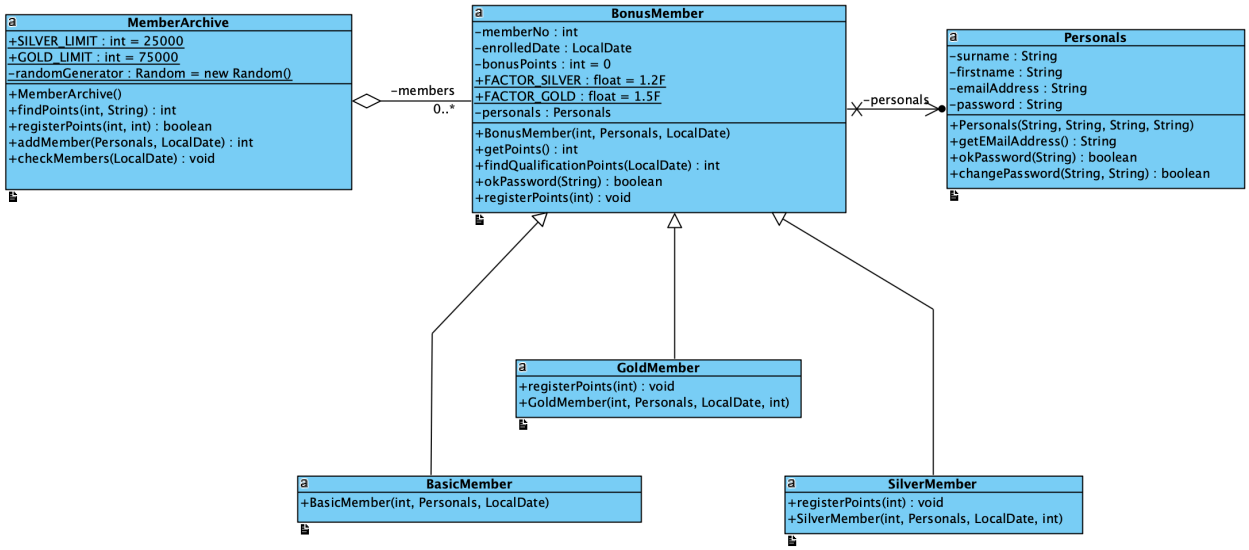
# Arv og polymorfi

## Problemstilling

Følgende klassediagram er gitt:




**Figur 1 Klassediagram for bonus ordning**

Et flyselskap tilbyr tre ulike typer bonuskort. En kunde tjener opp poeng hver gang han reiser med fly. Når en kunde melder seg inn i ordningen starter han alltid som medlem på nivå *Basic*.

Dersom kunden reiser ofte og tjener opp mange poeng det første året han er medlem i ordningen oppgraderes han automatisk til nivå *Sølv* eller *Gull*.

For å bli sølvmedlem må kunden tjene opp 25000 poeng i løpet av ett år. Kravet til gullmedlemskap er 75000 poeng. En kunde er enten basic-medlem, sølvmedlem *eller* gullmedlem.

Alle medlemmer tjener i utgangspunktet det samme for den samme reisen på samme klasse. (Og her stopper likheten mellom oppgaveteksten og velkjente bonusordninger...) Sølvmedlemmer får et påslag i antall poeng på 20%, mens gullmedlemmer får et påslag på 50%.

Eksempel: Dersom en tur gir 5.000 poeng i gevinst, skal 5.000 poeng registreres for basic-medlemmer. Sølvmedlemmer skal få 5.000 x 1.2 = 6.000 poeng, mens gullmedlemmer får 5.000 x 1.5 = 7.500 poeng.

***Forenklinger i denne oppgaven:***

- Vi lagrer ikke enkelttransaksjoner, bare poengsaldoen.
- For å kunne oppgraderes til sølv- eller gullmedlem, må kunden tjene henholdsvis 25.000 og 75.000 poeng *det første året* han/hun er medlem.

## Versjonskontroll

Dette er en ypperlig oppgave å bli kjent med versjonskontroll samtidig som du løser oppgaven. Opprett et *lokalt repository* i prosjektmappen, og legg deg til gode vaner med å *sjekke inn* (commit) endringer i koden din fortløpende. Husk gode kommentarer til commits.

## Oppgave 1

Du skal i denne oppgaven konsentrere deg om alle klassene på figur 1 *unntatt* klassen MemberArchive.

Klassen **Personals** er gitt (Se vedlegg).

Du skal programmer klassene **BonusMember**, **BasicMember**, **SilverMember** og **GoldMember**.

Klassen **BonusMember** har følgende objektvariabler/felt:

```
private final int memberNo;
private final Personals personals;
private final LocalDate enrolledDate;
private int bonuspoints = 0;
```

Klassen **LocalDate** ligger i pakken java.time (https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html).

I tillegg til feltene er det definert to *konstanter*: **FACTOR_SILVER** og **FACTOR_GOLD.** Disse kosntantene brukes for å regne om bonuspoeng som gis når du har oppnådd henholdsvis sølv eller gull medlemskap.

Konstantene har følgende verdi:

FACTOR_SILVER = 1.2

FACTOR_GOLD = 1.5

Så dersom en gullmedlem tjener 10 000 poeng, registreres 10 000 * 1.5 = 15 000 poeng.

Utfyllende forklaring til metodene i klassen BonusMember:

- De fire første metodene (**getMemberNo()**, **getPersonals()**, **getEnrolledDate()** og **getPoints()**) implementeres som vanlige get-metoder.
- **findQualificationPoints()** skal returnere antall poeng som kan kvalifisere til oppgradering av medlemskapet til sølv eller gull. Dersom innmeldingsdatoen ligger mindre enn 365 dager bak i tid i forhold til datoen som sendes inn som argument, returneres antall poeng. Hvis det er mer enn ett år siden kunden meldte seg inn, returneres 0 poeng. Du kan finne differansen mellom to objekter av klasse LocalDate på denne måten:

  ```
  long daysBetween = ChronoUnit.DAYS.between( this.enrolledDate, date );
  ```

- **okPassword()** tar et passord som argument, og returnerer *true* dersom det er ok.

- **registerPoints()** skal ta antall poeng som argument og registrere disse i henhold til reglene foran. (Du skal ikke tenke på oppgradering til gull- og sølvmedlemskap her.)

Her følger noen testdata (findQualificationPoints() i forhold til datoen 10.02.08):

| Person | Type | Startpoeng | Innmeldt dato | poeng*) | findQualificationPoints () | findPoints() | poeng | findQualificationPoints () | findPoints() |
|---|---|---|---|---|---|---|---|---|---|
| Ole | Basic | 0 | 15.02.06 | 30.000 | 0 | 30.000 | 15.000 | 0 | 45.000 |
| Tove | Basic | 0 | 05.03.07 | 30.000 | 30.000 | 30.000 | | | |
| Tove | Sølv | 30.000 | 05.01.07 | 50.000 | 90.000 | 90.000 | | | |
| Tove | Gull | 90.000 | 10.08.07 | 30.000 | 135.000 | 135.000 | | | |

*) poeng uten påslag for sølv-/gullmedlemskap

Tabellen kan du lese som følger: Ole registreres som medlem med nivå Basic. Han er innmeldt 15.02.06 med Basic medlemskap. Han får 30.000 poeng (kolonnen «poeng»). Dersom vi nå kaller *findQualificationPoints()* med datoen 10.02.08 som parameter, skal vi få 0 i retur. Kaller vi metoden *findPoints()* får vi i retur hvor mange poeng Ole har. Så legger vi til 15.000 poeng. Et kall til *findQualificationPoints()* med datoen 10.02.08 som parameter gir fortsatt 0 poeng.

Testdataene er lagt inn i en egen *testklasse* **BonusMemberTest** (Se vedlegg). Du kan bruke den ved testing - vurder om det er vesentlige ting som ikke blir testet.

## Oppgave 2

Implementer så de tre klassene **BasicMember**, **SilverMember** og **GoldMember**. Samtlige arver fra klassen **BonusMember**.

Implementer *konstruktøren* i hver av klassene.

I klassene **SilverMember** og **GoldMember** skal du *re-definere (override)* metoden *registerPoints()* slik at når denne metoden kalles, regnes poengene om i henhold til medlemsskap. Det vil si at dersom følgende kall gjøres:

```
SilverMember sm = new SilverMember(..);
sm.registerPoints(10000);
```

…så registreres 10 000 * FACTOR_SILVER = 12 000 poeng. Tilsvarende for GoldMember.

# Oppgave 3

Du skal nå programmere metodene i klassen **MemberArchive**. Følgende gjelder:

- **findPoints()** skal ta medlemsnummer og passord som argument og returnere antall poeng denne kunden har spart opp. Returner en negativ verdi hvis medlem med dette nummeret ikke fins, eller passord er ugyldig.
- **registerPoints ()** skal ta medlemsnummer og antall poeng som argument og sørge for at riktig antall poeng blir registrert for dette medlemmet. Returner false dersom medlem med dette nummeret ikke fins.
- **addMember()** skal ha følgende metodehode:

  ```
  public int addMember(Personals pers, LocalDate dateEnrolled)
  ```

  Metoden skal opprette et objekt av klassen **BasicMember** og legge dette inn i arkivet. (Alle medlemmer begynner som basic-medlemmer.) Metoden skal returnere medlemsnummeret som tildeles medlemmet. Metoden skal bestemme medlemsnummeret ved å kalle følgende private metode:

  ```
  private int findAvailableNo()
  ```

  Denne metoden skal hente ut et tilfeldig heltall (bruk klassen Random) som ikke allerede er i bruk som medlemsnummer.
- **checkMembers()** skal gå gjennom alle medlemmene og foreta oppgradering av medlemmer som er kvalifisert for det. Basic-medlemmer kan kvalifisere seg for sølv eller gull, mens sølvmedlemmer kan kvalifisere seg for gull. Metoden skal ta en dato som parameter (som typisk kan være dagens dato).
  *Tips*: Du trenger å finne ut hvilken klasse et objekt tilhører. Bruk operatoren ***instanceof***. Det er ikke mulig å omforme klassetilhørigheten til et objekt. Du må i stedet lage et nytt objekt med data fra det gamle. Det nye objektet må legges inn i samlingen på den plassen der det gamle lå.

Lag en enkel testklient der spesielt metoden *checkMembers()* blir prøvd ut. Du kan finne det hensiktsmessig å lage flere metoder i klassen **MemberArchive** for å få testet tilstrekkelig. Hvis du trenger å skrive ut hvilken klasse et objekt tilhører, kan du bruke metoden *getClass()* i klassen Object.

# Vedlegg 1 – Klassen Personals

```
package oblig2;

/**
 * The Personals-class holds personal information about a member,
 * like: surname, lastnam, e-mail address and password.
 * Since a member never changes his/her personals while changing membership level,
 * it is useful to use a separat class to hold this information.
 * The password can be changed, but it then needs to be different than the old
 * passowrd. The Password is not case sensitive.
 */
class Personals {
  private final String surname;
  private final String firstname;
  private final String emailAddress;
  private String password;

  /**
   * Creates an instance of Personals.
   *
   * @param firstname    first name of the person
   * @param surname      surname of the person
   * @param emailAddress email address of the person
   * @param password     the password of the person
   */
  public Personals(String firstname, String surname, String emailAddress, String password) {
    if (firstname == null
        || surname == null
        || emailAddress == null
        || password == null
        || firstname.trim().equals("")
        || surname.trim().equals("")
        || emailAddress.trim().equals("")
        || password.trim().equals("")) {
      throw new IllegalArgumentException("One or more of the parameters are invalid.");
    }
    this.firstname = firstname.trim();
    this.surname = surname.trim();
    this.emailAddress = emailAddress.trim();
    this.password = password.trim();
  }

  /**
   * Returns the persons first name.
   *
   * @return the persons first name.
   */
  public String getFirstname() {
    return firstname;
  }

  /**
   * Returns the persons surname.
   *
   * @return the persons surname.
   */
  public String getSurname() {
    return surname;
  }

  /**
   * Returns the persons e-mail address.
   *
   * @return the persons e-mail address.
   */
  public String getEMailAddress() {
    return emailAddress;
  }

  /**
```

```
    * Checks if the password given by the parameter matches the password
    * registered on the person. Returns <code>true</code> if password is
    * correct <code>false</code> if password is wrong.
    *
    * @param password the password to test for
    * @return <code>true</code> if password is correct
    * <code>false</code> if password is wrong
    */
   public boolean okPassword(String password) {
     return this.password.equalsIgnoreCase(password);
   }

   /**
    * Changes the password. The new password must be different from the old.
    * Passwords are not case sensitive.
    * Returns <code>true</code> if the password was changed successfully,
    * <code>false</code> if not.
    *
    * @param oldPassword the old password of the person
    * @param newPassword the new password to be changed to
    * @return <code>true</code> if the password was changed successfully,
    * <code>false</code> if not.
    */
   public boolean changePassword(String oldPassword, String newPassword) {
     // A Guard claus, hence return is allowed
     if (oldPassword == null || newPassword == null) {
       return false;
     }
     boolean changeSuccessfull;

     if (!password.equalsIgnoreCase(oldPassword.trim())) {
       changeSuccessfull = false;
     } else {
       password = newPassword.trim();
       changeSuccessfull = true;
     }
     return changeSuccessfull;
   }
}
```

# Vedlegg 2 – Testklassen BonusMemberTest

```
package oblig2;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import java.time.LocalDate;
import static org.junit.jupiter.api.Assertions.*;

/**
 * Test class used to test the calculation of bonus points in the different classes
 * representing the different memberships.
 * To use this class, you first have to set up the Unit-test framework in your IDE.
 */
class BonusMemberTest {

    private LocalDate testDate;
    private Personals ole;
    private Personals tove;

    @BeforeEach
    void setUp() {
         this.testDate = LocalDate.of(2008, 2, 10);
         this.ole = new Personals("Olsen", "Ole",
                 "ole.olsen@dot.com", "ole");
         this.tove = new Personals("Hansen", "Tove",
                 "tove.hansen@dot.com", "tove");
    }

    /**
     * Tests the accuracy of the calculation of points for the basic member Ole.
     * Ole was registered as Basic Member more than 365 days before 10/2-2008, and
     * should not qualify for upgrade.
     */
    @Test
    void testBasicMemberOle() {
        BasicMember b1 = new BasicMember(100, ole,
                LocalDate.of(2006, 2, 15));
        b1.registerPoints(30000);
        System.out.println("Test nr 1: No qualification points");
        assertEquals(0, b1.findQualificationPoints(testDate));
        assertEquals(30000, b1.getPoints());

        System.out.println("Test nr 2: Adding 15 000 points, still no qualification points");
        b1.registerPoints(15000);
        assertEquals(0, b1.findQualificationPoints(testDate));
        assertEquals(45000, b1.getPoints());
    }

    /**
     * Tests the accuracy of the calculation of points for the absic member Tove,
     * who was registered with basic membership less than 365 days before 10/2-2008,
     * and hence does qualify for an upgrade, first to Silver, then to Gold.
     */
    @Test
    void testBasicMemberTove() {
        BasicMember b2 = new BasicMember(110, tove,
                LocalDate.of(2007, 3, 5));
        b2.registerPoints(30000);

        System.out.println("Test nr 3: Tove should qualify");
        assertEquals(30000, b2.findQualificationPoints(testDate));
        assertEquals(30000, b2.getPoints());

        System.out.println("Test nr 4: Tove as silver member");
        SilverMember b3 = new SilverMember(b2.getMemberNo(), b2.getPersonals(),
                b2.getEnrolledDate(), b2.getPoints());
        b3.registerPoints(50000);
        assertEquals( 90000, b3.findQualificationPoints(testDate));
        assertEquals( 90000, b3.getPoints());
```

```
        System.out.println("Test nr 5: Tove as gold member");
        GoldMember b4 = new GoldMember(b3.getMemberNo(), b3.getPersonals(),
                b3.getEnrolledDate(), b3.getPoints());
        b4.registerPoints(30000);
        assertEquals( 135000, b4.findQualificationPoints(testDate));
        assertEquals( 135000, b4.getPoints());

        System.out.println("Test nr 6: Changed test date on Tove");
        testDate = LocalDate.of(2008, 12, 10);
        assertEquals( 0, b4.findQualificationPoints(testDate));
        assertEquals( 135000, b4.getPoints());

    }

    /**
     * Tests the passwords of both members.
     */
    @Test
    void testPasswords() {
        System.out.println("Test nr 7: Trying wrong password on Ole");
        assertFalse(ole.okPassword("000"));
        System.out.println("Test nr 8: Trying correct password on Tove.");
        assertTrue(tove.okPassword("tove"));
    }
}
```