# Lite repetition:

# Abstract class

- En abstract class är en class som inte kan skapa några objekt.
  - Syfte: Att samla gemensam kod för subclasses. *(Undvika kod-duplicering)*
  - Kan *(men måste inte)* ha metoder märkta abstract.
    - Dessa har ingen body och subclasses måste själva definera metoden *(om de inte själva är abstract).*
  - Kan ha konstruktorer; dessa kan enbart anropas via `this()` i den egna klassen eller `super()` i subclasses.

# Interface

- Ett interface är en specifikation av ett antal metod-signaturer som tillsammans bildar en typ.
  - Enbart signaturerna ges – alla metoder är helt abstrakta.

- Interfaces i Java får specificera konstanter - attribut som implicit är `public static final`, och som initialiseras till ett konstant värde.
  - Används sällan.
    Grundprincipen för ett interface är en samling metodsignaturer.

# Abstract class vs interface

Använd det som passar avsikten bäst. *Tumregel:*

- En abstract class representerar en generalisering av flera olika subclasses.
  - `Mammal` generaliserar `Cow, Sheep, Cat`
  - `Shape` generaliserar `Rectangle, Triangle, Circle`

- Ett interface representerar en egenskap som kan delas av många (helt) olika classes.
  - `Clonable` representerar alla objekt som kan skapas exakta kopior av
  - `Runnable` representerar alla objekt som kan köras som en separat thread
  - `Moveable` representerar alla objekt som kan förflytta sig

# OCP: The Open-Closed Principle

> Software modules should be open for extension, but closed for modification.

- Vad fyller principen för syfte?
  - Att skriva kod där ny funktionalitet enkelt kan läggas till (open for extension) samtidigt som så mycket som möjligt av existerande kod kan lämnas orörd (closed for modification).

- Varför är det bra att följa den?
  - Koden blir maintainable och extendable.

# Override *(överskuggning)*

- Överskuggning av metoder är en funktion som tillåter en subklass att tillhandahålla en specifik implementering av en metod som redan tillhandahålls av en av dess superklasser.
- Override vs Abstract:
  - Abstract används när alla subklasser kommer ha olika beteenden.
  - Override används när de flesta subklasser har samma beteende, men ett fåtal skiljer sig.

  - Abstract är obligatoriskt.
  - Override är valfritt.

  - Att använda Abstract är bättre med avseende på OCP.
  - Oftast är det bäst att designa sin kod så att man slipper överskugga allt för många metoder då det inte ger en lika tydlig bild av vad som händer när man anropar en överskuggad metod.

# Lite nytt inför nästa uppgift:

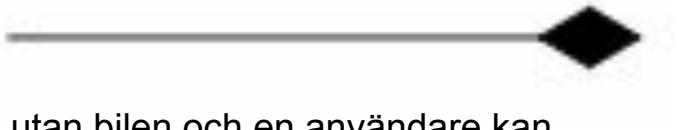
# Komposition

- **Arv** är när man designar sina klasser med tanke på vad de **<u>är</u>**.
- **Komposition** är när man designar sina klasser runt vad de **<u>gör</u>**.

*Två typer av Komposition:*

- Aggregation *(Has-A)*
  - "Has-A"-relation
  - Kan existera oberoende av varandra.
  - Exempel: En skola har elever, båda kan existera oberoende av varan.

- Composotion *(Starkt Has-A / Part-Of)*
  - Kan enbart existera tillsammans med en annan klass.
  - "Gömmer" funktionalitet för användaren.
  - Exempel: En bil har en motor, motorn kan inte existera utan bilen och en användare kan köra bilen utan att veta om motorn. *(OBS! Beror på hur man designar)*




Composition
over
Inheritance

# Delegering

- Använd ett annat objekt för att utföra en eller flera uppgifter.
- Det objekt som tar över upp-giften kallas för delegerare.
- En typ av composition.
- En delegerare ska inte ändra befintlig funktionalitet för klassen som den “hjälper”.
- Delegeraren är “gömd” i klassen som den hjälper.

- Exempel: En timmerlastbil har en lyftkran längst bak som hjälper till att lasta timmer på flaket. Men den ändrar inte själva lastbilens beteende, bara utökar.
- *En motor “hjälper” bilen att köra, men är ingen delegerare då den påverkar själva bilens beteende och krävs för att bilen ska fungera.*

# Vad är vad?

```
public class A {

    private B b = new B();

    public void methodA() {
        b.methodB();
    }

}
```

Delegering

```
public class A {

    private B b;

    public A(B b) {
        this.b = b;
    }
}

class C {

    public C() {
        B b = new B();
        A a = new A(b);
    }
}
```

Aggregation

```
public class A {

    private B b = new B();

    public A() {

    }

}
```

Komposition