INDKAPSLING

Programmering 1

Lektion 6

Progammering 1 - lektion 6

...

4. Klasser og objekter
5. Arv og komposition
6. Indkapsling (I DAG)
7. Abstrakte klasser og interfaces
8. Polymorfi

...

Læringsmål - i dag

- Hvad er indkapsling
  - `public` og `private`
  - `get`ter og `set`ter metoder
- Hvorfor bruge indkapsling?
  - API
  - Refaktorering
- Øvelser i grupper - og diskussion

Repetition af klasser og objekter

Repetition af klasser

```
package dk.kea.prog1.ex1;

class SavingsAccount {
  // fields
  double balance;
  double rate = 0.05;
  String owner;

  // constructor
  SavingsAccount(double balance, String owner) {
    this.balance = balance;
    this.owner = owner;
  }

  // method
  void applyInterest() {
    balance = balance * (1 + rate);
  }
```

Repetiton af objekter

```
package dk.kea.prog1.ex1;

class Runner {
  public static void main(String[] args) {

    SavingsAccount kimsAccount = new SavingsAccount(1000, "Kim");

    kimsAccount.applyInterest();

    System.out.println(kimsAccount.balance); // => 1050
  }
}
```

Hvad er indkapsling

DEMO

Opsparingskonto

Klassen, dens felter og metoder (og constructor) har access modifiers:

- `public`
- `private`
- `protected`
- "default" (ingen access modifier)

DEMO

Lad os gøre det hele `public` og se hvad der sker.

Ny adfærd: Hæve penge fra kontoen

```
package dk.kea.prog1; // NB: Ændret pakkenavn

SavingsAccount kimsAccount = new SavingsAccount(1000, "Kim");

kimsAccount.balance = kimsAccount.balance - 1200;

System.out.println(kimsAccount.balance); // => -200
```

Hvad er problemet her?

`get`ter og `set`ter metoder:

- Konvention: `get` og `set` foran feltnavn, dvs.

```
public double getBalance() {
  return balance;
}
```

```
public void setBalance(double balance) {
  this.balance = balance;
}
```

DEMO

Lad os lave get og set metoder for `balance`, så saldoen aldrig kan være negativ.

```
class SavingsAccount {
  private double balance;
  private double rate = 0.05;
  private String owner;

  public SavingsAccount(double balance, String owner) {
    this.balance = balance;
    this.owner = owner;
  }

  [...]

  public double getBalance() {
    return this.balance;
  }

  public void setBalance(double balance) {
    if (balance < 0)
      return;

    this.balance = balance;
  }
}
```

```java
SavingsAccount kimsAccount = new SavingsAccount(1000, "Kim");

kimsAccount.setBalance(kimsAccount.getBalance() - 1200);

System.out.println(kimsAccount.getBalance()); // => 1000
```

Er der noget vi kan gøre bedre?

Øvelse (20 min): At hæve/indsætte penge.

```
class SavingsAccount {
  private double balance;
  private double rate = 0.05;
  private String owner;

  public SavingsAccount(double balance, String owner) {
    this.balance = balance;
    this.owner = owner;
  }

  public void applyInterest() {
    balance = balance * (1 + rate);
  }

  public double getBalance() {
    return balance;
  }
}
```

Diskussion af øvelse: At hæve/indsætte penge

```
class SavingsAccount {
  private double balance;
  private double rate = 0.05;
  private String owner;

  SavingsAccount(double balance, String owner) {
    this.balance = balance;
    this.owner = owner;
  }

  public void applyInterest() {
    double interest = balance * (1 + rate);
    deposit(interest);
  }

  public void withdraw(double amount) {
    if (amount > balance)
      return;

    setBalance(balance - amount);
  }

  public void deposit(double amount) {
    if (amount < 0)
      return;

    setBalance(balance + amount);
  }

  public double getBalance() {
    return balance;
  }

  private void setBalance(double balance) {
    if (balance < 0)
      return;

    this.balance = balance;
  }
}
```

```
SavingsAccount kimsAccount = new SavingsAccount(1000, "Kim");

kimsAccount.withdraw(1200);
System.out.println(kimsAccount.getBalance()); // => 1000

kimsAccount.withdraw(200);
System.out.println(kimsAccount.getBalance()); // => 800
```

Resultat

- Mere kontrol over objektets tilstand
- Mere fleksibilitet til at ændre klassens implementering
- Mere læsbart og vedligeholdeligt kode

Hvorfor bruge indkapsling?

API - Application Programming Interface

- `public` metoder er klassens API - grænsefladen til klassens brugere
- `private` metoder er klassens interne implementering - skjult for klassens brugere
- Refaktorering: at ændre koden uden at ændre klassens API

Øvelse: Refaktorering af `SavingsAccount`

- Kan vi gøre koden mere læsbar uden at ændre klassens API?

Diskussion af øvelse: Refaktorering af `SavingsAccount`

```
class SavingsAccount {
  private double balance;
  private double rate = 0.05;
  private String owner;

  SavingsAccount(double balance, String owner) {
    this.balance = balance;
    this.owner = owner;
  }

  public void applyInterest() {
    deposit(calculateYearlyInterest());
  }

  [...]

  private double calculateYearlyInterest() {
    return balance * (1 + rate);
  }
}
```

Gruppearbejde (2 time) - Beregning af rente

1. Implementer daglig rente i stedet for årlig rente
2. Lav en ny klasse `CheckingAccount` med samme funktionalitet som `SavingsAccount`, der tillader overtræk (negativ saldo).
3. Implementer strafrente på overtrækket.
4. (Extra) Hvordan kan vi dele kode mellem `SavingsAccount` og `CheckingAccount`?
5. (Extra) Hvad gør access modifier `protected`?
6. (Extra) lav en `Bank`, der har funktionalitet til at tilskrive renter til flere konti.

Diskussion af gruppearbejde

Opsamling - hvad har vi lært?

Nævn tre ting I tager med fra i dag.

Næste gang

- Hvordan lader vi `Bank` have en liste med forskellige typer konti, dvs. `CheckingAccount`, `SavingsAccount` m.fl.?
- Hvordan sikrer vi os at alle konti har metoden `applyInterest()` til at tilføje renter?

Næste gang - Abstrakte klasser og interfaces

- Hvordan kan vi bruge interfaces til at definere en kontrakt for klasser?
- Hvordan kan vi bruge abstrakte klasser til at dele kode mellem klasser?