

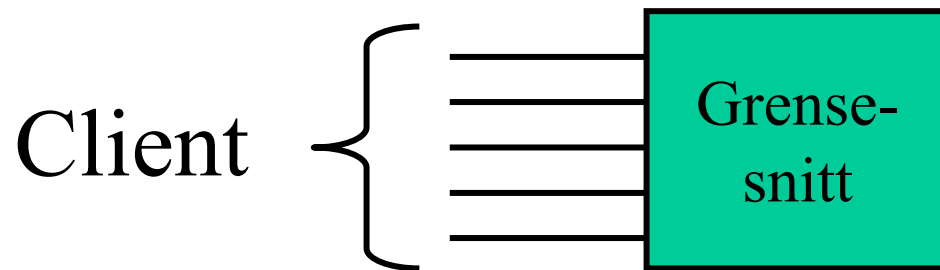
Læringsmål for forelesningen

- Objektorientering
 - Funksjonelle grensesnitt
- Java-programmering
 - Java 8-funksjoner
- Vi skal myse litt på *funksjonell programmering*
- Det blir mer om det neste uke
- Husk «kort forklart» og [wikien!](#)



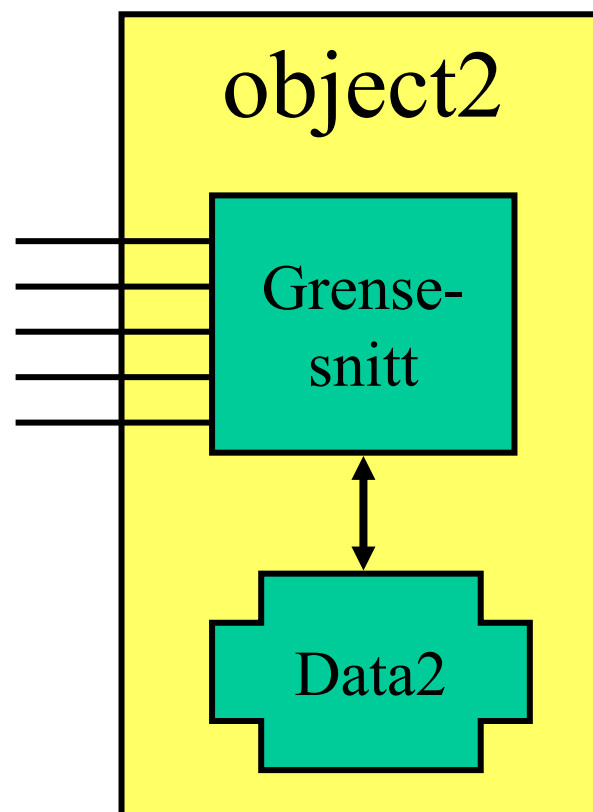
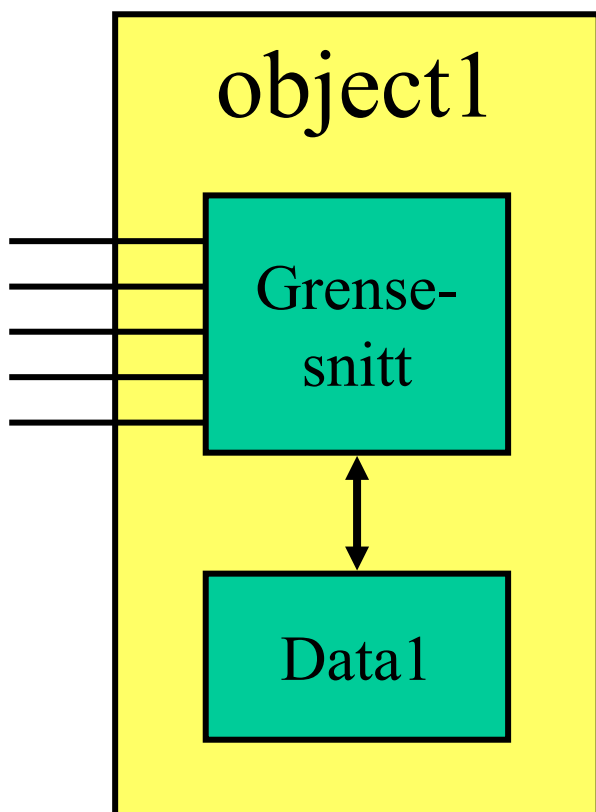


Grensesnitt, klasse med
metodedeklarasjoner,
uten tanke på innhold forøvrig



*Sett fra koden som bruker en klasse,
så er metodegrensesnittet alt en
trenger for å sikre gyldig kode.*

Ulik innmat, samme grensesnitt



Typisk ”funksjonelt” grensesnitt: Comparator

- Sortering er en generell funksjon, som kan deles i to deler:
 - den generelle *algoritmen*, som det finnes mange varianter av
 - *sorteringsrekkefølgen*, som er spesifikk for anvendelsen
- Sorteringsrekkefølgen er interessant å kunne tilpasse
 - spesifikk for typen objekt som skal sorteres, f.eks. **Person**
 - spesifikk for anvendelsen, f.eks. kan **Person**-objekter sorteres på navn, alder, høyde, vekt osv.
- Sorteringsrekkefølgen er fanget opp av grensesnittet **java.util.Comparator**
 - har én metode, **compare(o1, o2)**, som avgjør hvilke av to objekter som skal være først etter sorteringen
 - en kan tenke på grensesnittet som definisjonen av en *type* funksjon, og implementasjonen som en konkret *funksjon*

Comparator<T>

- Én metode:
 - `int compare(T t1, T t2);`
- Sammenligner to objekter av samme type T
- Sier om det første er mindre, like eller større enn det andre
- Returverdi sier om
 - o1 er mindre enn o2 => returverdi < 0
 - o1 er lik o2 => returverdi == 0
 - o1 er større enn o2 => returverdi > 0

Eksempel: NameComparator

- Implementerer sortering(rekkefølge) på for- og etternavn
- Utnytter at String har en egen compareTo-metode for alfabetisk sammenligning (nesten ihvertfall):

```
public class NameComparator implements Comparator<Person> {  
    @Override  
    public int compare(Person p1, Person p2) {  
        String s1 = p1.getFamilyName() + ", " + p1.getGivenName();  
        String s2 = p2.getFamilyName() + ", " + p2.getGivenName();  
        return s1.compareTo(s2);  
    }  
}
```

- Denne brukes nå i en sortering slik:
`Collections.sort(personer, new NameComparator());`
- Hver gang vi trenger en ny slik compare-funksjon, så må vi lage en ny, navngitt klasse, selv om vi bare trenger én instans der og da...

Typisk ”funksjonelt” grensesnitt: Predicate

- Søk/filtrering kan deles i to deler:
 - den generelle *algoritmen*, som går gjennom en datastruktur på jakt etter et eller flere objekter som tilfredsstillter visse krav
 - *kriteriet* for (ut)valg, som er spesifikk for anvendelsen
- Kriteriet er interessant å kunne tilpasse
 - spesifikk for typen objekt som det søkes/filtreres på, f.eks. **Person**
 - spesifikk for anvendelsen, f.eks. **Person**-objekter med en bestemt alder, høyde, vekt osv.
- Kriteriet kan fanges opp av grensesnittet **java.util.function.Predicate**
 - har én metode, **test(objekt)**, som avgjør om et objekt tilfredsstillter kriteriene
 - en kan tenke på grensesnittet som definisjonen av en *type* funksjon, og implementasjonen som en konkret *funksjon*

Predicate<T>

- Én metode:
 - `boolean test(T o);`
- Eksempel: Sjekk om en person har et gitt navn:

```
public class GivenNameTester implements Predicate<Person> {  
  
    private String name;  
  
    public GivenNameTester(String name) {  
        this.name = name;  
    }  
  
    @Override  
    public boolean test(Person p) {  
        return name.equals(p.getGivenName());  
    }  
}
```

PersonReg

```
private Collection<Person> persons = new ArrayList<>();
```

```
public Person findFirst(Predicate<Person> tester) {
    for (Person person : persons) {
        if (tester.test(person)) {
            return person;
        }
    }
    return null;
}
```

```
public Collection<Person> findAll(Predicate<Person> tester) {
    Collection<Person> result = new ArrayList<>();
    for (Person person : persons) {
        if (tester.test(person)) {
            result.add(person);
        }
    }
    return result;
}
```

Viktige funksjonelle grensesnitt

- `Predicate<T>` - `boolean test(T)` // tester T
 - eksempel
 - søk etter objekt i en datastruktur, som tilfredsstiller visse krav
 - klassen som “eier” datastrukturen gjør jobben med å gå gjennom strukturen
- `Consumer<T>` - `accept(T)` // bruker T
 - eksempel
 - gjøre noe på alle objektene i en datastruktur, f.eks. printe dem
 - klassen som “eier” datastrukturen gjør jobben med å gå gjennom strukturen
- `Supplier<T>` - `T get()` // gir ut en T
 - eksempel
 - produsere nye instanser av en type object (en såkalt *factory*)
 - generere tilfeldige tall for simulering
 - matematisk konstant i kalkulator

Viktige funksjonelle grensesnitt

- `Function<T, R>` - `R apply(T t)`
 - eksempel
 - hente ut data lagret i objekt, som skal sammenlignes med en gitt verdi i et søk
- `BiFunction<T1, T2, R>` - `R apply(T1 t1, T2 t2)`
 - som `Function`, men tar inn to argumenter
- `BinaryOperator<T>`
 - `T apply(T t1, T t2)` // returnerer `T` basert på to `T`-er
 - eksempel
 - matematisk operasjon i kalkulator eller tolker for uttrykk, som `+` og `*`
- `UnaryOperator<T>`
 - `T apply(T t)` // returnerer `T` basert på `T`
 - eksempel
 - matematisk operasjon eller funksjon, som `-` (negasjon), kvadratroten og sinus

Standard funksjoner på samlinger av objekter

- **Collection<R> map(Collection<T>, Function<T, R>)**
 - returnerer ny liste med avledete verdier, beregnet fra samlingens innhold vha. funksjon
 - kalles også **collect**
- **T reduce(Collection<T>, BinaryOperator<T>)**
 - kombinerer elementene i samlingen, med den gitte operatoren
 - eksempel
 - summere alle verdiene i en liste
- **R fold(Collection<T>, R r, BiFunction<R, T, R>)**
 - mer generell variant av reduce, hvor akkumulatoren er av en annen type enn elementene, og **r** brukes som startverdi
 - eksempel
 - lage en komma-separert String av alle elementene

Grensesnitt-instanser (anonyme indre klasser)

- Det er ofte upraktisk å måtte lage nye implementasjoner av grensesnitt som egne klasser, siden de ofte er
 - lite relevante å gjenbruke – brukes bare ett sted
 - forstås best der de brukes – navnet sier ikke alltid nok
- Med en såkalt *anonym indre klasse*, så kan en legge koden for klassen direkte inn der den brukes:

```

Collections.sort(personer, new Comparator<Person>() {
    @Override
    public int compare(Person p1, Person p2) {
        String s1 = p1.getFamilyName() + ", " + p1.getGivenName();
        String s2 = p2.getFamilyName() + ", " + p2.getGivenName();
        return s1.compareTo(s2);
    }
});

```

- Lager en instans som implementerer grensesnittet på direkten!
- Men dette er også tungvint, når det bare er snakk om én funksjon...

Spesialsyntaks for instanser av *funksjonelle* grensesnitt

- Funksjonelle grensesnitt, altså grensenitt med én metode, har mange anvendelser, som bygger på en egen *funksjonell* programmeringsstil
- Java 8 (vi bruker 21 nå) sin fremste “nyvinning” er:
 - en rekke generelle funksjonelle grensesnitt, f.eks. Predicate
 - spesialsyntaks for instanser av funksjonelle grensesnitt
 - klasser og metoder som bruker funksjonelle grensesnitt

- Spesialsyntaks:

```
Collections.sort(personer, (p1, p2) -> {
    String s1 = p1.getFamilyName() + ", " + p1.getGivenName();
    String s2 = p2.getFamilyName() + ", " + p2.getGivenName();
    return s1.compareTo(s2);
});
```

- Vi slipper å
 - navngi grensesnittet, siden det kan *utledes* fra typen som forventes
 - navngi metoden, siden det kan *utledes* fra grensesnittet
 - oppgi argument-typer, siden det kan *utledes* fra typen(e) som forventes

Spesialsyntaks forts.

- Spesialsyntaks:
 - parameterliste, uten typer (når de kan utledes)
 - parameterlista trenger ikke (...) rundt, dersom det bare er ett parameter
 - -> **<uttrykk>** når metoden er et enkelt uttrykk som returneres
 - -> { **<setninger>** } når metoden er mer enn et enkelt uttrykk
- Collections.sort(**personer**, (**p1**, **p2**) -> {
 return p1.getFamilyName().compareTo(p2.getFamilyName());
});
VS.
Collections.sort(**personer**, (**p1**, **p2**) ->
 p1.getFamilyName().compareTo(**p2**.getFamilyName()
);

Spesialsyntaks forts.

- Metoder med riktige typer kan brukes direkte

- **Klasse::metode** refererer til metode, f.eks. :

UnaryOperator<Double> **unOp** = Math::sin; tilsvarer

UnaryOperator<Double> **unOp** = (d) -> Math.sin(d);

Function<Person, String> **fun** = Person::toString(); tilsvarer

Function<Person, String> **fun** = (p) -> p.toString();

- **objekt::metode** refererer til metode kalt på objekt, f.eks. :

Consumer<Object> **cons** = **System.out**::println; tilsvarer

Consumer<Object> **cons** = (o) -> { **System.out**.println(o); };

- **Klasse::new** refererer til kall på konstruktør, f.eks. :

Supplier<Person> **sup** = Person::new; tilsvarer

Supplier<Person> **sup** = () -> new Person();

Function<String, Person> **fun** = Person::new; tilsvarer

Function<String, Person> **fun** = (s) -> new Person(s);

Eksempel: RPN-kalkulator

- RPN-kalkulator som støtter
 - binære operatorer som +, -, * og /
 - unære operatorer som – og kvadratroten
 - konstanter, som π og e
- RPN = Reverse Polish Notation
 - operander puttes på en “stack”
 - operatorer tar et antall elementer av stacken, beregner resultatet og putter det tilbake på stacken
 - Mye enklere å skrive kode for å parse (lese og tolke) enn for vanlig matte-notasjon.
 - les mer på wikipedia: https://en.wikipedia.org/wiki/Reverse_Polish_notation
- Tre typer operatorer
 - **BinaryOperator<Double>** – tar inn 2 verdier (og beregner resultatet), f.eks. + og *
 - **UnaryOperator<Double>** – tar inn 1 verdi, f.eks. sin, cos og kvadratroten
 - **Supplier<Double>** – tar inn 0 verdier, f.eks π og e

Hver operator-type har sin metode

```
// computes one value from two arguments
public interface BinaryOperator<T> {
    public T apply(T op1, T op2);
}
```

```
// computes one value from one argument
public interface UnaryOperator<T> {
    public T apply(T op);
}
```

```
// computes one value from no arguments
public interface Supplier<T> {
    public T get();
}
```

Operator-tabeller

- For hver operator-type lager vi en “dictionary” eller tabell over operator-navn og –objekt

```
private Map<String, BinaryOperator<Double>> binaryOperators = ...  
private Map<String, UnaryOperator<Double>> unaryOperators = ...  
private Map<String, Supplier<Double>> constants = ...
```

- Disse må fylles med instanser av de ulike operator-grensesnittene. Siden grensesnittene er funksjonelle, så kan man bruke den enkle funksjonssyntaksen

```
binaryOperators.put("+", (d1, d2) -> d1 + d2);  
binaryOperators.put("-", (d1, d2) -> d2 - d1);  
binaryOperators.put("*", (d1, d2) -> d1 * d2);  
binaryOperators.put("/", (d1, d2) -> d2 / d1);
```

```
unaryOperators.put("√", (d) -> Math.sqrt(d));
```

```
constants.put("π", () -> Math.PI);
```

Funksjonssyntaks

```
binaryOperators.put("+", (d1, d2) -> d1 + d2);
```

- Betyr “lage en instans av en ikke navngitt (anonym) klasse som implementerer grensesnittet som passer her
- Tilsvarende

```
binaryOperators.put("+", new BinaryOperator<Double>() {
    @Override
    public Double apply(Double d1, Double d2) {
        return d1 + d2;
    }
});
```

Syntaks for funksjonelle grensesnitt

- Eclipse har funksjon for å konvertere frem og tilbake: **Quick Assist** med Command-1

```
new BinaryOperator<Double>() {  
    @Override  
    public Double apply(Double op1, Double op2) {  
        return op1 + op2;  
    }  
}
```



```
(op1, op2) -> op1 + op2
```

Hovedprogrammet

- Avgjør om input er operand (**hasNextDouble()**) eller operator
- Hvis operator, sjekk (**containsKey**) og slå opp (**get**) i tabellene (etter tur) og bruk objektet en finner (**compute**) iht. antall operander den trenger

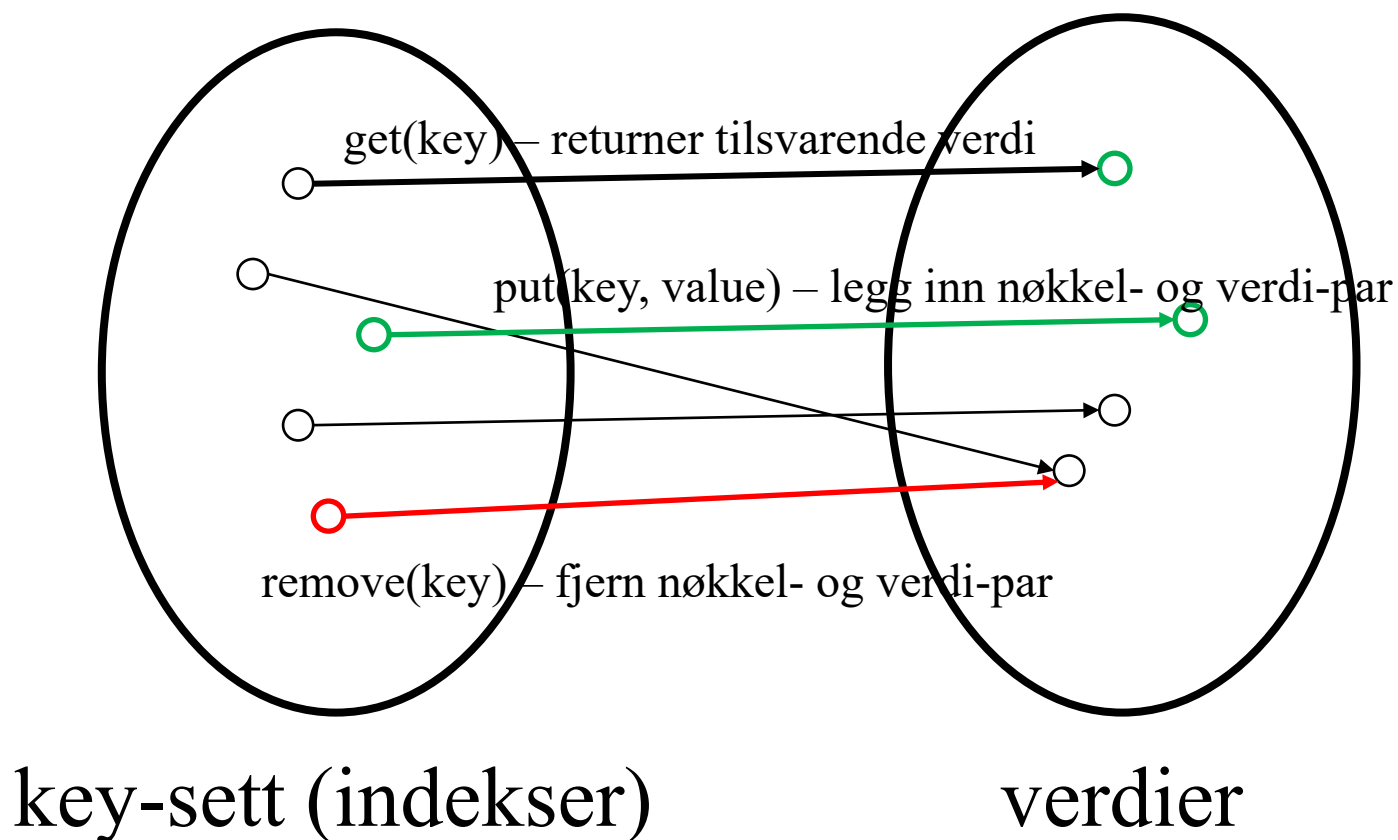
```
Scanner scanner = new Scanner(System.in);
while (scanner.hasNext()) {
    if (scanner.hasNextDouble()) {
        double d = scanner.nextDouble();
        operandStack.push(d);
    } else {
        String op = scanner.next();
        if (binaryOperators.containsKey(op)) {
            double d1 = operandStack.pop(), d2 = operandStack.pop();
            BinaryOperator<Double> operator2 = binaryOperators.get(op);
            operandStack.push(operator2.apply(d1, d2));
        } else if (unaryOperators.containsKey(op)) {
            double d1 = operandStack.pop();
            UnaryOperator<Double> operator1 = unaryOperators.get(op);
            operandStack.push(operator1.apply(d1));
        } else if (constants.containsKey(op)) {
            Supplier<Double> operator0 = constants.get(op);
            operandStack.push(operator0.get());
        }
    }
}
System.out.println(operandStack);
}
scanner.close();
```

Map-grensesnittet

- Grensesnitt tilsvarende Python sin “dictionary”-mekanisme
- Tabell hvor indeksen er “hva som helst”
- Fire essensielle metoder
 - **containsKey(key)** – er det lagt inn noen verdi for **key**
 - **get(key)** – hent ut verdien som er lagt inn for **key**
 - **put(key, value)** – legg inn **value** for **key**
 - **remove(key)** – fjern innslag for **key**
- Implementeres av bl.a. HashMap

Illustrasjon av Map

`containsKey(key)` – finnes key her?



Læringsmål for forelesningen

- Objektorientering
 - Grensesnitt
- Java-programmering
 - interface-konstruksjonen
 - implements-nøkkelordet

