NTNU

Norwegian University of Science and Technology

IDATT2001 – Programmering 2

Functional Programming

Læringsutbytte

- Functional Programming
  - Funksjonell grensesnitt
  - Lokale klasser
  - Litt om anonyme klassser (Kap. 13.8.2)
  - Lambdas
  - forEach metoden til collections
- Streams
- Pipelines

Lambda-uttrykk og funksjonelle grensesnitt

- Java 8 introduserer såkalte lambda-uttrykk
- Kan være svært nyttige til å forenkle kode og utføre funksjonell programmering i Java.
- Bruk av lambda fører ofte til mye mindre kode, og få for-løkker.
- For å kunne skjønne lambda-uttrykk bør vi først nevne litt om **funksjonelle grensesnitt**

Anonyme klasser

- En anonym klasse deklareres og instansieres i et uttrykk (samtidig)
- De er som lokale klasser bortsett fra at de ikke har et navn
- Er enten en subklasse av en annen klasse eller så implementerer den et (og bare ett) grensesnitt
- Lokale klasser er klasser som er definert i en annen klasse (blokk). Du finner typisk lokale klasser definert i kroppen av en metode.

Syntax of a local class

```
public class HelloWorldAnonymousClasses {
     public void sayHello() {
          class EnglishGreeting implements HelloWorld {
               String name = "world";

               public void greet() {
                    greetSomeone("world");
               }

               public void greetSomeone(String someone) {
                    name = someone;
                    System.out.println("Hello " + name);
               }
          }

          HelloWorld englishGreeting = new EnglishGreeting();

          englishGreeting.greet();
     }

     public static void main(String... args) {
          HelloWorldAnonymousClasses myApp = new HelloWorldAnonymousClasses();
          myApp.sayHello();
     }
}
```

Syntax of an anonymous class

```
public class HelloWorldAnonymousClasses {
    public void sayHello() {
        HelloWorld spanishGreeting = new HelloWorld() {
            String name = "mundo";

            public void greet() {
                greetSomeone("mundo");
            }

            public void greetSomeone(String someone) {
                name = someone;
                System.out.println("Hola, " + name);
            }
        };
        spanishGreeting.greet();
    }

    public static void main(String... args) {
        HelloWorldAnonymousClasses myApp = new HelloWorldAnonymousClasses();
        myApp.sayHello();
    }
}
```

| DEMO: FL02.eks.anonymous.HelloWorldAnonymousClasses

Funksjonelle grensesnitt

Definisjon: Er grensesnitt som har kun én (abstrakt) metode definert.

Eksempel:

```
public interface DoubleValueComputer {
    public double compute(double x, double y);
}
```

Grensesnittet over er et funksjonelt grensesnitt, for det har bare én metode, nemlig **compute**

Basic syntaks av Lambda

Normal (eksempel)

```
DoubleValueComputer adderNormal = new DoubleValueComputer() {
     @Override
     public double compute(double x, double y) {
     return x + y;
     }
};
```

Lamda (eksempel)

```
DoubleValueComputer adderLambda = (x, y) -> x + y;
```

Direkte implementasjon (anonym klasse)

```
DoubleValueComputer adder = new DoubleValueComputer() {
    @Override
    public double compute(double x, double y) {
        return x + y;
    }
};

DoubleValueComputer multiplier = new DoubleValueComputer() {
    @Override
    public double compute(double x, double y) {
        return x * y;
    }
};
```

Med lambda

```
DoubleValueComputer adder = (x, y) -> x + y;
DoubleValueComputer multiplier = (x, y) -> x * y;
```

| DEMO: FL02.eks.lambda.doublevalue

| DEMO : FL02.eks.lambda.numeric

Generisk funksjon

```
interface MyGeneric<T> {
     T compute(T t);
}
```

```
MyGeneric<String> reverse = (str) -> {
  String result = "";

  for (int i = str.length() - 1; i >= 0; i--)
     result += str.charAt(i);
     return result;
};
```

```
MyGeneric<Integer> factorial = (Integer n) -> {
  int result = 1;

  for (int i = 1; i <= n; i++)
     result = i * result;

     return result;
};
```

```
out(reverse.compute("Lambda Demo"));
out(factorial.compute(5));
```

| DEMO: FL02.eks.lambda.generic.IMyGeneric

| DEMO: FL02.eks.lambda.generic.IMyGeneric2

Innebygde funksjonelle grensesnitt i Java 8

For å kunne utnytte kraften i lambda til det fulle, er de oftest brukte funksjonelle grensesnittene implementert i Java, så man slipper å definere dem selv

- Predicate-grensesnittet har metoden test, som tar inn et objekt av hvilken som helst type (det vil si Object) som argument, og returnerer en boolean.
- Consumer-grensesnittet har metoden accept, som tar inn et objekt av hvilken som helst type (det vil si Object) som argument, og returnerer ingenting (void).
- BinaryOparator<T>-grensesnittet har metoden apply, som tar inn to objekt av typen T, og og returnerer ett objekt av samme type. For eksempel addisjon: Tar inn to doubles, returnerer summen (én double).

forEach method of collections

Vi ser på ArrayList collection

```
ArrayList<String> names = new ArrayList<String>();

names.add("Hans");
names.add("Ola");

for (int i=0;i<names.size();i++) {
    out(names.get(i));
}
```

Eller:

```
for (String name: names) {
    out(name);
}
```

… forEach method of collections

Lambda:

```
names.forEach(s -> System.out.println(s));
```

Eller:

```
names.forEach(s -> out(s));
```

| DEMO: FL02.eks.lambda.forEach.string.Example01

| DEMO: FL02.eks.lambda.forEach.string.Example02

| DEMO: FL02.eks.lambda.forEach.object

Sortere med Comparator

Tradisjonell måte:

```
studenter.sort(new Comparator<Student>() {
     @Override
     public int compare(Student a, Student b) {
          return a.getNavn().compareTo(b.getNavn());
     }
);
studenter.forEach(s->out(s));
```

Lambda:

```
studenter.sort((a,b) -> a.getNavn().compareTo(b.getNavn()));
studenter.forEach(s->out(s));
```

| DEMO: FL02.eks.lambda.sortering.eksempel1

Streams

- Streams gjør det veldig enkelt for oss å utføre operasjoner på lister på en kort og elegant måte, i kombinasjon med lambda

- I stedet for å forklare hva streams er, skal vi heller vise eksempler på bruken av det

Streams - anyMatch

Tradisjonell måte:

```
boolean womanExists = false;
for (Student student : studenter) {
    if (student.getKjoenn() == "Kvinne") {
        womanExists = true;
        break;
    }
}
out(womanExists);
```

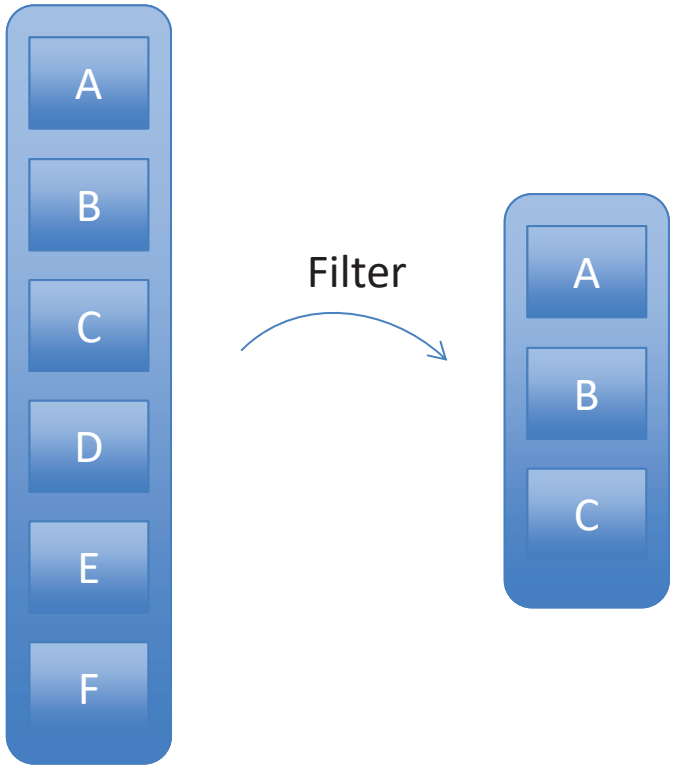
Lambda:

```
out(studenter.stream().anyMatch(p -> p.getKjoenn() == "Kvinne"));
```

| DEMO: FL02.eks.lambda.streams.anyMatch

Streams - filter

- Filter er en svært vanlig operasjon på lister som mange programmeringsspråk har støtte for.
- Filter kalles på en liste, og returnerer en ny liste med kun de elementene som tilfredsstiller et gitt betingelse



Streams - filter

Tradisjonell måte:

```
ArrayList<Student> kvinner = new ArrayList<Student>();
for (Student student : studenter) {
    if (student.getKjoenn() =="Kvinne") {
    kvinner.add(student);
    }
}
out(kvinner);
```

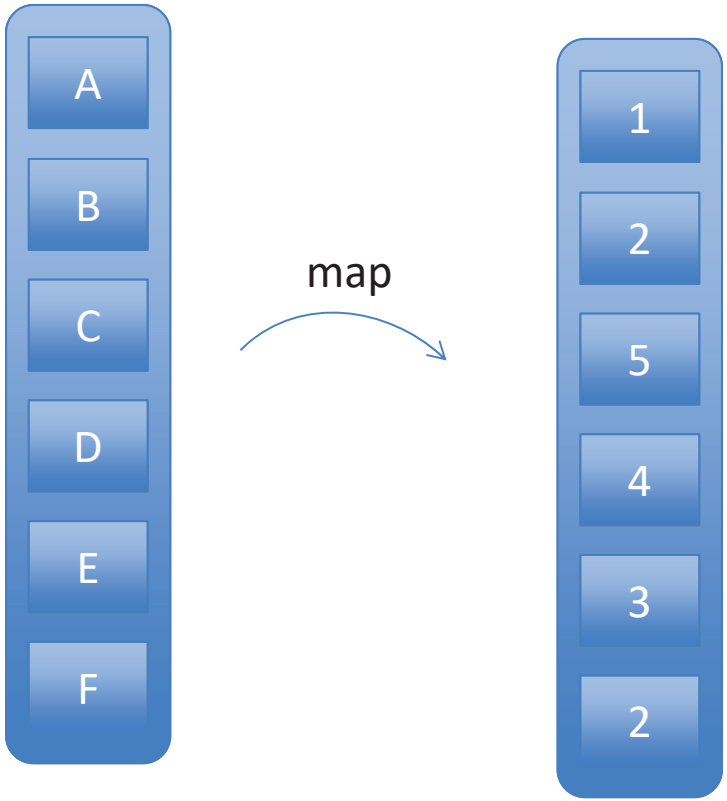
Lambda:

```
out(studenter.stream().
filter(p -> p.getKjoenn() =="Kvinne").collect(Collectors.toList()));
```

| DEMO: FL02.eks.lambda.streams.filter

Streams - map

Map brukes for å danne en ny liste av en annen liste, der en gitt funksjon blir kalt på alle elementene i lista



Streams – map (example)

```
ArrayList<Student> studenter = new ArrayList<Student>();
studenter.add(new Student("Ole Petter", "Hansen", 19801212,"Mann"));
studenter.add(new Student("Ingrid", "Olsen", 197512101,"Kvinne"));
studenter.add(new Student("Åse Marie", "Jensen", 19730506,"Kvinne"));
```

Tradisjonell måte:

```
out("\nTradisjonell måte: ");
ArrayList<Integer> fDatoer = new ArrayList<Integer>();
for (Student student : studenter) {
     fDatoer.add(student.getFdato());
}
```

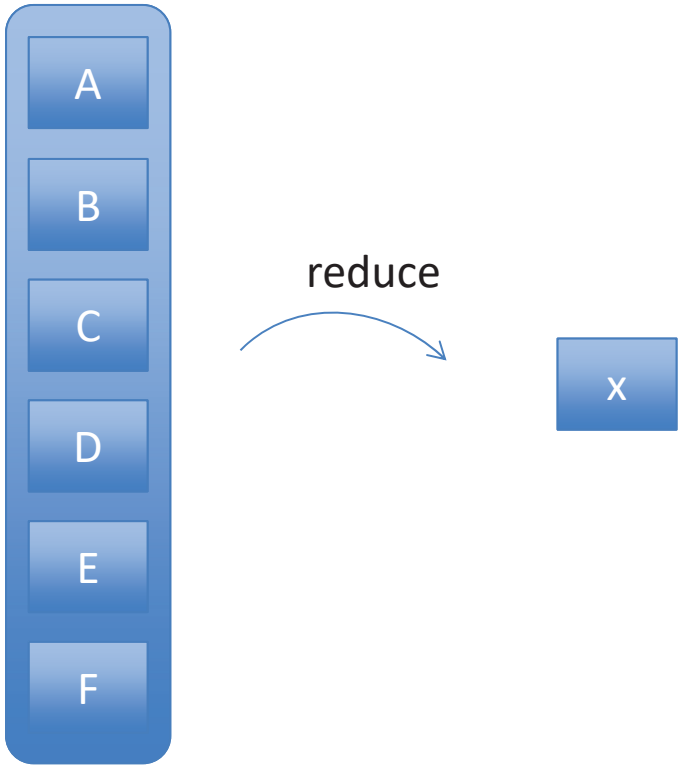
Lambda:

```
fDatoer = studenter.stream().map(Student::getFdato).collect(Collectors.toList());
```

| DEMO: FL02.eks.lambda.streams.map

Streams - reduce

Reduce brukes for å redusere en liste til ett enkelt resultat



Streams – map (eks: finne maks)

```
ArrayList<Student> studenter = new ArrayList<Student>();
studenter.add(new Student("Ole Petter", "Hansen", "1980/12/12","Mann"));
studenter.add(new Student("Ingrid", "Olsen", "1975/12/10","Kvinne"));
studenter.add(new Student("Åse Marie", "Jensen", "1973/05/06","Kvinne"));
```

Tradisjonell måte:

```
int maxAlder=0;
for (Student student : studenter) {
     int alder = student.getAlder();
     if(alder>maxAlder) {
          maxAlder = alder;
     }
}
```

Lambda:

```
maxAlder = studenter.stream().map(Student::getAlder).reduce(Math::max).get();
```

| DEMO: FL02.eks.lambda.streams.reduce. ReduceExampleMax

Streams – map (eks: finne sum)

```
ArrayList<Student> studenter = new ArrayList<Student>();
studenter.add(new Student("Ole Petter", "Hansen", "1980/12/12","Mann"));
studenter.add(new Student("Ingrid", "Olsen", "1975/12/10","Kvinne"));
studenter.add(new Student("Åse Marie", "Jensen", "1973/05/06","Kvinne"));
```

Tradisjonell måte:

```
int sumAlder=0;
for (Student student : studenter) {
     int alder = student.getAlder();
     sumAlder += alder;
}
```

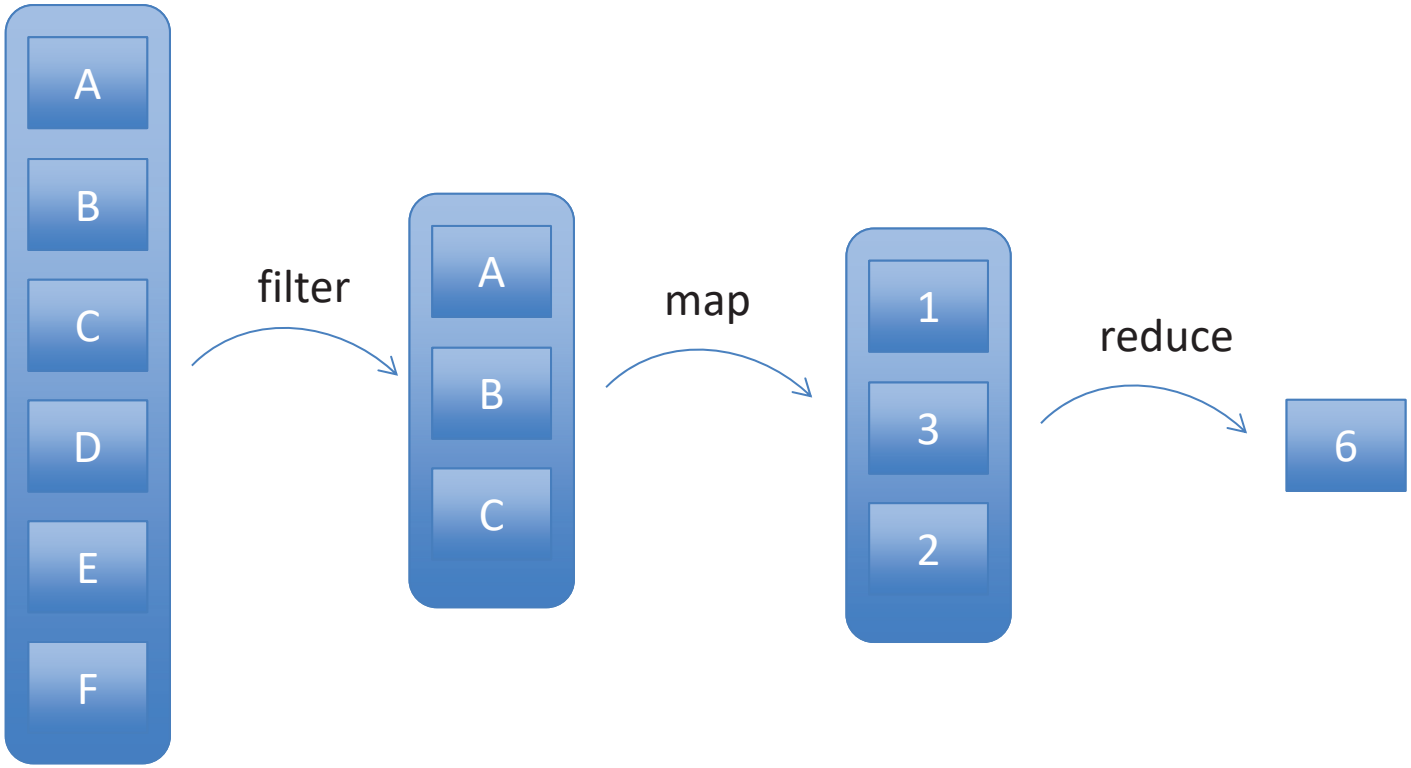
Lambda:

```
maxAlder =
studenter.stream().map(Student::getAlder).reduce((a,b) -> a+b).get();
```

| DEMO: FL02.eks.lambda.streams.reduce. ReduceExampleSum

Pipelines

Flere stream funksjoner kan kombineres som en pipeline for å utføre flere operesjoner under ett.



Pipeline – example 1

```
ArrayList<Student> studenter = new ArrayList<Student>();
studenter.add(new Student("Ole Petter", "Hansen", "1980/12/12","Mann"));
studenter.add(new Student("Ingrid", "Olsen", "1975/12/10","Kvinne"));
studenter.add(new Student("Åse Marie", "Jensen", "1973/05/06","Kvinne"));
```

Pipeline:

```
studenter.stream().filter(p -> p.getKjoenn() == "Kvinne")
.map(Student::getAlder)
.sorted()
.forEach(s->out(s));
```


| DEMO: FL02.eks.lambda.streams.pipeline. PipelineExample1

Pipeline – example 2

Pipeline:

```
List<String> myList = Arrays.asList("a1", "a2", "b1", "c2", "c1", "c0");

myList.stream().filter(s -> s.startsWith("c"))
.map(String::toUpperCase)
.sorted()
.forEach(s->out(s));
```


| DEMO: FL02.eks.lambda.streams.pipeline. PipelineExample2