Recap!

{ - En kort oppsummering av det siste vi har lært

Interface

- Vi vil ha en generell oppskrift på hvordan noe (en klassen) skal fungere
- **Interface List:** Alle klasser som implementerer denne skal være en ordnet sekvens av objekter.
- **Interace Iterable:** Alle klasser som implementerer denne skal kunne itereres over med en for(Foo a : Bar){} løkke
- Når man lager et IT-system bruker man stort sett alltid samlinger av objekter (liste av personer (Facebook), liste av sanger(Spotify))
- Man ønsker ikke implementere slike "samlinger" selv. Da bruker man noe som allerede er implementert.
- Hvordan vet man hvordan en gitt "samling" fungerer?
- **Interfacet** er det som definerer oppførselen (metodene og deres dokumentasjon).

Interface

- Men man kan ikke instansiere et interface:

```
List minListe = new List();  // Ikke lov
```

- Hva gjør man da?

```
List myList = new LinkedList();
```

- Eller

```
List myList = new ArrayList();
```

- Regel 1: En klassen kan implementere så mange interfaces den vil:

```
class A implementer B, C, D {}
```

- Regel 2: Klassen A må implementere ALLE metodene i B, C og D.

Generics

Vi ønsker å ha en generell beholder!

Problem:

```
List myList = new LinkedList();
myList.add( new Student("Magnus") );

Professor prof = (Professor) myList.getFirst();
```

Dette gir kjøre feil!! Alt tryner!!

Generics

Løsning: Generics!

```
List <Student> myList = new LinkedList<Student>();
myList.add( new Student("Magnus") );

Professor prof = myList.getFirst();
```

Dette gir kompilator feil! Sweet, lett å fikse før systemet blir satt i produksjon :)

Vi finner feilen fort, og retter opp:

```
Student stud = myList.getFirst();
```

Subklasser

Mange klasser er like. Vi ønsker ikke skrive lik kode flere ganger:

```
class Rektangel{
    public int getAreal(){
            return høyde * bredde;
    }
}

class Kvadrat{
    public int getAreal() {
            return høyde * bredde;
    }
}
```

Unødvendig! Bedre løsning med subklasser ->

Subklasser

```
class Firkant(){
    public int getAreal(){
            return høyde * bredde;
    }
}
class Rektangel extends Firkant // inger duplisering av kode
class Kvadrat extends Firkant

Firkant kv = new Kvadrat(10,10);
System.out.println(kv.getAreal());
```

Regel: En klasse kan bare extende 1 klasse

```
class A extends B{} // Og ingen fler..
```

Men vi kan kombinere:

```
class A extends B implements C, D, E {}
```

Polymorfi:

```
class Firkant{
      public int getOmkrets(){
            return høyde*2 + lengde*2;
       }
}
class Kvadrat extends Firkant{

       public int getOmkrets(){     // Overrider superklassen!!
            return høyde*4;
       }
}

Firkant kv = new Kvadrat(10,10);
System.out.println(kv.getOmkrets()); // Bruker den "nederste"
klassen sin versjon av getOmkrets()
```

Abstrakte klasser:

Noen ganger hadde det vært fint om et interface kunne inneholdt implementasjon av enkelte metoder:

```
Interface Figur {
        public double areal();
        public double omkrets();

        public double halvOmkrets(){
                return omkrets() / 2;
        }
}
```

Dette er ikke lov! Løsning –>

Abstrakte klasser:

Lag en abstrakt klasse!

```
public abstract class Figur {

    abstract public double areal();
    abstract public double omkrets();

    public double halvOmkrets(){
            return omkrets() / 2;
    }
}
```

Regel 1: En abstrakt metode har ingen implementasjon, men bare et semikolon på slutten

Abstrakte klasser:

Regel 2: Alle klasser som extender en abstrakt klasse må selv implementere alle de abstrakte metodene, eller deklarere metodene som abstrakte.

```
abstract class Stor{
        abstract void metode();
}
abstract class Mellom extends Stor{
        abstract void metode(); // Mellom må enten implementere metoden, eller kalle den
                                // abstract
}

class Liten extends Mellom {
        void metode(){
        System.out.println("Foo!");
        }
}
```

Regel 3: Hvis en klasse skal ha en abstrakt metode må klassen selv være abstract (Stå abstract foran class deklarasjonen).