NTNU
Norges teknisk-
naturvitenskapelige universitet

Fakultet for informasjonsteknologi,
matematikk og elektroteknikk
Institutt for datateknikk
og informasjonsvitenskap

BOKMÅL

EKSAMEN I FAG
TDT4100 Objektorientert programmering

Fredag 6. juni 2008
Kl. 09.00 – 13.00

Faglig kontakt under eksamen:
Hallvard Trætteberg, tlf (735)93443 / 918 97263

Tillatte hjelpemidler:

- Én og kun én trykt bok, f.eks. Liang: Introduction to Java Programming.

Sensuren faller 3 uker etter eksamen og gjøres deretter kjent på http://studweb.ntnu.no/.

Prosentsatser viser hvor mye hver oppgave teller innen settet.

Merk: All programmering skal foregå i Java.

Lykke til!

Rammen rundt oppgaven er spillet sjakk, men du trenger ingen dyp forståelse av spillet for å løse oppgaven. Reglene er noe forenklet. I oppgaven skal det lages klasser for brikkefarge, brikkene og brettet med alle brikkene.

a8 b8 c8 d8 e8 f8 g8 h8 **a7 b7 c7 d7 e7 f7 g7 h7** **a6 b6 c6 d6 e6 f6 g6 h6** **a5 b5 c5 d5 e5 f5 g5 h5** **a4 b4 c4 d4 e4 f4 g4 h4** **a3 b3 c3 d3 e3 f3 g3 h3** **a2 b2 c2 d2 e2 f2 g2 h2** **a1 b1 c1 d1 e1 f1 g1 h1**	Figuren til venstre viser koordinat-systemet. Figuren til høyre viser startoppstillingen, med 1 rad bønder og 1 rad med hhv. tårn, springer, løper, dronning, konge, løper, springer og tårn, for hver spiller. De hvite bøndene flytter oppover og de sorte bøndene nedover.	

Sjakk spilles på et 8x8 brett, hvor rutene angis med bokstavene a-h for kolonnene og tallene 1-8 for radene. Det er to spillere med hver sin farge, hvit eller sort, så en rute på brettet kan være tom eller inneholde en hvit eller sort brikke. Det er 6 forskjellige typer brikker, med hver sine regler for hvordan de flytter. Generelt kan en brikke enten flytte til et tomt felt eller til et felt med en brikke med motsatt farge, heretter kalt en motstanderbrikke. Det siste kalles å *slå*

og brikken som flyttes vil da erstatte motstanderbrikken. Forenklede regler for de 6 typene brikker er som følger:

- *Bonden* flytter 1 rute forover og slår motstandere 1 rute diagonalt forover. Merk at "forover" er oppover for hvit og nedover for svart.
- *Springeren* (hest) flytter 2 ruter i én retning og 1 rute i retning vinkelrett på første, f.eks. 2 ruter frem og 1 til venstre eller 1 rute bakover og 2 til høyre. I motsetning til de andre brikketypene kan en springer hoppe over andre brikker.
- *Løperen* (biskop) flytter 1 eller flere ruter diagonalt i en av 4 retninger og kan ikke hoppe over andre brikker.
- *Tårnet* flytter 1 eller flere ruter rett frem eller til siden i en av 4 retninger og kan ikke hoppe over andre brikker.
- *Dronningen* flytter 1 eller flere ruter rett frem, til siden eller diagonalt i en av 8 retninger og kan ikke hoppe over andre brikker.
- *Kongen* flytter 1 rute rett frem, til siden eller diagonalt i en av 8 retninger. Dersom kongen står slik at den kan slås av motstanderen i neste trekk, så er den *sjakk*.

I oppgaven vil det bli definert et grensesnitt kalt **Piece** og dette grensesnittet skal implementeres av de 6 konkrete brikkeklasser **Pawn** (bonde), **Knight** (springer), **Bishop** (løper), **Rook** (tårn), **Queen** (dronning) og **King** (konge). Merk at det kun er tre av disse som skal implementeres av dere.

Nedenfor vil vi beskrive det du skal implementere i mer detalj. *Dersom du ikke klarer å implementere en eller flere metoder, så husk at du likevel kan bruke dem i andre oppgaver, slik at du viser hva du kan!* Innfør gjerne hjelpemetoder for å gjøre løsningen ryddigere.

OPPGAVE 1 (40%): Sjakkbrikker- og brett

a) Implementer en enum-klasse kalt **PieceColor**, med verdiene **WHITE** og **BLACK**. Implementer metoden **getOtherColor**, som ikke tar noen parametre og som returnerer den andre fargeverdien. Dersom du ikke vet hvordan enum-klassen og metoden implementeres, så beskriv en alternativ teknikk for å representere fargen til en brikke. Hvorfor er en enum-klasse å foretrekke fremfor å bruke f.eks. **String** eller **int**?

```
public enum PieceColor {
	WHITE, BLACK;

	public PieceColor getOtherColor() {
		return this == WHITE ? BLACK : WHITE;
	}
}
```

Det er uheldig å bruke String eller int, fordi en da ikke kan begrense verdiområdet vha. statisk typesjekking. Skal en da begrense hvilke instanser som kan være gyldige verdier, må en legge inn en dynamisk sjekk i koden. Det finnes flere alternative løsninger, dersom en ikke bruker enum:

- Det beste er å innkapsle en Boolean i en PieceColor-klasse, og la false/true tilsvare white/black (eller omvendt). Dette vil fungere relativt bra siden en er garantert at det kun finnes to ulike Boolean-verdier. Det er mulig å bruke Boolean direkte, siden dette kan sees på som en boolean med to verdier. En får da ikke en ny klasse med navn PieceColor, og må definere getOtherColor som en hjelpemetode.

- En kan også innkapsle en String eller int, men må da sørge for at det kun kan lages to instanser, vha. riktig bruk av konstruktører og private-modifikatorer. Dette vil nesten tilsvare en reimplementasjon Java sin egen enum-mekanisme.

b) Klassen **Board** skal representere sjakkbrettet vha. en tabell (array) med **Piece**-objekter, med dimensjoner 8x8. Metodene som refererer til ruter skal bruke **String**-posisjoner (koordinater) på formatet "a1" til "h8", som vist i figuren over. Implementer **Board**-klassen med den interne tabellen og følgende to *innkapslingsmetoder*.

```
Piece getPiece(String position) { … }
void setPiece(String position, Piece piece) { … }
```

- **getPiece** returnerer brikken på ruta angitt med **position,** eller null dersom ruta er tom. F.eks. skal **getPiece("a1")** returnere innholdet i ruta nederst til venstre.
- **setPiece** plasserer brikken **piece** (som kan være null) i ruta angitt med **position**.

Hint: Definer hjelpemetoder for å konvertere til/fra **String**-posisjoner og indeksene til tabellen, og bruk disse i løsningen i denne og senere deloppgaver. Dersom du er usikker på hvordan disse implementeres, så definer presist hva de gjør og bruk dem likevel.

```
private Piece[][] pieces;

private static int getColumnIndex(String position) {
	return position.charAt(0) - 'a';
}
private static int getRowIndex(String position) {
	return position.charAt(1) - '1';
}

public Piece getPiece(String position) {
	return getPiece(getColumnIndex(position), getRowIndex(position));
}
private Piece getPiece(int column, int row) {
	return pieces[row][column];
}

public void setPiece(String position, Piece piece) {
	pieces[getRowIndex(position)][getColumnIndex(position)] = piece;
}
```

Løsningen inneholder 3 vesentlige elementer: 1) deklarasjonen av riktig tabelltype og riktig bruk av indekser, 2) omregning til indeks vha. char-aritmetikk og 3) riktig bruk private- og public-modifikatorer. Det er mulig, men ikke like bra, å bruke en én-dimensjonal tabell, så lenge den kapsles inn riktig. Det er også greit å klare seg uten hjelpemetoder, men det teller positivt dersom det ville gjort koden for øvrig enklere.

c) Implementer følgende hjelpemetoder i **Board**-klassen (disse metoden er nyttige når logikken for lovlige flytt skal implementeres i brikkeklassene):

```
boolean isStraight(String from, String to) { … }
boolean isDiagonal(String from, String to) { … }
boolean isOccupiedBetween(String from, String to) { … }
```

- **isStraight** returnerer en **boolean** som angir om et flytt fra **from** til **to** går *langs* en rad *eller* kolonne, altså i en av de 4 retningene parallelt med kantene på brettet.

- **isDiagonal** returnerer en **boolean** som angir om et flytt fra **from** til **to** går diagonalt, altså i en av de 4 retningene på skrå over brettet. Hint: Hva er forholdet mellom endringen i rad og kolonne i et diagonalt flytt?
- **isOccupiedBetween** skal returnere en **boolean** som angir om én eller flere av rutene mellom (og ikke inkludert) **from** og **to** er fylt med en brikke.

```
	public static boolean isStraight(String from, String to) {
		return getColumnDistance(from, to) == 0 || getRowDistance(from, to) ==
0;
	}
	public static boolean isDiagonal(String from, String to) {
		return getColumnDistance(from, to) == getRowDistance(from, to);
	}

	public boolean isOccupiedBetween(String from, String to) {
		int fromColumn = Board.getColumnIndex(from), fromRow =
Board.getRowIndex(from);
		int toColumn = Board.getColumnIndex(to), toRow =
Board.getRowIndex(to);
		int dColumn = (fromColumn == toColumn ? 0 : (toColumn - fromColumn) /
Math.abs(toColumn - fromColumn));
		int dRow = (fromRow == toRow ? 0 : (toRow - fromRow) / Math.abs(toRow
- fromRow));
		for (int column = fromColumn + dColumn, row = fromRow + dRow; column
!= toColumn || row != toRow; column += dColumn, row += dRow) {
			if (getPiece(column, row) != null) {
				return true;
			}
		}
		return false;
	}
```

Her er det et poeng å skjønne hvordan indeksene kan brukes til å avgjøre om flyttet er rett eller diagonalt. I løsningen har jeg definert to hjelpemetoder som er nyttige i andre metoder. isOccupiedBetween er litt fiklete å få til rett, men poenget er å lage en generell løkke hvor inkrementet er regnet ut på bakgrunn av differansene. En må da bruke variabler, += og == i løkka, siden en ikke ved hvilken retning (oppover eller nedover) løkkevariabelen beveger seg. I oppgaveteksten var det forøvrig ikke sagt at isOccupiedBetween bare skulle håndtere flytt som var rette eller diagonale, men ut fra reglene skulle det gå frem at den kun var relevant i dette tilfellet (og altså ikke for hesten).

d) Hva betyr modifikatoren **static**? Slike hjelpemetoder som **isStraight**, **isDiagonal** og **isOccupiedBetween** er ofte **static**, hvorfor?

Static-modifikatoren angir at metoden ikke utføres i kontekst av en (implisitt) instans, og følgelig ikke har tilgang til annet enn andre static-metoder og static-felt. Det er greit å markere at en metode ikke bruker instansdata ved å bruke static-modifikatoren. Merk at dette gjelder isStraight og isDiagonal, men ikke isOccupiedBetween, siden de første kun trenger parametrene for å klassifisere flyttet, mens sistnevnte trenger å sjekke brikkene som er på brettet.

e) Implementer **Rook** (tårn)-, **Queen** (dronning)- og **Knight**-klassene, som alle implementerer grensesnittet **Piece**:

```
	public interface Piece {
		public PieceColor getPieceColor();
		public boolean canTake(String from, String to, Board board);
```

```
        public boolean canMove(String from, String to, Board board);
    }
```

- **getPieceColor** returnerer fargen til brikken.
- **canMove** returnerer en **boolean** som angir om denne brikken, dersom den står på **from**-posisjonen, har lov til å flytte til **to**-posisjonen. **board** er brettet som brikken står på og kan brukes for å sjekke om trekket er mulig/lovlig.
- **canTake** returnerer en **boolean** som angir om denne brikken, dersom den står på **from**-posisjonen, har lov til å ta en motstanderbrikke på **to**-posisjonen. **board** er brettet som brikken står på og kan brukes for å sjekke om trekket er mulig/lovlig.

Hva er fordelen med å definere et slikt felles grensesnitt? Skriv også kode som sikrer at brikkefargen blir satt når brikken opprettes og ikke kan endres siden.

Følgende blir felles for de tre klassene:

```
    private final PieceColor pieceColor;

    public PieceColor getPieceColor() {
        return pieceColor;
    }

    public Klassenavn(PieceColor pieceColor) {
        this.pieceColor = pieceColor;
    }

    public boolean canTake(String from, String to, Board board) {
        return canMove(from, to, board);
    }
```

Følgende kode er spesifikk for hver brikketype:

Rook (tårn):

```
    public boolean canMove(String from, String to, Board board) {
        return Board.isStraight(from, to) && (! board.isOccupiedBetween(from,
to));
    }
```

Queen (dronning):

```
    public boolean canMove(String from, String to, Board board) {
        return (Board.isStraight(from, to) || Board.isDiagonal(from, to)) &&
(! board.isOccupiedBetween(from, to));
    }
```

Knight (springer/hest):

```
    public boolean canMove(String from, String to, Board board) {
        return Board.getColumnDistance(from, to) * Board.getRowDistance(from,
to) == 2;
    }
```

Poenget her er riktig bruk av hjelpemetodene, og at canTake kan kalle den andre direkte, evt. omvendt. Knight-implementasjonen kan gi litt mange tester, med mindre en finner en smart måte å sjekke om differansen i den ene retningen er +-2 og i den andre +-1, slik jeg har gjort her. Jeg har valgt å ikke teste på om to-feltet inneholder en motstanderbrikke for canTake og er tom for canMove, da dette likevel testes i main-metoden lenger ned. I vurderingen bør en sjekke at totaloppførselen blir riktig.

Fordelen med et felles grensesnitt er at annen kode kun trenger å forholde seg til logikken som er definert for grensesnittet og ikke de spesifikke implementasjonene. Dermed kan brettet og brikkene implementeres uavhengig av hverandre, med grensesnittdefinisjonen som "kontrakt". En generell fordel er at det er lettere å legge til nye implementasjoner, selv om det ikke er aktuelt her. Brikke-implementasjonene er heller ikke bundet til å arve fra en bestemt klasse, men kan gjøre det slik det er mest praktisk.

Merk at vi ikke har laget en set-metode for fargen og at vi bare har én konstruktør som tar ikke fargen, slik at vi sikrer at fargen ikke kan endres. Vi har også brukt final-modifikatoren, men dette krever vi ikke i løsningen.

OPPGAVE 2 (25%): Sjakk forts.

a) Du har kanskje oppdaget at brikkeklassene har en del kode som er lik. Beskriv hvordan en abstrakt superklasse, f.eks. kalt **AbstractPiece**, kan innføres og implementeres for å unngå duplisering av kode (om du ikke har implementert brikkeklassene, så svar generelt på spørsmålet). Her må du ta høyde for logikken/implementasjonen til de andre brikke-klassene, f.eks. **Pawn** (bonde). Hvorfor bør en slik superklasse være abstrakt?

Motivet er å spare kode, ved å samle det som er felles i en superklasse. I dette tilfellet er brikkefargen, konstruktøren og get-metoden og den ene av can -metodene.

```
public abstract class AbstractPiece implements Piece {

	private final PieceColor pieceColor;

	public PieceColor getPieceColor() {
		return pieceColor;
	}

	protected AbstractPiece(PieceColor pieceColor) {
		this.pieceColor = pieceColor;
	}

	public boolean canTake(String from, String to, Board board) {
		return canMove(from, to, board);
	}
}
```

Det er strengt tatt ikke nødvendig å implementere Piece, men det er en fordel fordi det jo er et krav for subklassene og en da ikke trenger å deklarere den andre can-metoden som en abstract-metode. Subklassene blir forenklet:

```
public class Rook extends AbstractPiece {

	protected Rook(PieceColor pieceColor) {
		super(pieceColor);
	}

	public boolean canMove(String from, String to, Board board) {
		… som før …
	}
}
```

Merk at konstruktøren må være med, siden en arvet konstruktør ikke kan brukes utenifra som vanlig metoder. Klassen må være abstrakt, siden det ikke skal gå an å lage en instans av den, siden den jo mangler en implementasjon av en metode fra Piece-grensesnittet.

b) Forklar hvordan du kan (kode for å) sikre at en **Board**-instans har en gyldig start-tilstand og at den ikke kan settes i en ugyldig tilstand utenifra (fra andre klasser), dvs. at kun lovlige trekk kan utføres?

Poenget her er å lage en konstruktør som lager og plasserer brikkene riktig, og i tillegg stramme inn innkapslingen slik at en kun har en public-metode for et *helt* flytt, f.eks. movePiece, istedenfor som nå, hvor en har en public setPiece-metode. Dersom setPiece-metoden ikke markeres som private, vil en kunne sette brettet i en ulovlig tilstand.

c) Definer en unntaksklasse for ulovlige posisjoner, f.eks. ”a0”, ”i3” og ”tdt4100”, og forklar hvordan og hvor den brukes for å si fra om slike feil. Begrunn valg av superklasse.

```
public class IllegalBoardPositionException extends RuntimeException {

	private String position;

	public IllegalBoardPositionException(String position) {
		super(position + " is an illegal chess board position");
		this.position = position;
	}

	public String getPosition() {
		return position;
	}
}
```

Det vanlige er å lage en valideringsmetode for posisjoner, som bruker throw new IllegalBoardPosition(position) for å si fra om unntaket. Denne kalles enten direkte eller indirekte fra alle metoder som har posisjonsparametre, f.eks. getPiece og setPiece. Her har vi valgt å definere en såkalt unchecked exception, fordi dette er feil som kan oppstå overalt og som kan unngås i koden som kaller vår, og som ikke skyldes eksterne faktorer vi ikke har kontroll med. Det riktigste hadde forøvrig vært å subklasse IllegalArgumentException, men denne er det ikke sikkert alle husker. I vurderingen er vi ikke så nøye med om posisjonen lagres, men en bør sørge for at getMessage returnerer noe fornuftig, enten ved å kalle superklassens konstruktør, eller ved å definere en egen toString eller getMessage-metode.

d) Anta at **Board** har en konstruktør som setter opp brikkene riktig og at klassen i tillegg har følgende metode:

```
	boolean isCheck(PieceColor color) { … }
```

- **isCheck** returnerer en **boolean** som angir om kongebrikken med fargen **color** er sjakk.

Implementer en main-metode , som oppretter et sjakkbrett og lar brukeren skrive inn flytt. Velg selv hva metoden skriver ut og hva brukeren må skrive inn. Husk på at hvit trekker først og at de deretter veksler mellom å trekke. Metoden må sikre at kun lovlige trekk blir utført. Merk at du ikke trenger å implementere resten av brikkeklassene! Du trenger ikke bry deg om regler ut over de som er beskrevet over.

Til høyre ser du mulig utskrift og input ved kjøring av main-metoden.

```
WHITE's turn:
        d2-d3
BLACK's turn:
        e7-e6
WHITE's turn:
        d2-d3
Illegal move!
WHITE's turn:
        d3-e4
BLACK's turn:
```

```
	public boolean isLegalMove(PieceColor pieceColor, String from, String to) {
		try {
			validateBoardPosition(from);
			validateBoardPosition(to);
		} catch (IllegalBoardPositionException e) {
			return false;
		}
		Piece piece = getPiece(from), otherPiece = getPiece(to);
		if (piece == null || piece.getPieceColor() != pieceColor) {
			return false;
		}
		return (otherPiece != null && otherPiece.getPieceColor() !=
pieceColor) ? piece.canTake(from, to, this) : otherPiece == null &&
piece.canMove(from, to, this);
	}

	public void movePiece(String from, String to) {
		Piece piece = getPiece(from);
		setPiece(from, null);
		setPiece(to, piece);
		firePieceMoved(from, to);
	}

	public static void main(String[] args) {
		Board board = new Board();
		PieceColor pieceColor = PieceColor.WHITE;
		BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));
		do {
			System.out.println(pieceColor + "'s turn: ");
			String line = null;
			try {
				line = reader.readLine();
			} catch (IOException e) {
			}
			if (line == null || line.length() == 0) {
				break;
			}
			String[] fromTo = line.split("-");
			if (fromTo.length != 2) {
				System.out.println("Illegal syntax, use from-to, e.g. d2-
d4");
				continue;
			}
			String from = fromTo[0].trim(), to = fromTo[1].trim();
			if (! board.isLegalMove(pieceColor, from, to)) {
				System.out.println("Illegal move!");
				continue;
			}
			board.movePiece(from, to);
			pieceColor = pieceColor.getOtherColor();
			for (PieceColor kingColor: PieceColor.values()) {
				if (board.isCheck(kingColor)) {
					System.out.println("The " + kingColor + " king is
check!");
				}
			}
		} while (board.findKing(PieceColor.WHITE) != null &&
board.findKing(PieceColor.BLACK) != null);
	}
```

Her er det mange ting å få med seg:

- en må opprette et brett
- en må lese inn tekst, og en bør lese med en Reader, som er for tekst, og ikke en InputStream, som er for bytes

- en bør håndtere IOException
- en bør validere posisjonene
- en må sikre at hvit og svart veksler om å flytte
- en må sjekke at en flytter en brikke med riktig farge
- en må sjekke at en ikke flytter til et felt med en brikke av samme farge
- en må bruke brikkens canMove eller canTake avhengig av om feltet er tomt eller inneholder en motstanderbrikke
- det er ikke sagt noe om hva en skal gjøre med isCheck-metoden, men fornuftig bruk trekker opp

Her er det så mange realiseringsalternativer at det viktigste er å få med seg logikken, enn å gjøre det på en bestemt måte.

OPPGAVE 3 (20%): Observert-observatør-samspill

Anta at du har laget **Board**-klassen over, bl.a. med metoder for å plassere/flytte brikkene på brettet. Forklar med tekst og evt. kodesnutter hvordan du vil gå frem for å gjøre **Board** ”observerbar”, dvs. gjøre det mulig for (instanser av) andre klasser å ”følge med” spillets gang. Hvilke klasser, grensesnitt og metoder vil du innføre og hvordan må de virke?

Det generelle mønsteret er som følger:

- En definerer et lytter-grensesnitt med en eller flere metoder som angir hva slags endring som har skjedd. Her har en valget mellom en metode som sier at en posisjon er endret og en som sier at en brikke er flyttet fra-til. I begge tilfeller må også brettet tas inn parameter.
- Board-klassen må kunne registrere slike lyttere.
- I metodene i Board-klassen som endrer brettet direkte, må en si fra til lytterne om endringen. Det er ryddigst å definere en egen fire-metode.

I løsningen under har vi kombinert de to variantene (vi krever ikke kode i besvarelsen).

```
public interface BoardListener {

	public void pieceMoved(String from, String to, Board board);
	public void squareChanged(String position, Board board);
}
```

Board-klassen:

```
	private List<BoardListener> listeners = new ArrayList<BoardListener>();

	public void addBoardListener(BoardListener listener) {
		listeners.add(listener);
	}

	public void removeBoardListener(BoardListener listener) {
		listeners.remove(listener);
	}

	private void fireSquareChanged(String position) {
		for (BoardListener listener: listeners) {
			listener.squareChanged(position, this);
		}
	}
```

```
	public void setPiece(String position, Piece piece) {
		validateBoardPosition(position);
		pieces[getRowIndex(position)][getColumnIndex(position)] = piece;
		fireSquareChanged(position);
	}

	private void firePieceMoved(String from, String to) {
		for (BoardListener listener: listeners) {
			listener.pieceMoved(from, to, this);
		}
	}
	public void movePiece(String from, String to) {
		Piece piece = getPiece(from);
		setPiece(from, null);
		setPiece(to, piece);
		firePieceMoved(from, to);
	}
```

OPPGAVE 4 (15%): Iterator

a) Anta at det finnes en **Iterator<String>**-implementasjon kalt **BoardIterator** som ”generer” alle gyldige **String**-posisjoner. Dette betyr at 64 kall til **next()**-metoden vil gi deg alle de 64 posisjonene ”a1” til ”h8” på brettet (i en eller annen rekkefølgen). Bruk **BoardIterator** til å implementere en metode String findKing(PieceColor pieceColor), som returnerer posisjonen til **King**-brikken med den angitte **pieceColor**.

Poenget her er å forstå iterator-basert iterasjon. Det er ikke nødvendig å kunne den moderne syntaksen.

```
	private String findKing(PieceColor pieceColor) {
		for (String position: this) {
			Piece piece = getPiece(position);
			if (piece instanceof King && piece.getPieceColor() ==
pieceColor) {
				return position;
			}
		}
		return null;
	}
```

b) Implementer en klasse **ArrayIterator** som implementerer **Iterator<String>** og itererer over innholdet i en en-dimensjonal tabell. Definer en passende konstruktør. Du trenger ikke støtte **remove**-metoden.

```
public class ArrayIterator implements Iterator<String> {

	private String[] elements;
	private int pos = 0;

	public ArrayIterator(String[] elements) {
		this.elements = new String[elements.length];
		System.arraycopy(elements, 0, this.elements, 0, elements.length);
	}

	public boolean hasNext() {
		return pos < elements.length;
	}

	public String next() {
```

```
            return elements[pos++];
        }
}
```

Det er vesentlig å bruke String riktig, både i deklarasjonen av klassen og metodene. Det er sikrest å kopiere elementene i konstruktøren. Det er lov å gå veien om ArrayList og f.eks. delegere til en iterator returnert fra Collection/List sin iterator()-metoden.

c) Implementer **BoardIterator**-klassen . Du velger selv rekkefølgen til posisjonene som returneres fra **next**-metoden. Du kan bruke **ArrayIterator**-klassen fra forrige deloppgave om du ønsker, og evt. endre **ArrayIterator**-implementasjonen for å gjøre **BoardIterator** enklere å implementere.

Her kan en enten bruke arv eller delegering. I begge tilfeller er poenget å først lage en tabell med alle mulige posisjoner, og så gi den til ArrayIterator.

Arv-varianten er enklest:

```
public class BoardIterator extends ArrayIterator {

        private static String[] positions = new String[64];

        private static String[] computePositions() {
                if (positions == null) {
                        int pos = 0;
                        for (char column = 'a'; column <= 'h'; column++) {
                                for (char row = '1'; row <= '8'; row++) {
                                        positions[pos++] = String.valueOf(column) +
String.valueOf(row);
                                }
                        }
                }
                return positions;
        }

        public BoardIterator() {
                super(computePositions());
        }
}
```

Med delegering må BoardIterator implementere hasNext- og next-metodene:

```
public class BoardIterator implements Iterator<String> {

        private static String[] computePositions() {
                … som over …
        }

        private ArrayIterator iterator = new ArrayIterator(computePositions());

        public boolean hasNext() {
                return iterator.hasNext();
        }

        public String next() {
                return iterator.next();
        }
}
```