Objekt Orientert Programmering og programmering i Java

“Object-oriented programming is an exceptionally bad idea which could only have originated in California.”

Edsger W. Dijkstra

Origin: Oslo

Innhold

Variabler

En enkel definisjon av variabler er "navn som refererer til verdier eller objekter."

Når vi deklarerer variabler, trenger vi bare å angi en type og et navn. Det er ikke nødvendig å tildele en verdi ved deklarering. Når vi oppretter nye variabler, kan vi bruke ulike modifikatorer. Disse må brukes i følgende rekkefølge:

1) **public/protected/private (Valgfritt)**
   Dette angir tilgangsnivået til variabelen. Public er tilgjengelig for alle, private er kun tilgjengelig innenfor klassen og ikke for klasser som utvider den. Protected er lik private, men klasser som utvider den har også tilgang. Hvis tilgangsnivået ikke spesifiseres, blir det automatisk public.

2) **Static (Valgfritt)**
   En statisk definert variabel er den samme for alle instanser av klassen. Statiske variabler kan brukes for globale tellere, da de beholder samme verdi uavhengig av instansen.

3) **Final (Valgfritt)**
   En final variabel kan bare tildeles en verdi én gang. Etterpå kan den ikke endres. Dette er nyttig for offentlige variabler i en klasse som skal være tilgjengelig for alle, men ikke endres.

4) **Type (Required)**
   En variabel MÅ ha en type, typen til variabelen sier hva den inneholder. Typen kan være en klasse du har laget String, int, char, boolean, void osv.

Variabler er navnholdere som holder på en verdi. Vi kan gruppere hva en variabel kan inneholde inn til 2 typer, primitive typer og objekter/instanser. Primitive typer er en boolean, char eller en av de forskjellige numeriske typene. Objekter/ instanser er referanser som betyr at de peker til et sted i minne. Dette gjør at i den følgene koden vil begge variablene bli påvirket siden de peker på det samme objektet i minne.

Eksempel kode på peking av objekter i minne

Filnavn: Teller.java

```
public class Teller {
	private int verdi;

	public Teller() {
		this.verdi = 0;
	}

	public void adder() {
		this.verdi++;
	}

	public int hentVerdi() {
		return this.verdi;
	}
}
```

Filnavn: TestProgram.java (Hovedprogram)

```
public class TestProgram {
	public static void main(String[] args) {
		// lag et objekt og la teller2 peke på det samme som peker1
		Teller teller1 = new Teller();
		Teller teller2 = teller1;

		teller1.adder();

		System.out.println(teller1.hentVerdi());
		System.out.println(teller2.hentVerdi());
	}
}
```

Program output

```
1
1
```

Regler til variabler

- Type og modifiers kan bare settes **EN GANG** så etter at det er definert kan du bare endre på verdien dens (med mindre den er final).

Fungerende Kode	IKKE Fungerende Kode
String verdi1; verdi1 = “verdi nummer 1”; String verdi2 = “verdi nummer 2”; verdi2 = “endret verdi 2”;	String verdi1; **String** verdi1 = “verdi nummer 1”; String verdi2 = “verdi nummer 2”; **String** verdi2 = “endret verdi 2”;

Primitive Typer

Type	Gyldig verdi for typen
boolean	Boolean kan inneholde en true/false verdi.
char	En bokstav som må deklareres med enkel quote → char verdi = ‘c’;
byte	Et nummer **uten desimaler** fra -128 til 127.
short	Et nummer **uten desimaler** fra -32 768 til 32 767.
int	Et nummer **uten desimaler** fra -2 147 483 648 til 2 147 483 647.
long	Et nummer **uten desimaler** fra -9 223 372 036 854 775 808 til 9 223 372 036 854 775 807
float	Et nummer **med desimaler** fra 3.40282347 x 10^38 til 1.40239846 x 10^-45
double	Et nummer **med desimaler** fra 1.7976931348623157 x 10^308 til 4.9406564584124654 x 10^-324

Hvorfor så mange nummer typer?

Hvorfor det er flere numeriske typer (int, short, long, float, double) har å gjøre med hvor mye plass de tar opp i minne. En int tar opp 4 bytes, short tar opp 2, long, tar opp 8 osv. Dette er bare nødvendig å tenke på hvis du skal bruke veldig store verdier eller tall med desimaler.

Inkrementasjon

Inkrementasjon er når en nummer verdi økes eller senkes med 1. Dette gjør du ved å bruke ++ eller -- tegnet. Inkrementasjon brukes ofte i sammenheng med loops.

String Typen

String typen er **IKKE EN PRIMITIV TYPE** som betyr at det er et objekt av en klasse. Strenger brukes til å holder på en array av char verdier som da gjør opp en samling av bokstaver/tall/symboler.

Regler for Strings:

- **ALLTID BRUK string1.equals(string2)** istedenfor string1 == string2 når du skal sammenligne 2 strings. Typen char derimot som er en bokstav kan du bruke == eller != på.
- Bruk **“** symbolet for strings istedenfor **‘** symbolet siden det brukes for char

Nyttige Funksjoner:

- **String.equals(String string2)** → boolean
  Dette brukes for å sjekke om 2 strings er like. **ALLTID** bruk dette for strings istedenfor == eller !=.

- **String.toCharArray()** → char[]
  Konverterer Stringen til en char array. Nyttig for å loope gjennom hver av bokstavene i Stringen.

Eksempel kode på iterasjon av alle chars i en string

```
String bokstav_samling = “abc”;

for (char bokstav : bokstav_samling.toCharArray()) {
        if (bokstav == ‘b’) {
                System.out.println(“Bokstav er en b”);
        }
        else {
                System.out.println(“Bokstav er ikke en b”);
        }
}
```

Program output

```
Bokstav er ikke en b
Bokstav er en b
Bokstav er ikke en b
```

- **String.contains(char bokstav)** → boolean
  Sjekker om en string inneholder en bokstav. Husk at bokstaven må være en char.

- **String.split(String split_tekst)** → String[]
  Splitter strengen til en array av strenger ved bokstavene i split_tekst. Dette kan være nyttig for å splitte en streng inn i ordene dens ved å splitte på " " mellomrom strengen.

Eksempel kode på splitting av setning og iterasjon av ordene dens

```
String setning = "det er mandag imorgen.";

String[] alle_ord = setning.split(" ");

for (String ord : alle_ord) {
        System.out.println(ord);
}
```

Program output

```
det
er
mandag
imorgen.
```

- **String.valueOf(Type obj)** → String
  Konverterer et objekt eller en primitiv type om til en String. Dette kan brukes til å konvertere en int verdi til en String med verdien som bokstaver.

- **String.endsWith(String slutt)** → boolean
  Sjekker om en string slutter med bokstavene i slutt strengen som er den gitte parameteren.

- **String.startsWith(String start)** → boolean
  Sjekker om en string starte med bokstavene i start strengen som er den gitte parameteren.

  String.substring(int start_index, int slutt_index) → String
  Henter ut alle bokstaver inne i en String fra en start posisjon til en slutt posisjon. Nyttig for å hente ut et ord eller nummere i en string. Hvis du

ikke gir en slutt_index verdi så vil du hente ut fra start_index til siste bokstav i strengen.

- **String.toLowerCase()** → void
  Gjør alle bokstavene i en string til små bokstaver

- **String.toUpperCase()** → void
  Gjør alle bokstavene i en string til store bokstaver.

Void

Void brukes for å si at en metode ikke returnerer noe..

Kommentarer

- Det finnes 2 måter å lage kommentarer på i java.

//	Brukes til å lage kommentar for en linje. Denne kommentaren påvirker ikke neste linje. Alt skrevet **ETTER //** symbolet blir en kommentar.
/* … */	Brukes til å lage en kommentar som varer flere linjer. Denne kommentaren vil vare fra **/*** symbolet helt til du skriver ***/**.

If Statements

If statements brukes for å utføre kode basert på en tilstanden programmet er i. Et if statement må ha en if (kondisjon) og **kan** ha en else if (kondisjon) og eller else (kondisjon). Else eller else if må alltid være en følge av et if statement. Kondisjoner skal være noe som returnerer true eller false. Operatorer som blir brukt i kondisjoner er:

x > y	gir bare true hvis x er mer enn y, ellers gir den false
x < y	gir bare true hvis y er mer enn x, ellers gir den false
x >= y	gir bare true hvis x er mer eller lik y, ellers gir den false
x <= y	gir bare true hvis x er mer eller lik y, ellers gir den false
x == y	gir bare true hvis x er lik y, ellers gir den false
x != y	gir bare true hvis x er ikke lik y, ellers gir den false
x && y	gir bare true hvis x og y er true, ellers gir den false
x || y	gir bare true hvis x eller y er true, ellers gir den false

Kapsulering av kondisjoner

For å kontrollere kombinasjonen av flere kondisjoner kan vi bruke parenteser () for å spesifisere hva som sjekkes først og putte 2 eller flere kondisjoner sammen.

Eksempel kode på kapsulering av kondisjoner i if statements

```
int x = 5;
int y = 2;
int z = 6;

// Vil bli oppfylt
// hvis ((x er større enn 10 og y er ikke 2) eller (z er 6))
if ((x > 10 && y != 2) || z == 6) {
        System.out.println(“kondisjon 1 ble oppfylt”);
}

// Vil ikke bli oppfylt
// hvis ((x er større enn 10) og (y er ikke 2 eller z er 6))
if (x > 10 && (y != 2 || z == 6)) {
        System.out.println(“kondisjon 2 ble oppfylt”);
}
```

Program output

```
kondisjon 1 ble oppfylt
```

Kapsulering av kondisjoner (TLDR)

Regelen for parenteser i en kondisjon er at det som skjer innerst inni parentesene vil skje først og så vil den gå ut ett etter ett parentes.

Looper

For Loops

Vanlige for loops bruker 3 statements splittet opp med **;** symbolet. Vanligvis definerer vi en variabel i den første statementen, setter kondisjonen for at loopen skal gjenta seg i det andre statementet og inkrementasjon i det 3 elementet.

Eksempel kode på vanlig for loop

```
for (int i = 0; i < 5; i++) {
        System.out.println(i);
}
```

Program output

```
0
1
2
3
4
```

For Each

For each loops er en mer elegant måte å loope gjennom en liste eller en klasse som er iterable. For each looper brukes vanligvis når man ikke har behov får å huske indeksen du er på. Du looper ved å spesifisere en typen på og navnet til variablen som holder på din nåverende plass i arrayen og ved å spesifisere array variabelen.

Eksempel kode på iterasjon med en for-each loop

```
int[] kostnader = new int[2];
kostnader[0] = 10;
kostnader[1] = 20;

// format på loop (type variabel : liste)
for (int verdi : kostnader) {
        System.out.println(verdi);
}
```

Program output

```
10
20
```

While Loops

En while loop er en loop som varer så lenge en kondisjon oppfylles. Denne kondisjonen fungerer likt som en if kondisjon så while loopen vil fortsette så lenge if kondisjonen oppfylles.

Eksempel kode på en while loop

```
int teller = 0;

While (teller < 3) {
        System.out.println(“fortsatt i loop”);
        teller++;
}
```

Program output

```
fortsatt i loop
fortsatt i loop
fortsatt i loop
```

Arrays

En array i java kan ses på som en samling av type. Det brukes til å kunne holde på flere verdier av den samme typen i en variabel. Arrays er kan **IKKE ENDRE STØRRELSE** i motsetning til en ArrayList. Størrelsen blir satt når du lager arrayen. Lengden på en array kan finnes ved å bruke **array.length**, dette funker bare hvis array variabelen holder på en array.

Regler for arrays

- Telling starter fra 0 siden arrays bruker indekser. Så første element i en array vil alltid være array[0].

- Størrelsen blir bestemt når du lager objektet ved “new Type[lengde]”.

- Arrayen kan byttes ut og lages på nytt. Ved å gjøre array = new Type[lengde] etter å ha tidligere definert den.

- Arrays kan lages av alle typer. Dette betyr også klasser du selv lager.

- Hvis du ikke gir en verdi til en av indeksene blir det til en null verdi. Hvis arrayen er av en primitiv type kan denne verdien bli noe annet.

Eksempel kode på array av int verdier

```
int[] kostnader = new int[3];
kostnader[0] = 400;
kostnader[1] = 350;
kostnader[2] = 199;

// første element: index 0
System.out.println(kostnader[0])

// andre element: index 1
System.out.println(kostnader[1])

// tredje element: index 2
System.out.println(kostnader[2])
```

Program output

```
350
199
400
```

Klasser

Hva er en klasse

En klasse er en brukerdefinert type som kan holde på variabler, metoder og mer. De blir brukt til å organisere data, strukturere prosjekter og organisere metoder og diverse infrastruktur.

Konstructor

En konstruktør er en metode som blir kalt når det lages en instanse/objekt av klassen. Konstuktører **MÅ VÆRE PUBLIC**, ellers vil den ikke bli kjørt når du lager en instanse/objekt av klassen. Konstruktøren **MÅ HA SAMME NAVN SOM KLASSEN**.

Eksempel kode på konstruktør til klasse

Filnavn: EksempelKlasse.java

```
public class EksempelKlasse {
        // Konstructor for klassen
        public EksempelKlasse() {
                System.out.println(“nytt objekt laget.”);
        }
}
```

Filnavn: TestProgram.java (Hovedprogram)

```
public class TestProgram {
        public static void main(String[] args) {
                // lager ny instanse/objekt av klassen
                // som automatisk kaller på konstruktøren.
                EksempelKlasse ek = new EksempelKlasse();
        }
}
```

Program output

```
nytt objekt laget
```

Instanse variabler

En instanse variabel er en variabel som er unikt for hver instanse/objekt, som er hvorfor den har instanse variabel. Dette gjøres ved å definere variabelen inne i klassen eller i en metode i klassen. En static variabel anses ikke som en instanse variabel, da er den heller bare static.

Metoder

En metode er en funksjon inne i en klasse. Metoder defineres som en variabel med modifiers, type og navn, men for å gjøre det til en metode **MÅ DEN HA PARENTES ETTER NAVNET**. Inne i parentesene kan man gi parametere som må brukes metoden kalles. Parameterene skrives ved som på formatet: (type var1, type var2 …). Hvis en metode gjøres static vil den kunne kalles uten å lage en instanse/objekt av klassen.

Extend (sub-klasse)

Å extende en klasse betyr å videreutbygge funskjonalitet. Når du extender en klasse beholder du dens metoder og funksjonalitet, men hvis du Overrider en av metodene så vil du bare kunne kalle på den forrige metoden ved å putte super foran. Hver klasse kan kan bare extende **EN** annen klasse.

Implementasjon (grensesnitt)

Implementering av interfaces/grensesnitt er brukt for å sørge for at funksjonalitet eksisterer. Interfaces er blueprints som sier hva av metoder og instanse variabler som **MÅ EKSISTERE** i klassen som implementerer det. Alt i en interface er abstrakt som betyr at det **MÅ OVERRIDES**. Klasser kan implementere et uendelig nummer av interfaces.

NB: inkluder @Override over metodene som interface får deg til å skrive.

Vanlige interfaces

- **Comparable<Type>**
  Krever at du lager en compareTo metode i klassen med et parameter av typen som du putter inne i <> ved siden av. Metoden skal returnere en int verdi som er -1 hvis parameteret er mindre i sammenheng med instansen compareTo kalles fra, 0 hvis den er lik og 1 hvis parameteret er mer enn instansen compareTo kalles fra.

- **Runnable**
  Krever at du lager en run metode som ikke returnerer noe. Denne interface brukes ofte i threads siden run metoden er det som kalles når du starter en thread. Snakker mer om threads lenger ned.

- **Iterable<Type>**
  Krever at du lager en iterator metode som returnerer en Iterator<Type>. Dette blir brukt til å spesifisere at du kan bruke en for-each loop på instanser av klassen. Snakker mer om iterators lenger ned.

Eksempel kode på implementasjon av comparable interfacen

Filnavn: Person.java

```
public class Person implements Comparable<Person> {
	public final alder;

	public Person(int alder) {
		thils.alder = alder;
	}

	public int compareTo(Person person) {
		// denne personen er yngre enn parameteret
		if (this.alder < person.alder) {
			return -1;
		};

		// denne personen er eldre enn parameteret
		else if (this.alder > person.alder) {
			return 1;
		}

		// denne personen like gammel parameteret
		return 0;
	}
}
```

Filnavn: TestProgram.java (Hovedprogram)

```
public class TestProgram {
	public static void main(String[] args) {
		Person person1 = new Person(18);
		Person person2 = new Person(17);

		// sammenligner de 2 instansene av person
		System.out.println(person1.compareTo(person2));
	}
}
```

Program output

```
1
```

Static referanse

At noe i java er static betyr at de er like for alt som accesser det. Så en static variabel vil være lik for alle uansett hva som leser eller endrer på det. En static metode i en klasse er metoder som kan kalles på uten å lage et objekt av klassen. Dette betyr at man **ikke kan bruke this for static variabler eller metoder**, de må istedenfor brukes gjennom klassen eller uten this fra klassen.

Eksempel kode på static og ikke static metoder

Filnavn: EksempelKlasse.java

```
public class EksempelKlasse {
	public static void snakk() {
		System.out.println(“Hello World”);
	}

	public void snakkUtenStatic() {
		// siden snakk metoden er static kan
		// vi ikke bruke this får å kalle på den.
		snakk();
	}
}
```

Filnavn: TestProgram.java (Hovedprogram)

```
public class TestProgram {
	public static void main(String[] args) {
		EksempelKlasse ek = new EksempelKlasse();

		// static kall på metode uten objekt
		EksempelKlasse.snakk();

		// metode inne i klassen kaller på en static måte
		ek.snakkUtenStatic()
	}
}
```

Program output

```
Hello World
Hello World
```

Hva er this

This er en referanse til det objektet koden skjer i. This blir brukt generelt for å holde styr på hvor variabler hører til og hvis du har et gitt parameter som heter det samme som en instanse variabel.

Eksempel kode på bruk av this

```
public class EksempelKlasse {
        private int verdi;

        // Konstruktør for klassen
        public EksempelKlasse(int verdi) {
                this.verdi = verdi;
        }
}
```

NB FOR THIS OG STATIC

This kan ikke brukes for static variabler eller metoder. This kan bare bli brukt i kode til en **ikke static** metode for objekter som er laget av en klasse.

Hva er super

Super brukes når klasser extendes. Super vil da være en peker til klassen som du extender slikt at du fortsatt kan bruke metoder som du har Overridet. Super kan også brukes for å beholde koden for konstruktoren fra den klassen du ekstender, men hvis du skal bruke superen sin konstruktør så må det være det første du kaller.

Eksempel kode på bruk av super

Filnavn: BaseKlasse.java

```
public class BaseKlasse {
        public BaseKlasse () {
                System.out.println(“BaseKlasse konstrukt”);
        }

        public String toString () {
                return “base klasse toString”;
        }
}
```

Filnavn: AndreKlasse.java

```
public class AndreKlasse extends BaseKlasse {
	public AndreKlasse () {
		// for å bruke BaseKlasse sin konstruktør må jeg
		// bruke super. Hvis konstruktøren til BaseKlasse brukte
		// parametre måtte de inkluderes i parantesene her
		super();

		System.out.println(“AndreKlasse konstrukt ”);
	}

	public String toString() {
		return super.toString() + “\nandre klasse toString”;
		System.out.println(ak);
	}
}
```

Filnavn: TestProgram.java (Hovedprogram)

```
public class TestProgram {
	public static void main(String[] args) {
		AndreKlasse ak = new AndreKlasse();
		System.out.println(ak);
	}
}
```

Program output

```
BaseKlasse konstrukt
AndreKlasse konstrukt
Base klasse toString
andre klasse toString
```

toString metoden

En vanlig metode å inkludere i klasser er en toString metode som gjør at du kan bestemme hva som printes ut når du bruker System.out.println på en instanse/objekt av klassen.

Abstraksjon

Abstraksjon brukes i java for å kontrollere at klasser inneholder visse funksjonaliteter. Abstrakte metoder kan ikke inneholde kode, den må overrides av en ikke abstrakt metode og så kjøres.

- Abstrakte klasser er klasser som ikke kan lages instanser av. For å bruke denne må du lage en ny klasse som extender den abstrakte klassen. Dette brukes for å sørge for at klasser inneholder visse funksjonaliteter på forhånd.

- Abstrakte metoder eller instansevariabler er metoder eller variabler som må lages eller overrides i en klasse som skal kunne instanseres. Dette blir ofte brukt i interfaces/grensesnitt for å sørge for at en klasse inneholder visse metoder eller instanse variabler.

Exceptions

Det finnes diverse exceptions som kan skje mens programmet kjører. De fleste exceptions extender noen andre exceptions men øverst på exception hierarkiet ligger Exception klassen som kan brukes for å dekke alle mulige feil.

Eksempel kode på catching av exceptions

```
try {
        int[] verdi = new int[2];
        System.out.println(verdi[10]);
}

catch (Exception e) {
        e.printStackTrace();
}
```

Program output

```
java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 2
    at Main.main(Main.java:5)
```

Man kan catche flere exception på en gang ved å bruke | operatoren

```
catch (ArrayIndexOutOfBoundsException | IOException e) {
        …
}
```

Generic Type

En generisk type er en hvilken som helst type som vanligvis vises med <T>. Dette brukes ofte i lenkelister som dekkes i en av de neste sidene.

Nyttige klasser

Nyttige klasser som vi har brukt i obliger.

ArrayList

En arraylist er en mutable array som betyr at den ikke har et fast antall elementer. Dette er nyttig for å ha en liste som du legger til og fjerner elementer fra konstant. Arraylister må spesifisere typen når du lager en instanse av dem. For å bruke ArrayList klassen må importeres fra import java.util.ArrayList. Eksempel på en ArrayList av String verider.

Eksempel kode på arraylist av string verdier
Import java.util.ArrayList; ArrayList<String> string_beholder = new ArrayList<>();

Viktige metoder til ArrayList

- **ArrayList.add(Type obj)** → void
  Legger til et element til arraylisten.

- **ArrayList.remove(Type obj eller int indeks)** → Type
  Fjerner et element som matcher det du gir som parameter eller elementet på posisjonen til en indeks du spesifiserer som en int verdi. Denne metoden returnerer det elementet du fjernet.

- **ArrayList.size()** → int
  Returnerer størrelsen på arraylisten.

- **ArrayList.set(int indeks, Type obj)** → Type
  Ender på et element på indeks parameteret til obj parameteret. Denne metoden returnerer det som tidligere var på indeksen.

- **ArrayList.clear()** → void
  Tømmer arraylisten for alle elementer.

- **ArrayList.contains(Type obj)** → boolean
  sjekker om et objekt eksisterer i arraylisten. Gir true for at den eksisterer og false for ikke.

- **ArrayList.indexOf(Type obj)** → int
  Returnerer indeks posisjonen til obj parameteret.

- **ArrayList.sort()** → void
  Sorterer arraylisten. Dette krever enten at elementene i arraylisten er nummer verdier eller at typen til elementene har en compareTo.

Iterasjon av ArrayList

Eksempel kode på arraylist iterasjon

Import java.util.ArrayList;
 	
ArrayList<String> liste = new ArrayList<>();
 	
liste.add(“bil”);
liste.add(“ball”);
 	
for (String element : liste) {
  System.out.println(element);
}

Program output
bil ball

HashMap

En hashmap er som en python dict. Det er et register av nøkler og verdier. Hver nøkkel har en verdi som hører til seg så “cars” kan ha en verdi av 50 mens “planes” kan ha en verdi av 32. Et hashmap krever 2 typer når den lages, En for nøkkel typen og en for verdi typen. For å bruke hashmap må det importeres fra java.util.HashMap.
Eksempel på et hashmap av string nøkler og int verdier

Eksempel kode på hashmap med string nøkler og int verdier
Import java.util.HashMap; HashMap<String, int> register = new HashMap<>();

Viktige metoder til HashMap

- **HashMap.put(Type key, Type value)** → void
  Lagrer value parameteret ved nøkkel parameteret.
  NB: hvis nøkkelen og verdien eksisterte blir den overskrevet.

- **HashMap.get(Type key)** → Type
  Henter en verdi som ligger ved nøkkel parameteret.

- **HashMap.remove(Type key)** → Type
  Fjerner en nøkkel og dens verdi. Verdien som var med nøkkelen blir returnert.

- **HashMap.size()** → int
  Returnerer antall nøkler med verdier i registeret.

- **HashMap.clear()** → void
  Fjerner alle nøkler og verdier

- **HashMap.keySet()** → Set<Type>
  Returnerer et set av nøklene i registere som kan itereres gjennom.

Iterasjon av HashMaps

Eksempel kode på iterasjon av hashmaps

```
HashMap<String, int> register = new HashMap<>();

register.add(“cars”, 34);
register.add(“planes”, 11);

// iterering teknikk 1
for (String key : register.keySet()) {
        System.out.println(key);
        System.out.println(register.get(key));
        System.out.println();
}

// iterering teknikk 2
// ! krever også import java.util.Map; !
for (Map.Entry<String, int> entry : register.entrySet()) {
        System.out.println(entry.getKey());
        System.out.println(entry.getValue());
        System.out.println();
}
```

Program output

```
cars
34

planes
11

cars
34

planes
11
```

Lenkelister

En lenkeliste er en liste av objekter hvor hvert objekt peker på den neste i lenken. I en lenkeliste har du en start node som peker på den neste noden, og den neste noden peker på den neste noden osv. Hvis en node ikke har en etter seg er neste null. Eksempel kode på en lenkeliste tatt fra oblig 3

Eksempel kode på en lenkeliste

```
public class Lenkeliste<T> {
	protected Node start = null;

	class Node {
		public Node neste = null;
		public T data;

		public Node(T x) {
			this.data = x;
		}
	}

	public void leggTil(T x) {
		// lag nyNode av x
		Node nyNode = new Node(x);

		// hvis start er tomt sett nyNode på start
		if (this.start == null) {
			this.start = nyNode;
		}

		else {
		// iterer til node ikke har en neste
		// og så sett nyNode på node.neste
		Node node = this.start;
		while (node.neste != null) {
			node = node.neste;
		}
		node.neste = nyNode;
		}
	}

	public T hent() {
		// returner dataen til første node hvis den ikke er null
```

```
            if (this.start != null) {
                return this.start.data;
            }
            return null;
        }

        public T fjern() {
            // hvis start ikke er null fjern start noden
            // og sett start.neste til start
            if (this.start != null) {
                T node_data = hent();
                this.start = this.start.neste;
                return node_data;
            }

            return null;
        }
}
```

Iteratorer

En iterator er et objekt hvor man kan bruke en for-each loop for å gå gjennom hvert element. En iterator må implementerer Iterator<Type> interface mens en klasse som skal gi en iterator implementerer Iterable<Type> interface. Ved å bruke koden i forrige eksempel kan vi lage en iterable klasse. Vi extender klassen fra lenkeliste eksempelet her.

Eksempel kode på en lenkeliste iterator

```
import java.util.Iterator;

public class Liste<T> extends Lenkeliste<T> implements Iterable <T> {
    public class ListeIterator implements Iterator<T> {
        Node current = start;

         @Override
        public boolean hasNext() {
            // sjekke om en neste node eksisterer
            return (current != null);
        }
```

```
        @Override
        public T next() {
            // henter ut den nåværende noden
            Node node = current;

            // flytter seg til neste node;
            current = current.neste;

            // returnerer dataen fra noden
            return node.data;
        }
    }

    public Iterator<T> iterator() {
        // lager og returnerer en iteratoren instanse/objekt
        return new ListeIterator();
    }
}
```

Threads

En thread er en kode som kan kjøre separat fra main programmet. Threads brukes for å øke hastighet på et program og dele belastning av utregninger. For å lage en thread må du ha en klasse som implementerer runnable. Dette er fordi når en thread startes er det run metoden som kjøres. Eksempel på 3 threads som kjører samtidig.

Eksempel kode på laging og kjøring av threads

Filnavn: **LoopTraad.java**

```
public class LoopTraad implements Runnable {
    @Override
    public void run() {
        // printer alle tall fra 0 til 99
        for (int i = 0; i < 100; i++) {
            System.out.println(i);
        }
    }
}
```

Filnavn: **TestProgram.java (Hovedprogram)**

```
public class TestProgram {
	public static void main(String[] args) {
		// lager 3 threads
		Thread t1 = new Thread(new LoopTraad());
		Thread t2 = new Thread(new LoopTraad());
		Thread t3 = new Thread(new LoopTraad());

		// når join metoden til threads kalles MÅ man
		// catch hvis det skjer InterruptedExceptions
		try {
			t1 = new Thread(new LoopTraad());
		}

		// trenger ikke ha noe som skjer hvis brukeren
		// stopper så jeg lar catchen sin body være tom
		catch (InterruptedException e) {}
	}
}
```

Program output

```
0
0
0
1
1
1
2
2
2
…
```

Thread locks

En thread lock er en lås som bare en kan holde på om gangen. Se på det som en sirkel av mennesker hvor bare den som har puten får lov til å snakke. Dette brukes får å unngå konflikt i at 2 threads redigerer på samme objekt samtidig. For å bruke en thread lock må vi importere disse klassene:

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
```

Eksempel kode på threads med en thread lock

Filnavn: LoopTraad.java

```
import java.util.concurrent.locks.Lock;

public class LoopTraad implements Runnable {
	private Lock thread_lock;

	public LoopTraad (Lock thread_lock) {
		// lagrer thread låsen for å kunne brukes
		This.thread_lock = thread_lock;
	}

	@Override
	public void run() {
		this.thread_lock.lock()

		// koden som skal kjøres mens denne threaden
		// holder på thread locken.
		try {
			// printer alle tall fra 0 til 99
			for (int i = 0; i < 100; i++) {
				System.out.println(i);
			}
		}

		// låser opp låsen selv om noe går galt
		finally {
			this.thread_lock.unlock();
		}
	}
}
```

Filnavn: TestProgram.java (Hovedprogram)

```
public class TestProgram {
	public static void main(String[] args) {
		// lag en lås for trådene å bruke
		Lock thread_lock = new ReentrantLock();
```

```
            // lager 3 threads og gir dem låsen som parameter
            Thread t1 = new Thread(new LoopTraad(thread_lock));
            Thread t2 = new Thread(new LoopTraad(thread_lock));
            Thread t3 = new Thread(new LoopTraad(thread_lock));

            // når join metoden til threads kalles MÅ man
            // catch hvis det skjer InterruptedExceptions
            try {
                t1.join();
                t2.join();
                t3.join();
            }

            // trenger ikke ha noe som skjer hvis brukeren
            // stopper så jeg lar catchen sin body være tom
            catch (InterruptedException e) {}
        }
}
```

Program output

```
0
1
2
…
98
99
0
1
2
…
98
99
0
1
2
…
98
99
```

Synchronized

I Java er synchronized et nøkkelord som brukes for å sikre at en blokk av kode eller en metode kun kan utføres av én tråd om gangen. Dette er nyttig for å hindre uventede resultater og feil når flere tråder prøver å utføre samme blokk av kode samtidig. Dette er spesielt relevant når det arbeides med delte ressurser.

Her er et eksempel som viser hvordan synchronized kan brukes:

Eksempel kode på threads med en synchronized metode

Filnavn: .java

```
ublic class SynchronizedExample {

    private int counter = 0;

    // Metoden som er synkronisert
    public synchronized void increment(int nr) {
        // Denne blokken er kun tilgjengelig for én tråd om gangen
        counter++;
        System.out.println("Thread" + nr +" Counter: " + counter);
    }

    public static void main(String[] args) {
        SynchronizedExample example = new SynchronizedExample();

    // Oppretter to tråder ved å gi dem en instans av Runnable-
//objektet
        Thread thread1 = new Thread(new MyRunnable(example, 1));
        Thread thread2 = new Thread(new MyRunnable(example, 2));

        // Starter trådene
        thread1.start();
        thread2.start();

        try {
            // Venter på at begge trådene skal fullføre
```

```
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

// Runnable-implementering
class MyRunnable implements Runnable {

    private SynchronizedExample example;
    private int nr;

    public MyRunnable(SynchronizedExample example, int nr) {
        this.example = example;
        this.nr = nr;
    }

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            example.increment(nr);
            try {
                Thread.sleep(100); // Simulerer litt forsinkelse
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Program output
Thread1 Counter: 1 Thread2 Counter: 2 Thread1 Counter: 3 Thread2 Counter: 4 Thread1 Counter: 5 Thread2 Counter: 6 Thread1 Counter: 7 Thread2 Counter: 8 Thread1 Counter: 9 Thread2 Counter: 10

Gui

Fuck GUI og alt som er design relatert