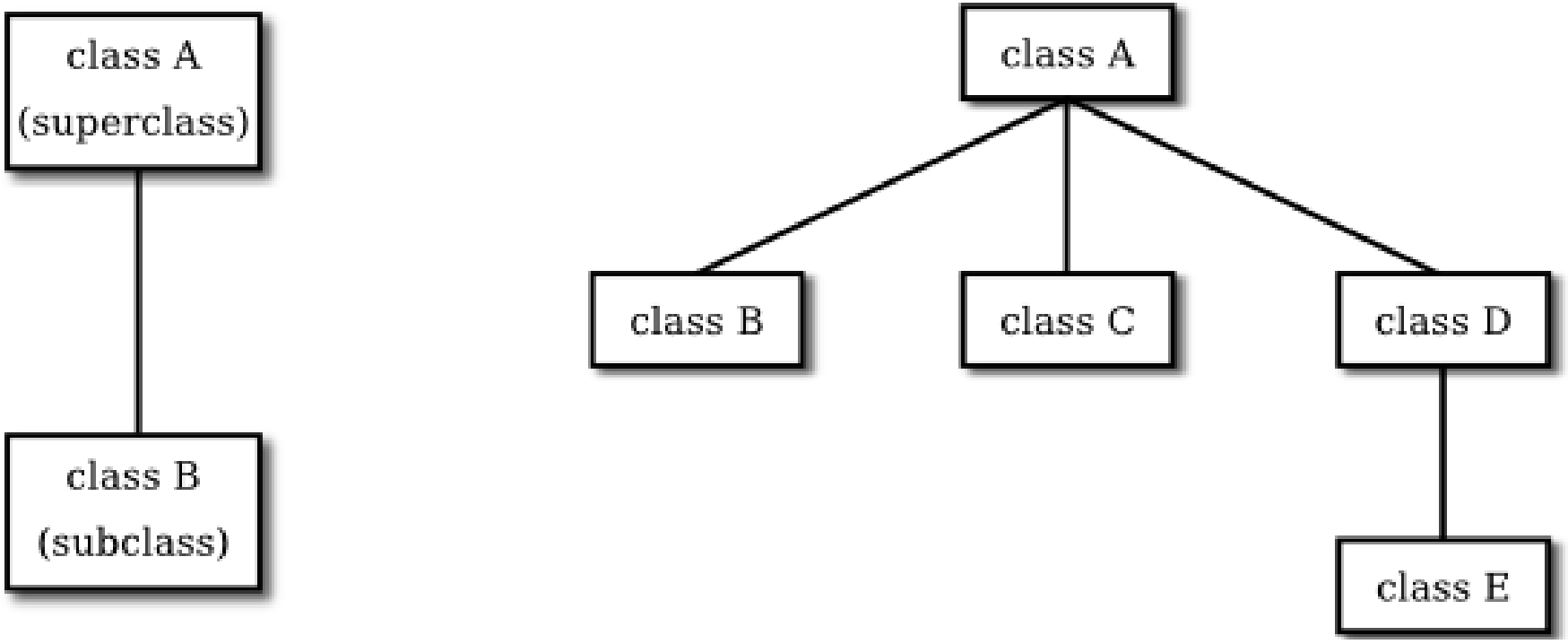
- Subklasser og superklasser har en «er en» relasjon til hverandre.
- En personbil «er en» bil, da er det naturlig å opprette disse som subklasse og superklasse.
- Vi samler det som er generellt i superklassen (egenskaper alle biler har) og det som er mer spesialisert i subklassen.
- Vi kan da bruke egenskapene til superklassen i subklassen som om alt var definert i samme klasse.

```
class Bil {
    protected String regnr;
    ...
}
```

```
class Personbil extends Bil {
    protected int antallSeter;
    ...
}
```

Subklasser og superklasser

En klasse kan ha mange subklasser i nivået under seg, men kun en superklasse i nivået over.



Protected

- Nøkkelord: protected
- protected gjør at variable er tilgjengelige i subklassene (og i pakken)
- Kompromiss mellom private og public.

```
class Bil {
    protected String regnr;
    ...
}
```

```
class Personbil extends Bil {
    protected int antallSeter;
    ...
}
```

Klassen Object

- Alle klasser har minst én superklasse: Object
- Object er «alle klassers mor», når du lager en ny klasse i Java så blir denne automatisk satt til å extende Object.
- Her finnes bl.a. metodene toString() og equals(). Metodene i Objekt finnes i alle klasser! Det er derfor vi alltid kan kalle toString på et objekt selv om vi ikke har implementert denne metoden enda.
- Ofte opp til oss å programmere disse metodene til å gjøre mer fornuftige ting.

```
class Bil {
    ...
}
```

```
class Personbil extends Bil {
    ...
}
```

Hvorfor subklasser?

- Vi forsøker ofte å avspeile virkeligheten i programmene våre ved å modellere virkelige objekter.
- Klassehierarkiet gir struktur til systemet vårt slik at det blir mer oversiktlig.
- Gjenbruk av klasser og metoder sparer oss for arbeid.

Referanser og Pekere

- En referanse (eller peker) er ikke et objekt men kan peke på et objekt.
- I java må vi definere hva som skal pekes på.
- Det er pekeren vi bruker som bestemmer hvordan vi ser (har tilgang til) objektet vi peker på, og dermed bestemmer hvilke metoder og variabler vi har tilgang til.
- Unntaket er når man overskriver virtuelle metoder (polymorfi, mer senere).

Referanser, forts

- Vi kan peke på en personbil med en Bil-peker

  Personbil bilA = new Personbil();

  Bil bilB = bilA;

- Dette går fordi en Personbil har alle egenskaper til en Bil
- Vi kan ikke peke på en Bil med en Personbil-peker:

  Bil bilC = new Bil();

  Personbil bilD = bilC;

- Dette gir feil fordi en Bil ikke nødvendigvis har alle egenskaper til å kunne refereres til som en Personbil.

```
class Bil {
    protected String regnr;
    ...
}
```

```
class Personbil extends Bil {
    protected int antallSeter;
    ...
}
```

Referanser, forts

- Et object forblir samme type (uforanderlig) etter at det er blitt opprettet men måten vi ser på det (briller) forandrer seg med pekerne vi bruker.
- Vi kan fritt endre pekeren så lenge vi bevger objektet oppover i klassehierarkiet; alle objekter kan pekes på av sin superklasse-peker.
- Hvis vi skal bevege et objekt nedover i hierarkiet må vi foreta oss noe

```
Bil bilA = new Personbil();
Personbil bilB = (Personbil) bilA;
```

- Dette kalles class-casting og vil kompilere men vi risikerer å få kjøretidsfeil hvis bilA ikke faktisk er en Personbil.
- For å unngå dette bør vi sikre oss at objektet vi refererer til er av riktig type.
- Nøkkelord: instanceof (boolean)

```
if (bilA instanceof Personbil){
    Personbil bilB = (Personbil) bilA;
}
```

Polymorfi

- En bedre måte å finne ut hva et objekt er er gjennom polymorfi.
- I java er alle metoder som ikke er **final** det vi kaller virtuelle metoder.
- Det betyr at vi kan overskrive metodene i subklassene og at det alltid vil være den dypeste metoden (i den nederste subklassen som har den) som kjøres.
- Et eksempel på dette er når vi skriver våre egne toString() metoder.
- Når vi overskiver metoder markerer vi dem med @Override

Polymorfi, forts

- Vi overskriver metoden i Bil ved å bruke den samme metode-signaturen i subklassen.
- Vi får tilgang til subklassens metode selv om vi bruker en superklasse peker.

```
BilA = new Bil();
BilB = new Personbil();
System.out.println(”En ” + BilA.type() + ” og en ” + BilB.type())
```

- Dette gir utskriften «En bil og en personbil»
- Selv om metoden har samme signatur kan innholdet være veldig forskjellig.

```
class Bil {
    protected String regnr;

    public String type(){
        return ”bil”
    }
    ...
}
```

```
class Personbil extends Bil {
    protected int antallSeter;

    @Override
    public String type(){
        return ”personbil”
    }
    ...
}
```

Konstruktører

- Nøkkelord: super
- Når vi kaller en konstruktør i en subklasse så må denne kalle sin superklasses konstruktør. Dette gjøres ved et kall på super().
- Hvis vi ikke gjør dette kallet eksplisitt så gjør Java det for oss.
- super() må inneholde de nødvendige parametre for at superklassens konstruktør skal virke.
- Hvis vi ønsker tilgang til superklasse-metoder som er overskrevet i klassen vi er i kan vi kalle f.eks; super.type()
- IKKE redeklarer varible som allerede er definert i superklassen!

```
class Bil {
        protected String regnr;

        public Bil(String rnr){
                regnr = rnr
        }

        public String type(){
                return ''bil''
        }
        ...
}
```

```
class Personbil extends Bil {
        protected int antallSeter;

        public Personbil(String rnr, int seter){
                super(rnr);
                AntallSeter = seter;
        }

        @Override
        public String type(){
                return ''personbil'';
        }
}
```

Abstrakte klasser og metoder

- Nøkkelord: abstract
- Noen ganger ser vi at det er klasser som vi trenger i hierarkiet vårt som vi aldri (burde) lager faktiske objekter av. Da er det hensiktsmessig å deklarere disse som abstrakte.
- I eksempelet vårt kan kanskje Bil gjøres abstrakt siden alle biler er en eller annen type.
- Dette betyr at vi aldri kan si f.eks new Bil(''ABC'')
- Abstrakte klasser kan ha abstrakte metoder (uten innhold).
- Da sikrer vi at hver subklasse må skrive sin egen versjon av denne metoden og den kan kalles på Bil-nivå.
- Abstrakte klasser kan altså fortsatt brukes som pekere.

```
abstract class Bil {
	protected String regnr;

	public Bil(String rnr){
		regnr = rnr
	}

	public abstract String type();
}
```

```
class Personbil extends Bil {
	protected int antallSeter;

	public Personbil(String rnr, int seter){
		super(rnr);
		AntallSeter = seter;
	}

	@Override
	public String type(){
		return ''personbil'';
	}
}
```

Interface

- I java kan en klasse kun arve fra én annen klasse.
- Noen ganger så har en klasse også egenskaper som den deler med andre klasser som ikke er dens superklasse
- ’Multippel arv’ kan oppnås med grensesnitt/interface
- Eksempler dere kjenner; Liste, Comparable, Iterable
- Grensesnitt sikrer at et objekt har visse egenskaper, uavhengig av klassehierarkiet det tilhører.

```
interface SlipperCO2 {

    int cO2Utslipp();
}
```

```
class SpesifikkBil extends Personbil implements SlipperCO2 {
    int cO2;

    public SpesifikkBil(String rnr, int seter, int cO2){
        super(rnr, seter);
        this.cO2 = cO2;
    }

    public int cO2Utslipp(){
        return cO2;
    }
}
```

Interface, forts

- Et interface inneholder egenskaper som ikke naturlig hører hjemme i et arve-hierarki, og som mange forskjellige objekter kan ha.
- Man kan arve fra mange grensesnitt
- Inteface har kun abstrakte metoder, klassene som implementerer det må selv fylle inn disse metodene (og deklarere variabler)
- Et interface ligner en abstrakt klasse, uten variabler. (bedre kjent som Abstract DataType).
- Nøkkelord: interface, implements

```
interface SlipperCO2 {

    int cO2Utslipp();
}
```

```
class SpesifikkBil extends Personbil implements SlipperCO2 {
    int cO2;

    public SpesifikkBil(String rnr, int seter, int cO2){
        super(rnr, seter);
        this.cO2 = cO2;
    }

    public int cO2Utslipp(){
        return cO2;
    }
}
```

Interface, forts

- Interfaces kan også brukes som referanser/pekere;

  SlipperCO2 utslipperA = new SpesifikkBil(”ABC”, 3, 50000);
- Da har man kun tilgang til metodene som er definert i interfacet.
- Dette gjør at man kan behandle objekter av helt forskjellige typer som «like» under de riktige omstendighetene

```
interface SlipperCO2 {

    int cO2Utslipp();
}
```

```
class SpesifikkBil extends Personbil implements SlipperCO2 {
    int cO2;

    public SpesifikkBil(String rnr, int seter, int cO2){
        super(rnr, seter);
        this.cO2 = cO2;
    }

    public int cO2Utslipp(){
        return cO2;
    }
}
```

Hva kan peke på hva?

- A a = new A();
- a = new B();
- I i = (I) a;
- G g = (G) a;
- A a2 = a;
- C c = new D();
- g = (G) c;
- D d = new A();
- d = (D) c;
- d = c;

```
interface I {};
interface G {};

class A
class B extends A implements I
class C extends A
class D extends C implements G
```

Hva kan peke på hva?

- A a = new A(); Riktig
- a = new B(); Riktig
- I i = (I) a; Riktig
- G g = (G) a; Feil
- A a2 = a; Riktig
- C c = new D(); Riktig
- g = (G) c; Riktig
- D d = new A(); Feil
- d = (D) c; Riktig
- d = c; Feil

```
interface I {};
interface G {};

class A
class B extends A implements I
class C extends A
class D extends C implements G
```