Mutabilitet

INF101 forelesning 24. Februar 2023

Torstein Strømme

Stikkord: dype kopier, refererte typer, mutable/immutable, rekursjon

I dag

- Dype strukturer
  - Likhet
  - Kopiering
- Mutabilitet

Repetisjon: referanser

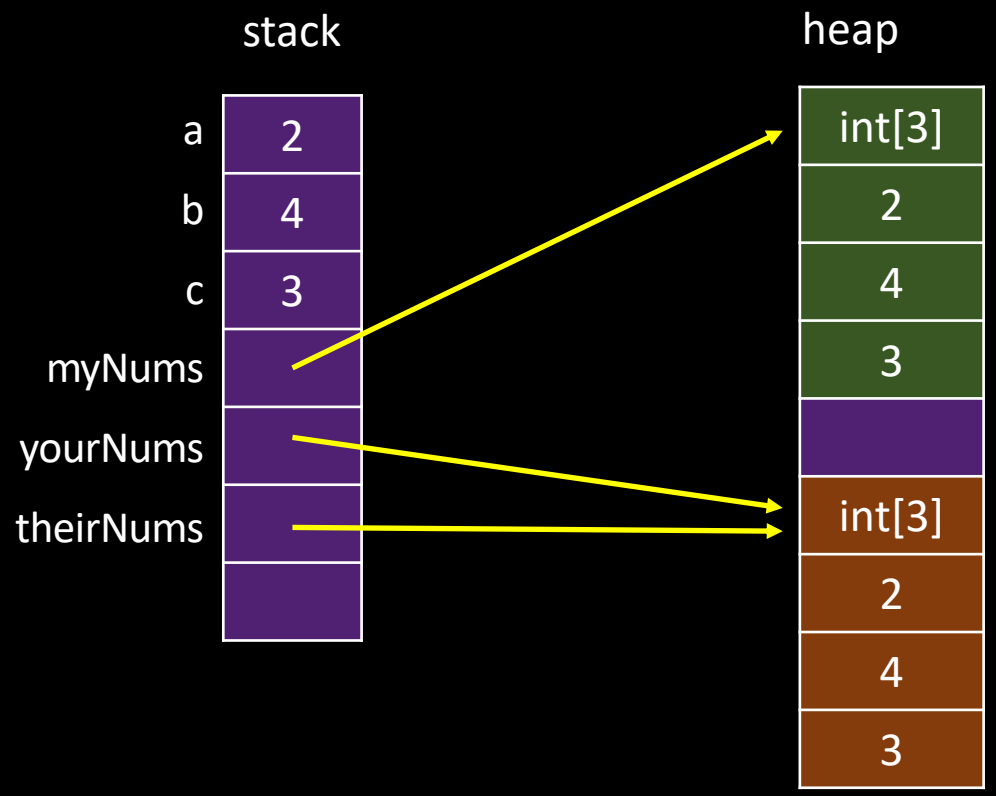
- Alle typer er enten *primitive* eller *refererte typer*
- I Java finnes det 8 primitive typer:

```
boolean, byte, short, int, long, float, double, char
```

- Primitive typer lagres direkte på stack
- Alle andre typer (definert av en klasse, et grensesnitt, array-typer, enum og så videre) er refererte typer. Stack'en har kun en refranse.
- En referanse peker på et objekt, og er egentlig en minneadresse.
- Et objekt er en samling av primitive verdier og referanser til andre objekter.

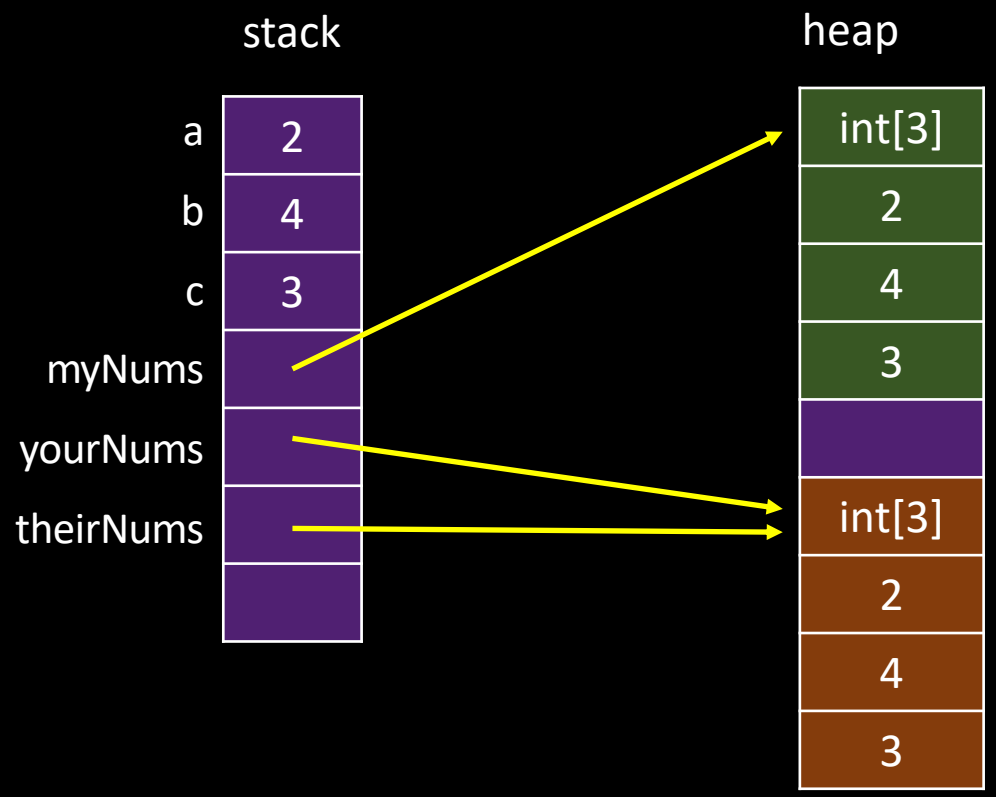
Repetisjon: referanser

```
int a = 2;
int b = 4;
int c = 3;

int[] myNums = { a, b, c };
int[] yourNums = { a, b, c };
int[] theirNums = yourNums;
```



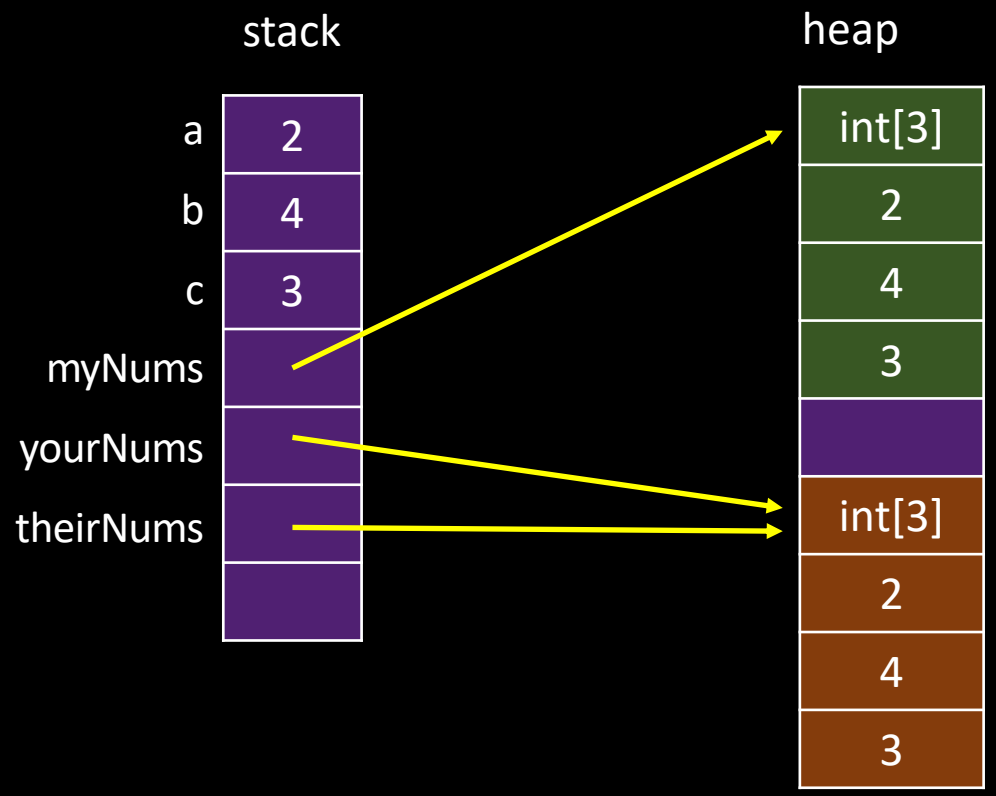
Repetisjon: referanser

```
int a = 2;
int b = 4;
int c = 3;

int[] myNums = { a, b, c };
int[] yourNums = { a, b, c };
int[] theirNums = yourNums;

a = 102;
myNums[1] = 104;
yourNums[2] = 103;
```



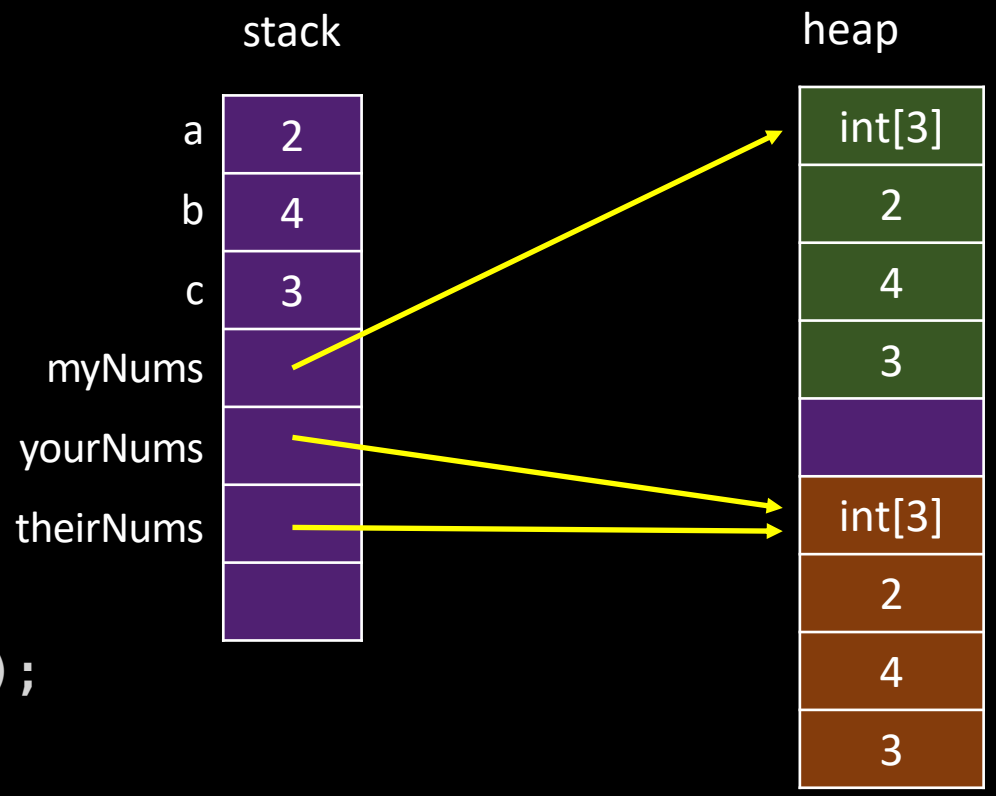
Repetisjon: likhet

```
int a = 2;
int b = 4;
int c = 3;

int[] myNums = { a, b, c };
int[] yourNums = { a, b, c };
int[] theirNums = yourNums;

System.out.println(a == myNums[0]);
System.out.println(myNums == yourNums);
System.out.println(yourNums == theirNums);
```



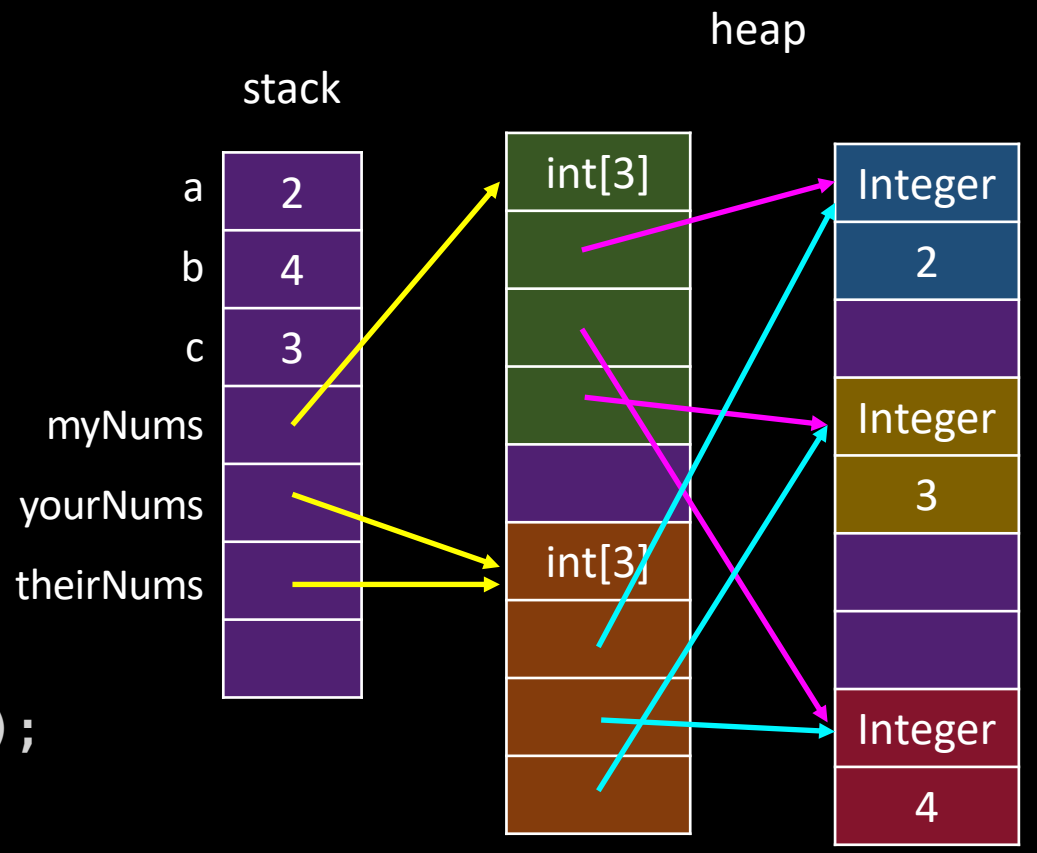
Repetisjon: likhet

```
int a = 2;
int b = 4;
int c = 3;

int[] myNums = { a, b, c };
int[] yourNums = { a, b, c };
int[] theirNums = yourNums;

System.out.println(a.equals(myNums[0]));
System.out.println(myNums.equals(yourNums));
System.out.println(yourNums.equals(theirNums));
```



Repetisjon: likhet

```
Integer a = 2;
Integer b = 4;
Integer c = 3;

Integer[] myNums = { a, b, c };
Integer[] yourNums = { a, b, c };
Integer[] theirNums = yourNums;

System.out.println(a.equals(myNums[0]));
System.out.println(myNums.equals(yourNums));
System.out.println(yourNums.equals(theirNums));
```

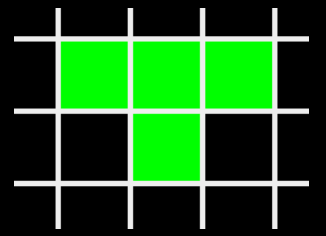


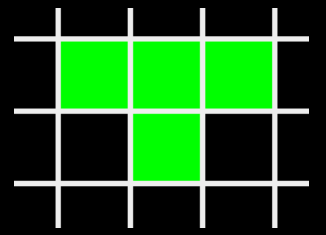
Sjekke likhet av to arrays

```
static boolean arraysAreEqual(Object[] arrA, Object[] arrB) {
    if (arrA.length != arrB.length) {
        return false;
    }
    for (int i = 0; i < arrB.length; i++) {
        Object a = arrA[i];
        Object b = arrB[i];
        if (!a.equals(b)) {
            return false;
        }
    }
    return true;
}
```

Hva med to-dimensjonale arrays?

```
Boolean[][] myTetrisPiece = new Boolean[][] {
    {  true,  true,  true },
    { false,  true, false }
};

Boolean[][] yourTetrisPiece = new Boolean[][] {
    {  true,  true,  true },
    { false,  true, false }
};
```





```
System.out.println(arraysAreEqual(myTetrisPiece, yourTetrisPiece));  // false
```

Dyp likhet for arrayer

```
static boolean arraysAreDeepEqual(Object[] arrA, Object[] arrB) {
    if (arrA.length != arrB.length) {
        return false;
    }
    for (int i = 0; i < arrB.length; i++) {
        Object a = arrA[i];
        Object b = arrB[i];
        if (a instanceof Object[] && b instanceof Object[]) {
            Object[] aCast = (Object[]) a;
            Object[] bCast = (Object[]) b;
            if (!arraysAreDeepEqual(aCast, bCast)) return false;
        }
        else if (!arrA[i].equals(arrB[i])) {
            return false;
        }
    }
    return true;
}
```

Dyp likhet for arrayer

- Dyp likhet: bruk metoden `Arrays.deepEquals`
  - Denne benytter .equals for å sammenligne selve elementene
  - Går så dypt som nødvendig ved hjelp av rekursjon
- Overflatelikhet: bruk `Arrays.equals`
  - Denne benytter *også* .equals for å sammenligne selve elementene
  - fordi .equals er implementert som == for arrays, virker det likevel ikke for 2-dimensjonale arrayer

- Hva med java.util.List? (ArrayList/LinkedList, Arrays.asList etc)
  - Dyp likhet er allerede implementert med .equals

Kopiering

Dyp kopiering

- Hva er problemet her?

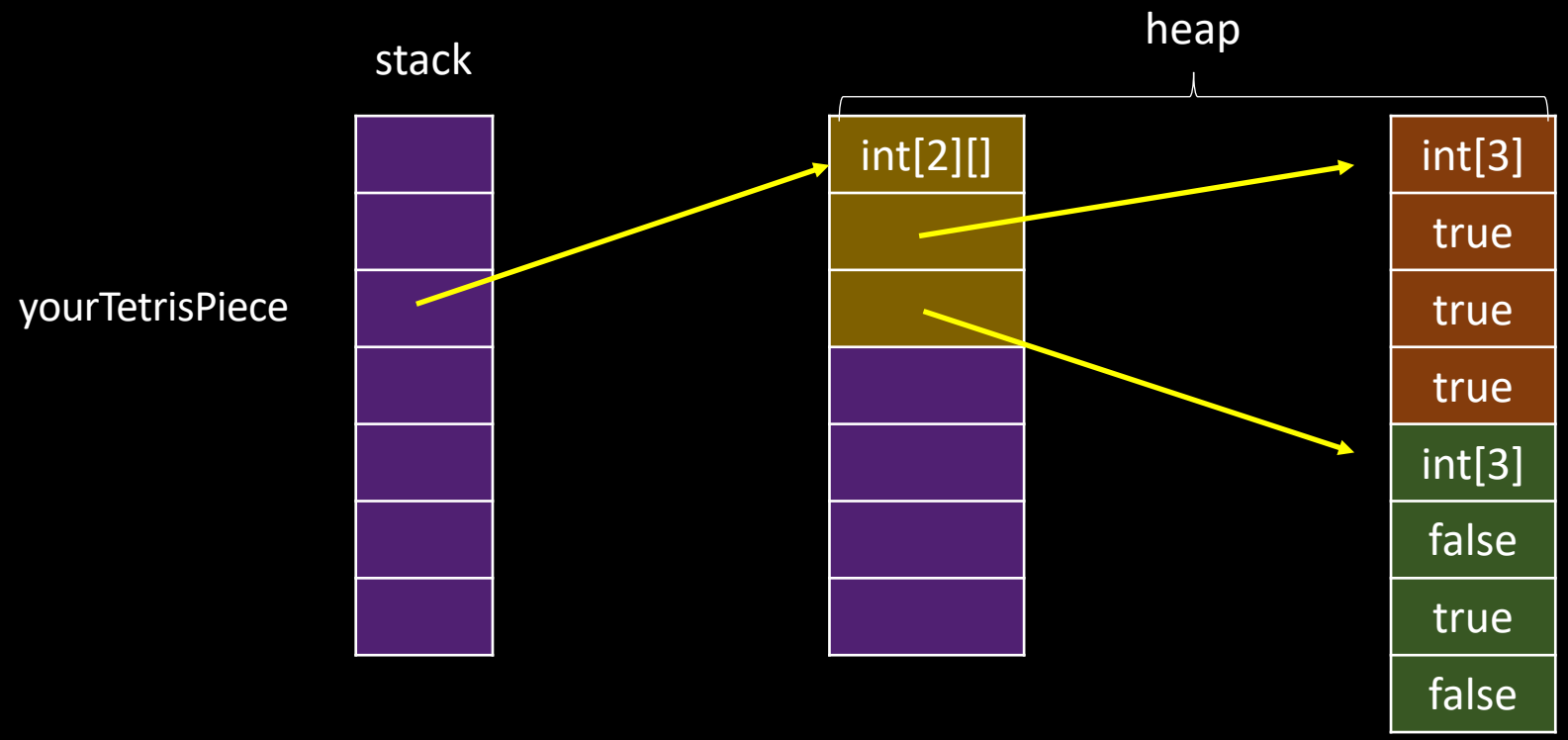
```
boolean[][] theirTetrisPiece = makeCopy(yourTetrisPiece);


static boolean[][] makeCopy(boolean[][] source) {
    int length = source.length;
    boolean[][] target = new boolean[length][];
    for (int i = 0; i < length; i++) {
        target[i] = source[i];
    }
    return target;
}
```

Dyp kopiering

```
boolean[][] yourTetrisPiece = new boolean[][] {
    {  true,  true,  true },
    { false,  true, false }
};
```

- Hva er problemet her?

```
boolean[][] theirTetrisPiece = makeCopy(yourTetrisPiece);
```

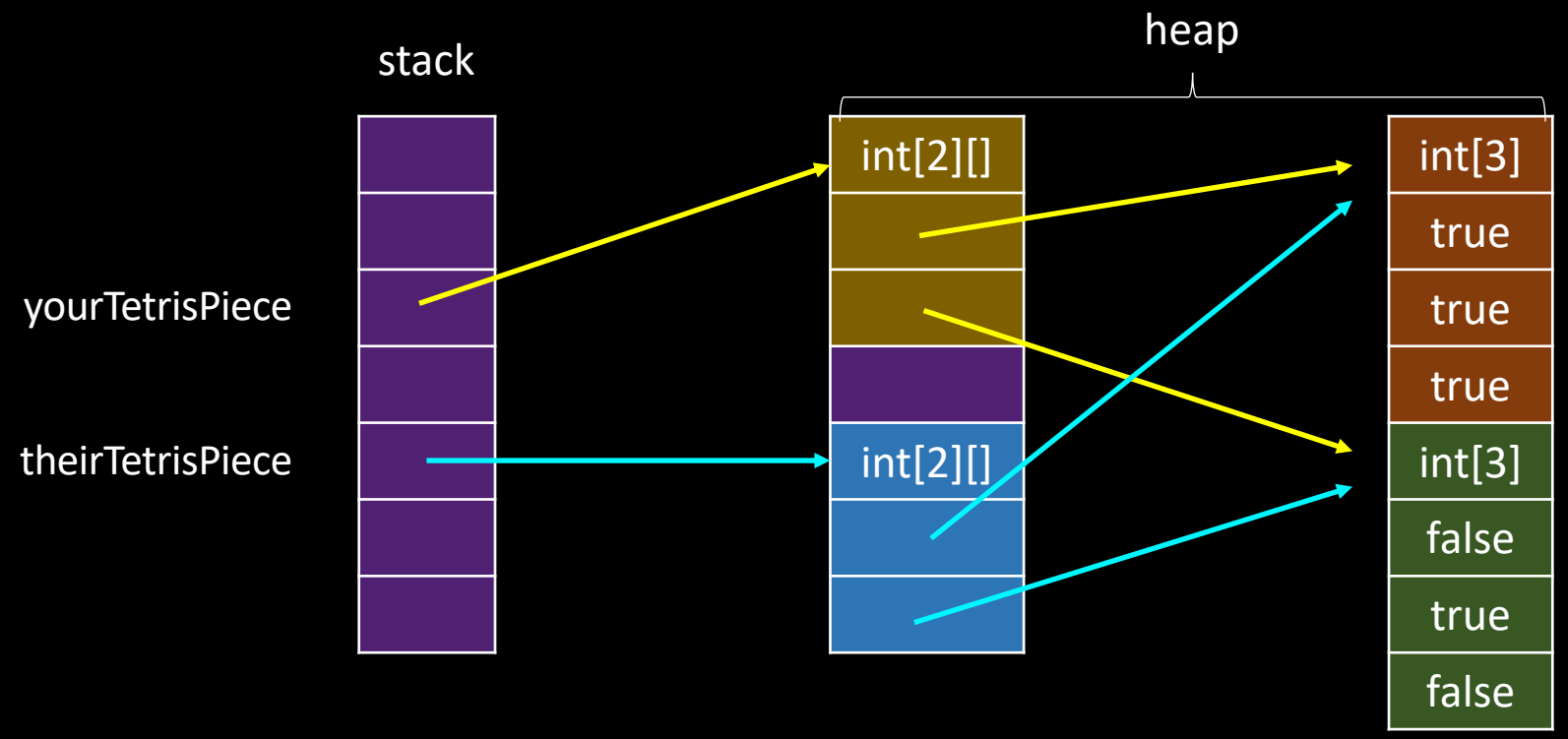


Dyp kopiering

```
static boolean[][] makeCopy(boolean[][] source) {
    int length = source.length;
    boolean[][] target = new boolean[length][];
    for (int i = 0; i < length; i++) {
        target[i] = source[i];
    }
    return target;
}
```

- Hva er problemet her?

```
boolean[][] theirTetrisPiece = makeCopy(yourTetrisPiece);
```



Dyp kopiering i 2 dimensjoner

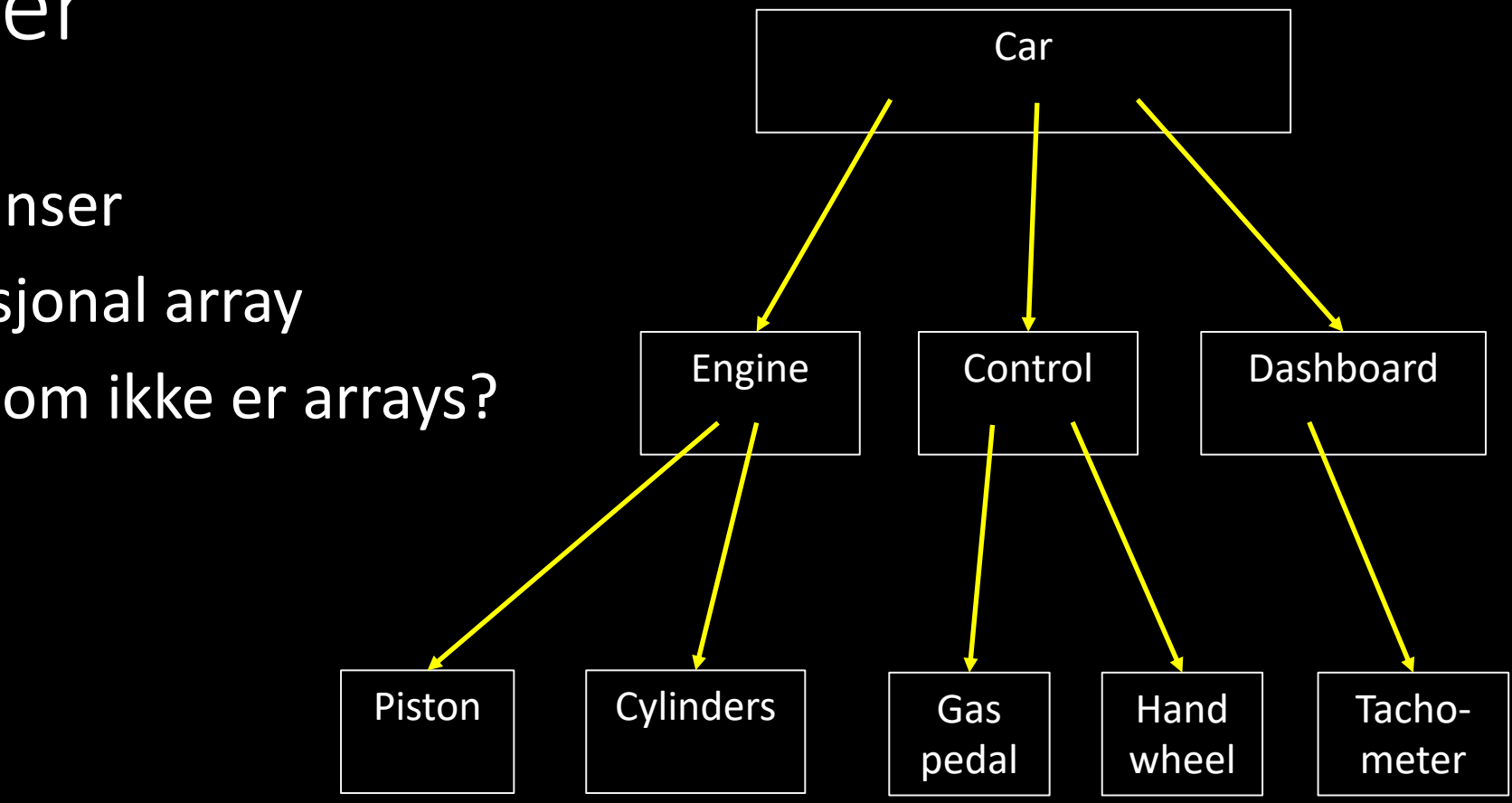
```
target = new Boolean[source.length][];
for (int i = 0; i < source.length; i++) {
    target[i] = new Boolean[source[i].length];
    for (int j = 0; j < source[i].length; j++) {
        target[i][j] = source[i][j];
    }
}
```

Dyp kopiering av arrays generelt

```
static <T> T[] makeDeepCopy(T[] source) {
    int length = source.length;
    T[] target = getNewArrayOfType(source.getClass(), length);
    for (int i = 0; i < length; i++) {
        if (source[i] instanceof Object[]) {
            Object[] sourceItem = (Object[]) source[i];
            @SuppressWarnings("unchecked") // Danger zone
            T copy = (T) makeDeepCopy(sourceItem);
            target[i] = copy;
        }
        else {
            target[i] = source[i];
        }
    }
    return target;
}
```

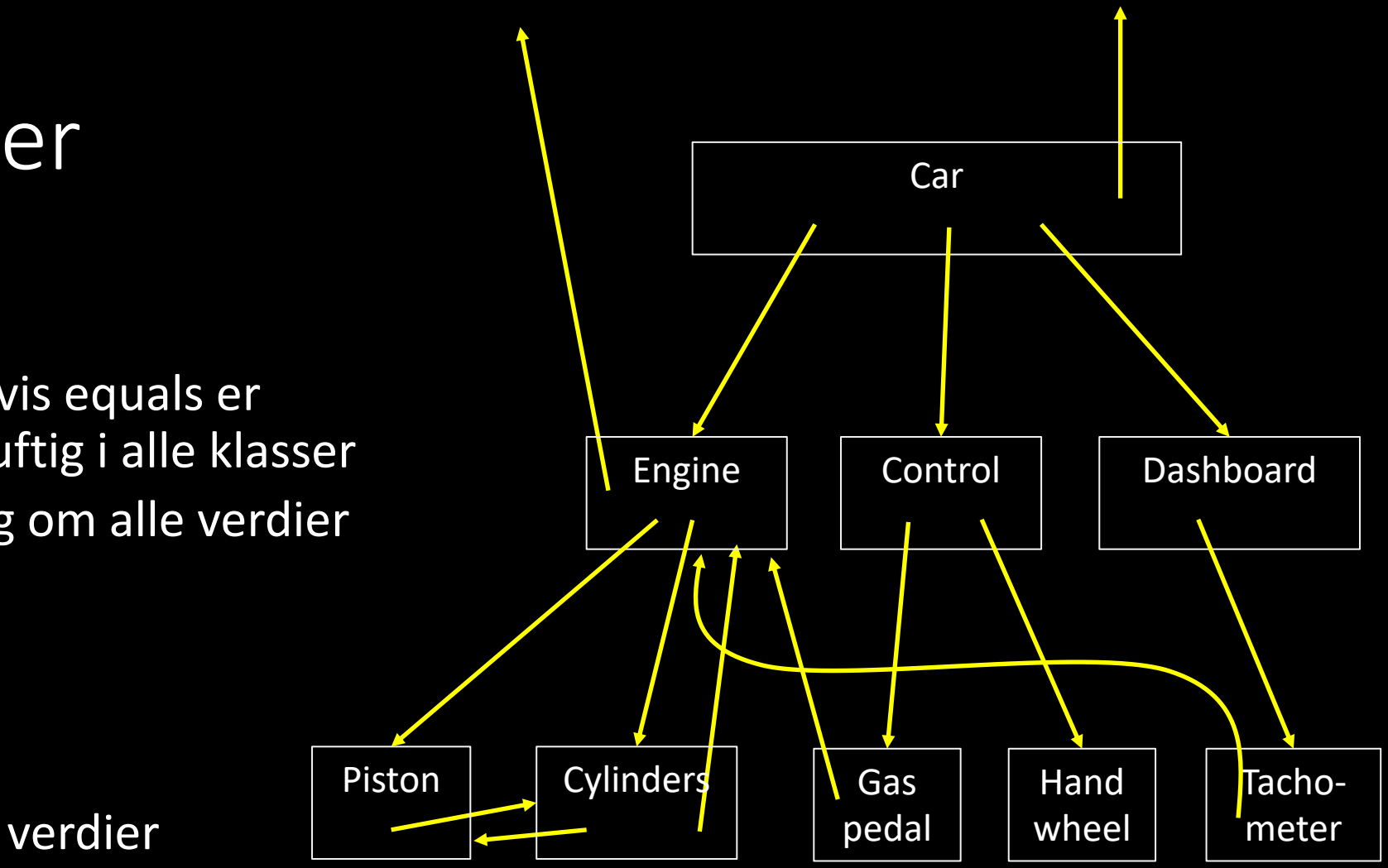
Dype strukturer

- Referanser til referanser
- Eksempel: 2-dimensjonal array
- Hva med Objekter som ikke er arrays?



Dype strukturer

- Likhet
  - Kan fungere dypt hvis equals er implementert fornuftig i alle klasser
  - Trenger ikke bry seg om alle verdier
- Kopiering
  - Ikke opplagt
  - Må bry seg om alle verdier
  - Lettere jo flere objekter som er ”immutable”
  - Bruk grensesnittet `Cloneable`



Immutable

- En klasse er “immutable” eller “uforanderlig” dersom objekter i klassen er umulig å endre på etter at konstruktøren er ferdig kjørt.
  - Alle (tilgjengelig utenfra) feltvariabler er final
  - Det finnes ingen tilgjengelige metoder som kan endre tilstanden til objektet
  - Dersom en intern feltvariabel er mutable:
    - Kan den ikke være injisert som et argument til konstruktøren, slik at andre kan ha en referanse til den som endrer den.
    - Kan den ikke eksponeres på noen måte slik at andre kan endre på den

- Eksempler: `String, Integer, Boolean, LocalDate`

Mutable

- En klasse er “mutable” eller “foranderlig” dersom den ikke er immutable.
  - Objekter har intern “state” som kan endre seg
- Eksempler: `int[]`, `StringBuilder`, `ArrayList`, `Date`

Mutable vs immutable

Mutable	Immutable
Å endre tilstand krever bare endring i én feltvariabel	Å endre tilstand krever å lage et helt nytt objekt.
Flere refreanser til samme objekt kan gjøre det lettere å kommunisere mellom ulike deler av programmet.	Kommunikasjon mellom ulike deler av programmet kan bli litt omstendelig
Flere referanser til samme objekt kan føre til forvirring og uventede bugs.	Flere referanser til samme objekt vil ikke føre til forvirring eller bugs.
Å lage en kopi av objektet krever omtanke.	Å lage en kopi av objektet er helt unødvendig.
Vanskeligere å tenke på	Lettere å tenke på
Vanskeligere å skrive modulær kode	Lettere å skrive modulær kode

Eksempel: Vårfest

```
/** @return the first day of spring this year */
Date startOfSpring() {
    return askSnowdrop();
}
```

```
// somewhere else in the code...
void partyPlanning() {
    Date partyDate = startOfSpring();
    // ...
}
```

Eksempel: Vårfest

```
/** @return the first day of spring this year */
Date startOfSpring() {
    if (this.snowdropAnswer == null) {
        this.snowdropAnswer = askSnowdrop();
    }
    return snowdropAnswer;
}

// somewhere else in the code...
void partyPlanning() {
    Date partyDate = startOfSpring();
    // ...
}
```

Eksempel: Vårfest

```
/** @return the first day of spring this year */
Date startOfSpring() {
    if (this.snowdropAnswer == null) {
        this.snowdropAnswer = askSnowdrop();
    }
    return snowdropAnswer;
}

// somewhere else in the code...
void partyPlanning() {
    Date partyDate = startOfSpring();
    // ...
    // some other constraint showed up, need to move date
    partyDate.setMonth(partyDate.getMonth() + 1);
    // ... uh-oh. what just happened?
}
```

Eksempel: Vårfest

Dårlig løsning 1: endre dokumentasjonen

```
/** @return the first day of spring this year.
The caller may never change the returned object. */
Date startOfSpring() {
    if (this.snowdropAnswer == null) {
        this.snowdropAnswer = askSnowdrop();
    }
    return snowdropAnswer;
}
```

Eksempel: Vårfest

Dårlig løsning 2: returnere en kopi

```
/** @return the first day of spring this year.
*/
Date startOfSpring() {
    if (this.snowdropAnswer == null) {
        this.snowdropAnswer = askSnowdrop();
    }
    return snowdropAnswer.clone();
}
```

Eksempel: Vårfest

God løsning: bruk en immutable data-type

```
/** @return the first day of spring this year.
*/
LocalDate startOfSpring() {
    if (this.snowdropAnswer == null) {
        this.snowdropAnswer = toLocalDate(askSnowdrop());
    }
    return snowdropAnswer;
}
```

Mutable vs immutable

- Prøv å lage klasser immutable hvis mulig
  - I stedet for å ha metoder som endrer objektet, ha metoder som returnerer en endret kopi av objektet.
  - Godt egnet for klasser med relativt små mengder data
- Dersom du bruker mutable klasser
  - Pass på at du ikke klusser det til for andre
  - Pass på å dokumenter godt alle side-effekter i metoder som muterer objektet
  - Pass på at kopier er tilstrekkelig dype for formålet