

Tillatte hjelpemidler: alle skrevne og trykte.  
Antall sider (inkludert vedlegg): 21.

- Les igjennom og skaff deg oversikt over hele eksamenssettet før du begynner å løse det.
- Når du løser oppgavene kan du benytte resultater fra oppgaver og deloppgaver du ikke har lyktes i å løse.
- Prosentatsene ved hver oppgave angir omtrentlig vekt ved sensur. Deloppgaver til en oppgave kan være ulikt vektet.
- I kodevedleggene er detaljer utelatt fra implementasjonen når disse ikke er relevante.
- Vedlagte klasser og kontrakter kan dekke flere sider.

1. (15%) Se vedlagt kode for `IBrikke` og `Posisjon`.

Tårn er en sjakkbrikke som kan flyttes én eller flere ruter horisontalt (langs en rad) eller vertikalt (langs en linje), på sjakkbrettet. Anta at en klasse `Taarn` implementer `IBrikke` er implementert for å representere slike brikker. Implementer en metode for å teste `void Taarn.flytt(Posisjon nyPosisjon)` i en ny klasse, `TaarnTest` extends `JUnit.TestCase`. Identifiser minst fire aspekter som bør testes og sørg for at `TaarnTest` tester hver av dem minst én gang.

**Løsning:**

I henhold metodespesifikasjonen funnet i kontrakten `IBrikke` er det naturlig å formulere følgende fire tester på `void Taarn.flytt()`:

- fungerer for horisontale flytt.
- fungerer for vertikale flytt.
- kaster `SjakkException` for trekk som ikke er lovlige for tårn.
- kaster `SjakkException` for trekk til nåværende posisjon.

```
import junit.framework.TestCase;

public class TestTaarn extends TestCase {

    private Taarn tårn;
```

```
@Override
public void setUp(){
    this.tårn = new Taarn(new Posisjon(1,1));
}

public void testFlytt(){

    //test vertikal flytting
    tårn.flytt(new Posisjon(3,1));
    assertTrue(tårn.hentPosisjon().equals(new Posisjon(3,1)));

    //test horisontal flytting
    tårn.flytt(new Posisjon(3,3));
    assertTrue(tårn.hentPosisjon().equals(new Posisjon(3,3)));

    //test feil ved diagonalt trekk
    flyttMedUnntak(new Posisjon(4,4));

    //test feil ved trekk til samme posisjon
    flyttMedUnntak(new Posisjon(3,3));

}

//feiler ekplisitt dersom ikke unntak kastes
//når this.tårn flyttes til nyPosisjon
private void flyttMedUnntak(Posisjon nyPosisjon){
    try{
        tårn.flytt(nyPosisjon);
    }
    catch(SjakkException se){
        return;
    }

    fail();
}
}
```

2. (20%) Se vedlagt kode for IRokadeBrikke og AbstraktBrikke.

En del funksjonalitet er felles for alle brikker i et sjakkspill. Instruksjonene for å flytte en brikke er for eksempel lik for alle brikker, mens restriksjonene for hvilke trekk som er

lovlig er forskjellig. Dette er utnyttet ved å lage en abstrakt klasse **AbstraktBrikke** som kan brukes som et utgangspunkt for å implementere spesifikke brikker, som for eksempel tårn.

I tillegg til de vanlige flyttereglene for tårn i sjakk kan et spesialtrekk, rokade, utføres én gang av hver spiller i løpet av et spill. Trekket innebærer at et tårn og sjakkbrikken konge reposisjoneres samtidig. Brikker som kan inngå i en rokade skal implementere kontrakten **IRokadeBrikke**. En av betingelsene for rokade er at brikkene som skal inngå i rokaden ikke tidligere er flyttet, derfor har **IRokadeBrikke** en metode **boolean flyttet()** som angir om brikken har vært flyttet eller ikke.

Implementer en klasse **Taarn**. Klassen skal:

- implementere kontrakten **IRokadeBrikke**.
- arve fra klassen **AbstraktBrikke**. Implementer konstruktøren og den abstrakte metoden, og overkjør andre metoder der det er nødvendig.

#### Løsning:

```
public class Taarn extends AbstraktBrikke implements
    IRokadeBrikke {

    private boolean flyttet = false;

    public Taarn(Posisjon posisjon) {
        super(posisjon);
    }

    @Override
    public void flytt(Posisjon posisjon) {
        super.flytt(posisjon);
        this.flyttet = true;
    }

    @Override
    public boolean flyttet() {
        return flyttet;
    }

    @Override
    public boolean gyldigTrekk(Posisjon nyPosisjon) {
        return !nyPosisjon.equals(hentPosisjon())
            && (hentPosisjon().hentLinje() == nyPosisjon
```

```
        .hentLinje() || hentPosisjon().hentRad() == nyPosisjon
        .hentRad());
    }
}
```

3. (10%) Se vedlagt klasse `ObjektLagrer`.

- (a) Vis hvordan du vil bruke klassen for å lagre et objekt av typen `List<Integer>` til en fil med navn "heltallsliste.sjo".

**Løsning:**

```
La interlist være en variabel bundet til et objekt av typen List<Integer>:
ObjektLagrer<List<Integer>> lagrer =
    new ObjektLagrer<List<Integer>>();
lagrer.lagreTilFil(integerlist, "heltallsliste.sjo");
```

- (b) Hva er fordelene med å definere metodene `void lagre(E objekt, OutputStream utstrøm)` og `E last(InputStream innstrøm)`, fremfor å skrive de enkle metodekroppene på passende sted i `void lagreTilFil(E objekt, String filnavn)` og `E lastFraFil(String filnavn)` ?

**Løsning:**

generelle strømmer som `OutputStream` og `InputStream` kan kolbes til forskjellige målstrømmer ettersom man vil lese fra / skrive til f.eks. en fil, en nettverkstilkobling, en database eller et annet mål. Dersom logikken for å skrive til en strøm er adskilt fra logikken med å opprette strømmen, koble den til riktig mål, og lukke den igjen, er det lettere å gjenbruke `lagre` og `last` når de blir aktuelt å lagre til og lese fra andre mål.

- (c) Finnes det tilfeller hvor et unntak av typen `ClassCastException` kan oppstå under utføring av metoden `E last(InputStream innstrøm)`? Forklar.

**Løsning:**

Et `ClassCastException` vil kastes av `last` dersom objektet returnert av `innstrøm.readObject()` ikke er av forventet type. Selv om koden garanterer for at objekter lagret med samme objekt som brukes til innlesing ikke vil forårsake et slikt unntak, garanterer den ikke i mot at objekter lagret med et annet `ObjektLagrer`-objekt (med en annen typeparameterisering) blir forsøkt lest. Så et `ClassCastException` kan oppstå.

4. (10%) Se vedlagte klasser `Spillkort`, `Farge` og `Verdi`.

Klassen `Spillkort` representerer vanlige spillkort. Hvert kort har en farge (kløver, hjerte, spar eller ruter) og en verdi (tallene 2-10 eller knekt, dronning, konge eller ess). Farge og verdi er representert ved hver sin oppramstype. Jeg minner om at oppramstyper implementerer kontakten `java.util.Comparable` og har en `compareTo`-metode som definerer en naturlig ordning for oppramstypen som samsvarer med den rekkefølgen oppramskonstantene er deklartert i. For eksempel har oppramskonstantene til `Farge` følgende ordning: `SPAR < KLØVER < RUTER < HJERTER`.

For klassen `Spillkort` skal det defineres en naturlig ordning som ordner kortene etter farge først, og etter verdi for kort med lik farge.

- Skriv full metodesignatur og implementasjon for metoden som `Spillkort` må implementere for å oppfylle kontrakten `Comparable<Spillkort>`.
- Implementer metodene `boolean equals(Object o)` og `int hashCode()` for klassen `Spillkort`.

**Løsning:**

Ferdig implementert klasse, `Spillkort`.

```
/**
 * Representerer et spillkort.
 */
public class Spillkort implements Comparable<Spillkort> {

    // kode gitt i vedlegg, ikke vist.

    @Override
    public boolean equals(Object o) {
        if (o instanceof Spillkort) {
            Spillkort kort = (Spillkort) o;
            return this.farge.equals(kort.farge)
                && this.verdi.equals(kort.verdi);
        }

        return false;
    }

    @Override
    public int compareTo(Spillkort annetKort) {

        int fargeforskjell = this.farge
            .compareTo(annetKort.farge);
        if (fargeforskjell != 0) {
            return fargeforskjell;
        }
    }
}
```

```
    }

    int verdiforskjell = this.verdi
        .compareTo(annetKort.verdi);
    return verdiforskjell;
}

@Override
public int hashCode() {
    return this.verdi.hashCode()
        + this.farge.hashCode();
}

@Override
public String toString() {
    return farge.toString() + ", " + verdi.toString();
}
}
```

- (c) Forklar forholdene som generelt skal holde mellom de tre metodene du har implementert i denne oppgaven.

**Løsning:**

I henhold til kontrakten `java.util.Comparable` er det sterkt anbefalt at `compareTo` og `equals` skal være implementert i henhold til hverandre. Det vil si at `a.compareTo(b) = 0`, hvis og bare hvis `a.equals(b)`. I henhold til spesifikasjonene for `equals` og `hashCode` slik er beskrevet i dokumentasjonen for `java.lang.Object`, skal `hashCode` alltid returnere samme verdi for objekter som er lik i henhold til `equals`. `hashCode` trenger ikke (og kan ofte ikke) returnere forskjellig verdi for alle objekter som er ulik i henhold til `equals`, men nytteverdien av hash-strukturer er avhengig av at dette vanligvis er tilfellet.

5. (10%) En kortstokk er en ordnet samling av kort.

- (a) Implementer en klasse, `Kortstokk`, for å representere en kortstokk.

- `Kortstokk` skal arve fra en av listene i `java.util`. Bruk generisk notasjon for å sikre at en kortstokk ikke kan inneholde andre elementer enn objekter av typen `Spillkort`.
- Lag en konstruktør som oppretter kortstokken med de 52 vanlige kortene (et kort for hver kombinasjon av farge og verdi).

**Løsning:**

```

/**
 * Representerer en kortstokk av vanlige spillkort
 */
public class Kortstokk extends ArrayList<Spillkort> {

    public Kortstokk(){
        for (Farge f: Farge.values()){
            for (Verdi v: Verdi.values()){
                add(new Spillkort(f, v));
            }
        }
    }
}

```

- (b) Forklar hvorfor vi kan si at `Kortstokk` implementerer `java.util.List` og vis hvordan du vil opprette referanser av typen `java.util.List` for å referere til et `Kortstokk`-objekt.

**Løsning:**

`Kortstokk` arver fra en klasse som implementerer `java.util.List` og er således garantert å ha en definisjon for alle metodene definert i denne kontrakten. Type-sikre referanser kan defineres slik:

```
List<Spillkort> stokk2 = new Kortstokk();
```

6. (15%) For kortstokken du implementerte i oppgave 5, lag en metode `void Kortstokk.stokk()`, som stokker kortstokken ved å flette sammen første og andre halvdel av stokken. Dersom  $a_i$  betegner kort nummer  $i$  i den første halvdel av stokken, og  $b_i$  betegner kort nummer  $i$  i andre halvdel av stokken, skal kortene etter stokking være ordnet slik:  $a_0, b_0, a_1, b_1, \dots, a_{25}, b_{25}$ , eller slik:  $b_0, a_0, b_1, a_1, \dots, b_{25}, a_{25}$ .

Denne stokkeprosedyren bør gjentas for å få en tilfredstillende stokking, men det trenger du ikke ta høyde for.

**Løsning:**

To forslag:

```
public void stokk(){
```

```
int halvStokk = this.size()/2;
for (int i = halvStokk; i > 0; i--){
    this.add(i, this.remove(this.size()-1));
}
}
```

og:

```
public void stokk(){
    int size = size();
    List<Spillkort> andreHalvpart =
        new ArrayList<Spillkort>(subList(size()/2, size()));
    this.removeAll(andreHalvpart);

    for (int i=0; i < size; i+=2){
        add(i, andreHalvpart.remove(0));
    }
}
```

7. (20%) Se vedlagt klasse `SortertSett`.

Klassen `SortertSett` implementerer et sett (kontrakten `java.util.Set`) ved å lagre data i en privat liste som til enhver tid er sortert. Oppslag og innsetting gjøres ved å søke etter riktig posisjon for element som skal finnes eller settes inn. Ettersom listen alltid er sortert kan binærøøk benyttes.

Kun én av metodene som må implementeres i henhold til kontrakten `java.util.Set` er vist i vedlegget. Denne er tatt med for å illustrere bruken av den private metoden `int finnIndeks(E element)`.

- (a) Forklar den generiske typen spesifisert med `<E extends Comparable<? super E>>`. Hvilke betingelser må være oppfylt for typer som `E` skal parameteriseres til når det opprettes objekter av klassen `SortertSett`?

**Løsning:**

`E` må implementere kontrakten `Comparable<A>`, hvor `A` er av samme type som `E` eller `A` er typen til en klasse som `E` arver i fra. Typen er spesifisert slik fordi `E` kan være definert av en klasse som arver sin `Comparable`-implementasjon.

- (b) Hva er fordelene med å benytte `LinkedList` fremfor for eksempel `ArrayList` i denne situasjonen?

**Løsning:**

Anta at objektene blir lagt til settet i vilkårlig rekkefølge. Når oppdateringer ikke systematisk gjøres til enden av datastrukturen har `LinkedList` fordeler ved sletting og innsetting av data, da oppdateringer av indekser kun innebærer å oppdatere referanser i nabonodene til elementet som ble slettet, mens f.eks. `ArrayList` må oppdatere alle indekser høyre enn elementet som ble slettet / satt inn.

- (c) Med hensyn på å sikre at datastrukturen forblir sortert til enhver tid, hva er fordelene med å organisere listen som et privat felt, fremfor å la `SortertSett` arve fra `LinkedList` og overkjøre relevante metoder?

**Løsning:**

Dersom `SortertSett` arver fra `LinkedList` vil den ha metodedefinisjoner for innsetting til vilkårlig indeks. Dette vil bryte med sorteringen. Om man velger å implementere en sortering eller kaste et unntak ved kall til disse metodene vil man bryte kontrakten `List`. Ved å kun tilby metoder som holder settet sortert kan man garantere at ikke klassen blir brukt feil.

- (d) Bruk binærsøk til å implementere metoden `int finnIndeks(E element)` i klassen `SortertSett`. Legg til hjelpemetoder og private felter dersom det er nødvendig.

**Løsning:**

Løsning ved hjelp av rekursiv privat metode. Kan også løses ved iterasjon:

```
private int finnIndeks(E element) {
    return finnIndeks(element, 0, this.liste.size());
}

private int finnIndeks(E element, int nedre, int øvre) {

    int str = øvre - nedre - 1;
    int midt = øvre - 1 - str / 2;

    if (øvre == nedre) {
        return nedre;
    } else if (this.liste.get(midt).compareTo(element) > 0) {
        return finnIndeks(element, nedre, midt);
    } else if (this.liste.get(midt).compareTo(element) < 0) {
        return finnIndeks(element, midt + 1, øvre);
    } else {
        return midt;
    }
}
```

}
---

Lykke til.  
Edvin Fuglebakk

## Vedlegg - oppgave 1

```
public interface IBrikke {

    /**
     * Henter brikkens posisjon på sjakkbrettet.
     *
     * @return Brikkens posisjon.
     */
    public Posisjon hentPosisjon();

    /**
     * Flytter brikken til ny posisjon.
     *
     * @param nyPosisjon
     *         posisjonen brikken skal flyttes til.
     * @throws SjakkException
     *         (ikke-kontrollert unntak) dersom
     *
     *         - nyPosisjon ikke kan nåes i henhold til
     *         flyttereglene for brikken.
     *
     *         - nyPosisjon er den samme som nåværende
     *         posisjon.
     */
    public void flytt(Posisjon nyPosisjon);

    /**
     * Finner ut om et trekk til angitt posisjon er lovlig i
     * henhold til flyttereglene for brikken.
     *
     * @param nyPosisjon
     *         posisjonen brikken skal flyttes til.
     * @return true dersom trekket er lovlig, false ellers,
     *         false også dersom brikken allerede står på
     *         nyPosisjon.
     */
    public abstract boolean gyldigTrekk(Posisjon nyPosisjon);
}
```

```
/**
 * Representerer en posisjon på et sjakkbrett, angitt ved
 * rad- og linjenummer.
 *
 * Rader er de horisontale rekkene vanligvis betegnet med
 * heltallene 1-8.
 *
 * Linjer er de vertikale kolonnene vanligvis betegnet med
 * bokstavene a-h (men her altså representert med heltallene
 * 1-8).
 */
public class Posisjon {

    // private felter, ikke vist.

    /**
     * Oppretter posisjonen for angitt rad og linje.
     *
     * @param rad
     *         raden for denne posisjonen.
     * @param linje
     *         linjen for denne posisjonen.
     * @throws SjakkException
     *         (ikke-kontrollert unntak) dersom rad
     *         eller linje ikke er i intervallet [1,8].
     */
    public Posisjon(int rad, int linje) { // kode for konstruktør, ikke vist. }

    /**
     * Henter raden til posisjonen.
     *
     * @return raden som heltall i intervallet [1,8].
     */
    public int hentRad() { //kode for hentRad, ikke vist. }

    /**
     * Henter linjen til posisjonen.
     *
     * @return linjen som heltall i intervallet [1,8].
     */
    public int hentLinje() { //kode for hentLinje, ikke vist. }

    @Override
    public boolean equals(Object o) { // kode for equals, ikke vist. }
```

```
@Override  
public int hashCode() { // kode for hashCode, ikke vist. }  
}
```

## Vedlegg - oppgave 2

```
/**
 * Brikker som oppfyller kontrakten kan inngå i en rokade.
 */
public interface IRokadeBrikke extends IBrikke {

    /**
     * Angir om brikken har vært flyttet.
     *
     * @return true dersom brikken har vært flyttet, false
     *         ellers.
     */
    public boolean flyttet();

}
```

```
/**
 * Representerer en sjakkbrikke.
 *
 * Implementerer funksjonalitet for å manipulere posisjonen
 * til en brikke. Metoden som kontrollerer at reglene for
 * flytting er fulgt må implementeres i subclasser.
 */
public abstract class AbstraktBrikke implements IBrikke {

    private Posisjon posisjon;

    /**
     * Oppretter en brikke med angitt posisjon.
     *
     * @param posisjon
     *         brikkens posisjon på sjakkbrettet.
     */
    public AbstraktBrikke(Posisjon posisjon) {
        this.posisjon = posisjon;
    }

    @Override
    public Posisjon hentPosisjon() {
        return posisjon;
    }

    @Override
    public void flytt(Posisjon nyPosisjon) {
        if (!gyldigTrekking(nyPosisjon)) {
            throw new SjakkException("Ulovlig trekk.");
        }

        posisjon = nyPosisjon;
    }

    @Override
    public abstract boolean gyldigTrekking(Posisjon nyPosisjon);
}
}
```

## Vedlegg - oppgave 3

```
import java.io.*;

/**
 * En enkel serialiserer / deserialiserer.
 *
 * @param <E>
 *         typen til objekter som skal kunne serialiseres
 *         og deserialiseres.
 */
public class ObjektLagrer<E> {

    /**
     * Serialiserer angitt objekt til angitt strøm.
     *
     * @param objekt
     *         objektet som skal serialiseres.
     * @param utstrøm
     *         strømmen objektet skal skrives til.
     * @throws IOException
     *         ved feil ved skriving til strøm.
     */
    public void lagre(E objekt, ObjectOutputStream utstrøm)
        throws IOException {
        utstrøm.writeObject(objekt);
    }

    /**
     * Deserialiserer et objekt av typen <E> fra angitt strøm.
     *
     * @param innstrøm
     *         strømmen objektet skal lese fra.
     * @return det deserialiserte objektet.
     * @throws IOException
     *         ved feil ved lesing fra strøm.
     * @throws ClassNotFoundException
     *         dersom klassedefinsjonen til det leste
     *         objektet ikke blir funnet.
     */
    public E last(ObjectInputStream innstrøm)
        throws IOException, ClassNotFoundException {
        return (E) innstrøm.readObject();
    }
}
```

```
/**
 * Serialiserer angitt objekt til filen med angitt
 * filnavn. Dersom filen finnes fra før vil tidligere
 * innhold bli overskrevet.
 *
 * Skriver eventuelle feilmeldinger til System.err.
 *
 * @param objekt
 *         objektet som skal serialiseres.
 * @param filnavn
 *         filnavnet på filen objektet skal skrives
 *         til.
 */
public void lagreTilFil(E objekt, String filnavn) {
    // kode for lagreTilFil, ikke vist.
}

/**
 * Deserialiserer et objekt av typen <E> fra filen med
 * angitt navn.
 *
 * Skriver eventuelle feilmeldinger til System.err.
 *
 * @param filnavn
 *         navnet på filen som objektet skal leses i
 *         fra.
 * @return det deserialiserte objektet, null dersom det
 *         har oppstått feil ved lesing fra fil.
 * @throws ClassNotFoundException
 *         dersom klassedefinsjonen til det leste
 *         objektet ikke blir funnet.
 */
public E lastFraFil(String filnavn)
    throws ClassNotFoundException {
    // kode for lastFraFil, ikke vist.
}
}
```

## Vedlegg - oppgave 4

```
/**
 * Representerer et spillkort.
 */
public class Spillkort implements Comparable<Spillkort> {

    private Farge farge;
    private Verdi verdi;

    /**
     * Oppretter spillkort med angitt farge og verdi.
     *
     * @param farge
     *           spillkortets farge.
     * @param verdi
     *           spillkortets verdi.
     */
    public Spillkort(Farge farge, Verdi verdi) {
        this.farge = farge;
        this.verdi = verdi;
    }

    /**
     * Henter fargen til spillkortet.
     *
     * @return fargen til spillkortet.
     */
    public Farge hentFarge() {
        return farge;
    }

    /**
     * Henter verdien til spillkortet.
     *
     * @return verdien til spillkortet.
     */
    public Verdi hentVerdi() {
        return verdi;
    }

    //implementer Comparable<Spillkort>, se oppgavetekst.
}
```

```
/**
 * De fire "fargene" som vanlige spillkort kan ha (fransk
 * kortstokk).
 */
public enum Farge {

    SPAR, KLØVER, RUTER, HJERTER

}

/**
 * Verdiene som vanlige spillkort kan ha.
 */
public enum Verdi {

    TO, TRE, FIRE, FEM, SEKS, SYV, ÅTTE, NI, TI,
    KNEKT, DRONNING, KONGE, ESS

}
```

## Vedlegg - oppgave 7

```
import java.util.*;

/**
 * Sett som holder data sortert til enhver tid. Binær søk
 * brukes for innsetting, oppslag og fjerning av data.
 *
 * @param <E>
 *         Forklar i eksamensbesvarelsen, se oppgavetekst.
 */
public class SortertSett<E extends Comparable<? super E>>
    implements Set<E> {

    private List<E> liste = new LinkedList<E>();

    /**
     * Definert i java.util.Set<E>.
     *
     * Legger til angitt element dersom dette ikke allerede
     * finnes i settet.
     *
     * @param element
     *         elementet som skal legges til settet.
     * @return true dersom elementet ble lagt til, false
     *         dersom et identisk element allerede var i
     *         settet.
     */
    @Override
    public boolean add(E element) {
        int indeks = finnIndeks(element);
        if (liste.size() > indeks
            && liste.get(indeks).equals(element)) {
            return false;
        } else {
            liste.add(indeks, element);
            return true;
        }
    }
}
```

```
/**
 * Finner indeks i liste hvor element skal befinne seg.
 *
 * @param element
 *         elementet som skal finnes, eller settes
 *         inn, i settet.
 * @return indeks til elementets posisjon i listen.
 */
private int finnIndeks(E element) {
    // skriv koden for finnIndeks, se oppgavetekst.
}

// øvrige metoder av kontrakten Set, ikke vist.
}
```