Objektorienterad programmering (OOP) i Java

Tema: Från procedural kod till en objektorienterad lösning

Case: Salbokningssystemet

Tid: 25 minuter

Aman Arabzadeh

2026-01-21

Högskolan Kristianstad

Lärandemål

Efter föreläsningen ska du kunna:

1. Förklara varför dubbelbokning lätt uppstår i procedural kod – och hur OOP förhindrar det

2. Beskriva inkapsling och abstraktion, och varför de är viktiga

3. Känna igen när arv används och förstå syftet med det

4. Förklara polymorfism med ett enkelt exempel

Dessa mål fokuserar på OOP:s kärnprinciper för att bygga robusta och underhållbara system.

- **Idag:** Vi använder salbokningssystemet för att se nyttan direkt
  - Fokus: **abstraktion** och **inkapsling**
    Arv och polymorfism: introduceras kort, mer i kommande lektioner

Vad är OOP?

- OOP = modellera verkligheten med **objekt**
- Objekt = **data + metoder**
- Klass = **mall/ritning** (t.ex. `LaroSal`)
- Objekt = **instans** (t.ex. `E228`, `E330`)
- Varför OOP: **data + regler ihop** → mer robust, färre fel

Klass: Lärosal

E228

E330

C200

Problemet med procedural kod i salbokning

Tillståndet är öppet i `main`
→ lösa variabler: `salNummer`, `bokadAv`, `bokadTid`

Inget skydd mot felaktiga ändringar
→ man kan skriva över bokningar utan varning

Regler sprids och kopieras
→ konfliktkontroll måste göras på flera ställen
→ svårt att underhålla och skala
→ lätt att missa → **dubbelbokning**

Det här är ett klassiskt spagettikod-problem: data här, regler där, och massa if-satser som försöker hålla ihop allt. Ju mer vi bygger på, desto mer trasslar det.

OOP-lösningen: Data + Regler i samma objekt

LäroSal äger sitt eget tillstånd

→ salNummer, bokadAv, bokadTid (private!)

Regeln bor i metoden boka(naman, bokadTid)

→ Redan bokad? → return false

→ Ingen dubbelbokning möjligt!

main blir enklare och säkrare

→ Vi frågar objektet istället för att pilla i variabler

Så: data + regler i samma objekt. `main` kan inte fuska. Därför kan dubbelbokning inte smyga in.

LaroSal
- salNummer: String - bokadTid String - bokadAv: String
+ LaroSal(salNummer: String) + boka(person: String, tid: String): boolean + status(): String + getBokadAv(): String

- betyder private (inte ändras utifrån)

+ betyder public (det som får användas)

De fyra grundpelarna i OOP

- **Abstraktion**
  - *Vad* vi gör – inte *hur* det görs internt
  - Enkelt gränssnitt som döljer komplexitet
- **Inkapsling**
  - Skyddar objektets data med regler
  - `private` fält + kontrollerade metoder
- **Arv (“är-en”-relation)**
  - Återanvänd gemensam kod, minska upprepning
  - Ex: `Person` → `Student` / `Lärare`
- **Polymorfism**
  - Samma metodanrop – olika beteende
  - JVM väljer rätt version automatiskt

Abstraktion: Göm detaljer – visa det viktiga

- Användarkoden behöver bara veta:

  → boka(...)

  → status()

- Den behöver INTE veta:

  → Hur bokningar lagras internt

  → Hur validering sker

  → Hur tid hanteras

Fördel: Vi kan ändra insidan senare utan att bryta resten av programmet!

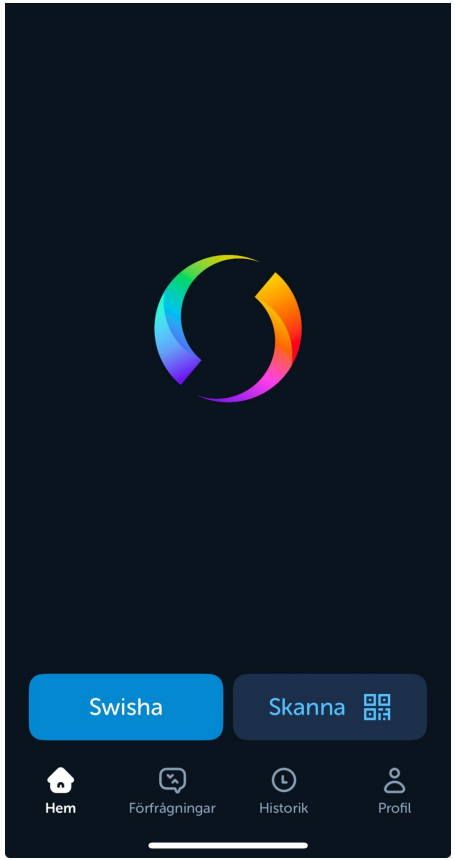
Analogi: Tänk på Swish!

Du behöver bara:

- Skanna QR-koden
- Ange belopp & skicka

Du behöver INTE veta:

- Hur pengarna flyttas mellan banker
- Vilka säkerhetskontroller som sker
- Vilka servrar som är inblandade



Inkapsling = Skydda objektets tillstånd och integritet

- Gör fält private (tillståndet är inte öppet utifrån)
- Ändra tillstånd via metoder där regler kontrolleras
  - t.ex. boka(...), status(), getBokadAv()
- Poängen: Ändra state via metoder – inte genom att skriva direkt i fälten
- Skyddar viktiga regler: Ingen dubbelbokning möjligt!

```
// FEL (ska inte gå)
sal.bokadTid = "2026-01-21 10:00";

// RÄTT
sal.boka("Sara", "2026-01-21 10:00");
```

Arv: Återanvänd gemensam kod

"är-en"-relation: Student och Lärare är båda Personer

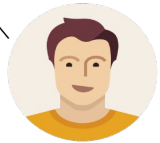
Fördelar:
→ Allt gemensamt (namn, e-post, adress, kontaktInfo()) skrivs bara EN GÅNG
→ Mindre kod, lättare att underhålla, färre fel

Person (basklass)

Typ	Namn	Beskrivning
String	**namn**	Förnamn och efternamn
String	**extraNamn**	Mellannamn eller smeknamn
String	**ePost**	E-postadress
String	**adress**	Postadress
Metod	**kontaktinfo()**	Returnerar kontaktuppgifter

Student (subklass)

Typ	Namn	Beskrivning
String	**studentID**	Unikt ID för studier
String	**program**	Vilket program man läser
int	**termin**	Nuvarande termin



Lärare (subklass)

Typ	Namn	Beskrivning
String	**anställningsNr**	Unikt anställningsnummer
String	**institution**	Institution eller avdelning
String	**titel**	T.ex. Lektor eller Professor

Polymorfism: Samma anrop – olika beteende

Polymorfism ger oss:

- Samma kod för alla Person-objekt
- Mycket flexibel och utbyggbar design
- JVM väljer rätt version automatiskt!

```java
public class MedPolymorfism {
    public static void main(String[] args) {
        Person[] personer = {
                new Student("Anna", "anna@uni.se", "Gatan 1", "s123", "CS"),
                new Larare("Ola", "ola@uni.se", "Gatan 2", "a777", "Lektor"),
                new Person("Erik", "erik@exempel.se", "Storgatan 10")
        };

        for (Person p : personer) {
            System.out.println(p.kontaktInfo());
            System.out.println("----------");
        }
    }
}
```

```java
public class UtanPolymorfism {
    public static void main(String[] args) {
        Person[] personer = {
                new Student("Anna", "anna@uni.se", "Gatan 1", "s123", "CS"),
                new Larare("Ola", "ola@uni.se", "Gatan 2", "a777", "Lektor"),
                new Person("Erik", "erik@exempel.se", "Storgatan 10")
        };

        for (Person p : personer) {
            if (p instanceof Larare l) {
                System.out.println(l.kontaktInfoLarare());
            } else if (p instanceof Student s) {
                System.out.println(s.kontaktInfoStudent());
            } else {
                System.out.println(p.kontaktInfo());
            }
            System.out.println("----------");
        }
    }
}
```

1. Ny typ (t.ex. Gäst)? Med polymorfism: Lägg bara till ny klass – loopen funkar!
2. Utan: Ändra alla if-satser.

Take-away – Vad tar vi med oss?

Procedural kod:

- Regler hamnar utspridda → lätt att missa
- Lätt att skriva över tillstånd (dubbelbokning!)

OOP – lösningen (de 4 grundpelarna):

- **Inkapsling:** private fält + kontrollerade metoder → svårare att kringgå regler
- **Abstraktion:** använd objekt via få metoder (boka(), status()) → insidan kan ändras fritt
- **Arv** (bonus idag): återanvänd gemensam kod (Person → Student/Lärare)
- **Polymorfism** (bonus idag): samma anrop → olika beteende (kontaktInfo() via Person)

Fördjupning – rekommenderad läsning:

- Oracle Java Tutorials – Classes and Objects (https://docs.oracle.com/javase/tutorial/java/javaOO/)
- Bloch, J. – Effective Java (3:e upplagan – kapitel om OOP)
- Martin, R. C. – Clean Code (kapitel om objekt och inkapsling)
- W3Schools – Java OOP (snabb repetition)

Tack för er tid!

Skanna mig! =)

All kod, slides och exempel finns publikt här:

https://github.com/aman-arabzadeh/OOP-F-rel-sning

Kopiera gärna repot och testa exemplen själva!

Frågor?

Jag svarar gärna!

Aman Arabzadeh

Högskolan Kristianstad

2026-01-21

Mini-quiz (10 sekunder) – Testa er själva!

1) **Varför blir det lätt dubbelbokning i procedural kod, men inte i vår OOP-version med LaroSal?**

   ✅ → I procedural kod är data öppen och reglerna ligger utspridda, man kan råka skriva över en bokning utan kontroll. I OOP är tillståndet **private** och allt måste gå via boka(...) där regeln alltid kollar om salen är ledig, dubbelbokning kan inte smyga in.

2) **Vad är poängen med abstraktion i OOP?**

   ✅ → Användaren behöver bara veta **vad** man kan göra (boka() och status()), inte **hur** det fungerar internt. Då kan vi ändra lagring, validering eller tidshantering senare, utan att bryta koden som använder klassen.

3) **Varför är inkapsling viktigt i vårt salsbokningssystem?**

   ✅ → För att skydda objektets tillstånd. När fälten är **private** kan ingen ändra bokningar eller tider direkt, allt måste gå via metoder som boka(...) där reglerna kontrolleras. Det förhindrar att objektet hamnar i ogiltigt tillstånd, som dubbelbokning.