Innkapsling

INF101 forelesning 8. Februar 2022

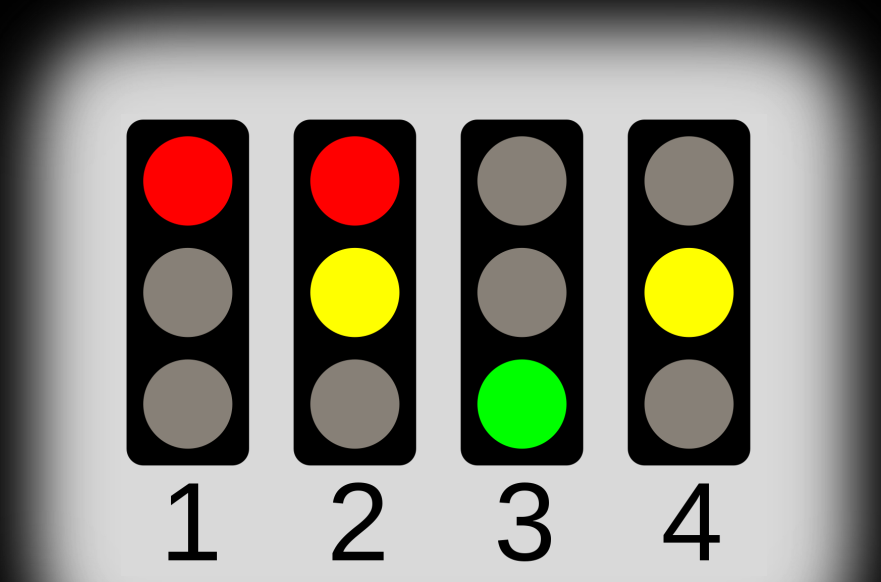
Torstein Strømme

Også innom: enum, exceptions, grensesnitt, Runnable

Eksempel: Trafikklys

- ITrafficLight representerer hvordan lyset brukes

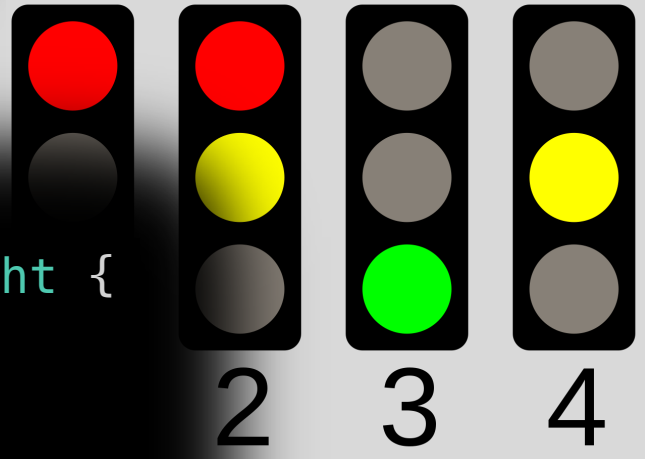
```
public interface ITrafficLight {
    void goToGreen();
    void goToRed();

    boolean getGreen();
    boolean getYellow();
    boolean getRed();

}
```



Eksempel: Trafikklys

- TrafficLight representerer tilstand som en *enum* -type

```
public class TrafficLight implements ITrafficLight {

    static enum State {
        BLANK, STOP, GETREADY, GO, HURRY
    }

    State state;

    TrafficLight() {
        this.state = State.BLANK;
    }
}
```

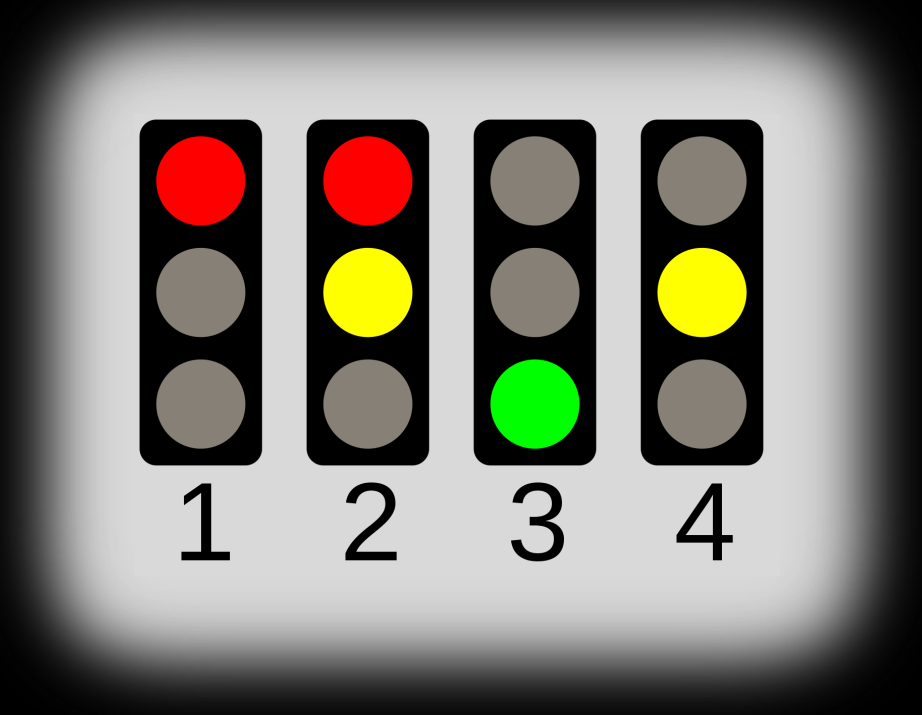


Hendig for å unngå rotete tilstander

Eksempel: Trafikklys

- TrafficLight implementerer metodene fra interface

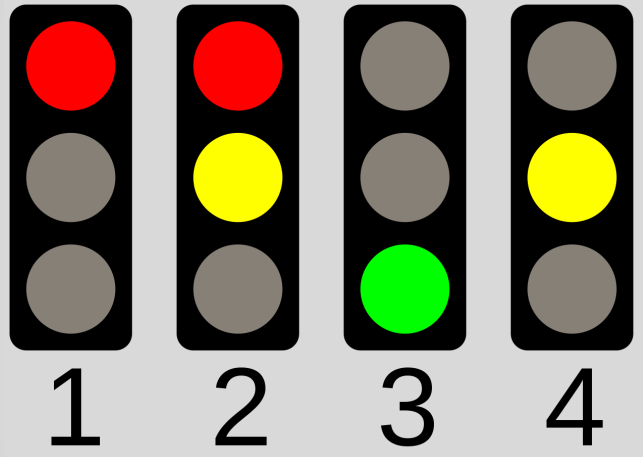
```
@Override
public boolean getYellow() {
    return this.state == State.GETREADY ||
           this.state == State.HURRY;
}
```



Eksempel: Trafikklys

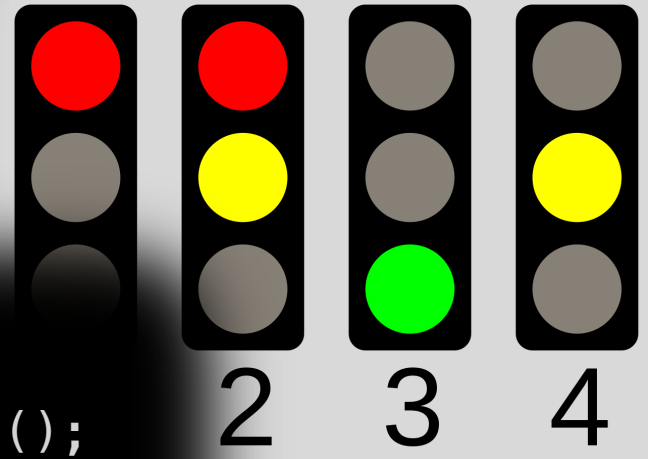
- TrafficLight passer på at lysene kommer i riktig rekkefølge

```
void goToNextState() {
    if (this.state == State.BLANK) {
        this.state = State.BLANK;
    }
    else if (this.state == State.STOP) {
        this.state = State.GETREADY;
    }
    else if (this.state == State.GETREADY) {
        this.state = State.GO;
    }
    ...
}
```



Eksempel: Trafikklys

- Hva hvis brukeren av TrafficLight – klassen ikke skjønner hvordan den skal brukes?




```
public static void main(String[] args) {
    TrafficLight trafficLight = new TrafficLight();

    // Setting light to green
    trafficLight.state = TrafficLight.State.GO;

    // Changed my mind
    trafficLight.state = TrafficLight.State.STOP;
}
```

Tilgangskontroll

- Hvilke klasser har tilgang til variabler og metoder?
- Ved å spesifisere enten
  - private
  - public
  - protected
  - (default)

  vil metoder og variabler være tilgjengelig fra andre klasser i ulik grad

Tilgangsmodifikatorer (access modifiers)

Hvilke klasser har tilgang til variabelen/metoden?

	public	protected	default	private
Samme klasse	Ja	Ja	Ja	Ja
Samme pakke	Ja	Ja	Ja	Nei
Klasse som arver	Ja	Ja	Nei	Nei
Alle	Ja	Nei	Nei	Nei

Hvorfor skjule?

- “Low coupling”
- Modularitet
- Testbarhet
- Skiller ansvarsområder – “single-responsibility principle”
- Hindrer andre i å klusse til dine strukturer uten å endre på dine filer

Hvordan skjule?

- Grensesnitt
  - Hovedregel: alltid benytt et grensesnitt som som type for variabler
  - *Benytt det mest restriktive grensesnittet som lar deg utføre oppgaven du har ansvar for*
- Tilgangsmodifikatorer
  - Benytt private som utgangspunkt for både variabler og metoder
  - Utvid tilgang gradvis ved behov; men vær sikker på at det er nødvendig.
  - *Eksponér heller en metode enn en variabel* (bruk *getters*)

en *getter* er en (public) metode som returnerer en (private) instans-variabel

Restriktive grensesnitt

- Å bruke grensesnitt er kanskje den beste måten å oppnå innkapsling
- Restriktive grensesnitt fører til
  - Høy grad av abstraksjon
  - Høy modularitet
  - Høy grad av innkapsling
  - Høy grad av gjenbruk

- Noen grensesnitt er spesielt viktige
  - Iterable, Comparable
  - Runnable, Consumer, Function, Supplier

Runnable

```
package java.lang;

public interface Runnable {

    public abstract void run();

}
```

Runnable

```
interface Runnable {
    void run();
}
```

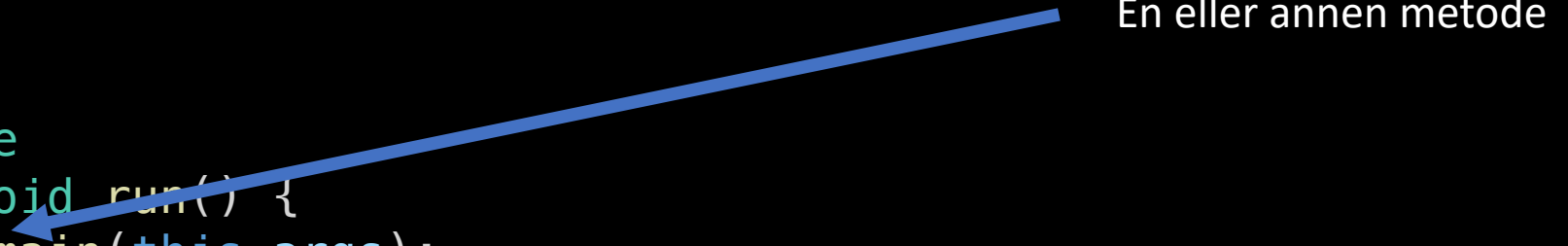
- Brukes for å kjøre kode parallelt i flere “tråder”
- Kan brukes når du ønsker å sende en metode som argument

```
public class TimeIt {

    public static long timeIt(Runnable method) {
        Date timeBefore = new Date();
        method.run();
        Date timeAfter = new Date();
        return timeAfter.getTime() - timeBefore.getTime();
    }

}
```

Runnable

```
interface Runnable {
    void run();
}
```

```
class RunApp implements Runnable {
    String[] args;

    RunApp(String[] args) {
        this.args = args;
    }

    @Override
    public void run() {
        App.main(this.args);
    }

}

public static void main(String[] args) {
    long ms = TimeIt.timeIt(new RunApp(args));
    System.out.println("Running App took " + ms + "ms");
}
```



Runnable (bonus-slide, ikke pensum)

```
interface Runnable {
    void run();
}
```

```
public static void main(String[] args) {
    long ms = TimeIt.timeIt(new Runnable() {

        @Override
        public void run() {
            App.main(args);
        }
    });
    System.out.println("Running App took " + ms + "ms");
}
```

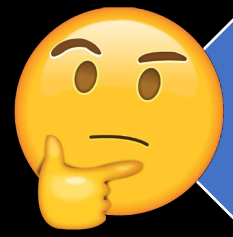
Hva er hva?

```java
package inf101v22.forelesning;

public class App implements Runnable {

    public static final String DEFAULT_TEXT = "Hello World";
    private String text;

    public static void main(String[] args) {
        App app = new App(DEFAULT_TEXT);
        app.run();
    }

    App(String text) {
        this.text = text;
    }

    @Override
    public void run() {
        System.out.println(this.text);
    }
}
```

pakke
klasse
objekt
grensesnitt/interface
instans-variabler
lokale variabler
globale variabler
static metoder
instans -metoder
tilgangsmodifikatorer
konstruktør
main –metode
parameter
argument
array
retur-type
signatur

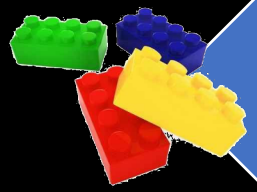
private
public
Runnable
stack
heap
refererte typer
primitive

Objektorientert programmering



Abstraksjon

- Hvilke egenskaper er relevante?

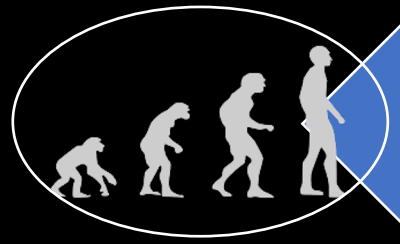


Modularitet

- Lag kode som byggeklosser

Innkapsling

- Skjul detaljer; ha kontroll over egne strukturer.



Gjenbruk

- DRY – don’t repeat yourself
- Biblioteker, polymorfisme og generiske typer

Ulike typer – samme kode

INF101 forelesning 8. Februar 2022

Torstein Strømme

Stikkord:

DRY - Don’t repeat yourself

Ulike typer – samme kode: MatteQuiz

```java
public static void main(String[] args) {
    MatteQuiz quiz = new MatteQuiz();
    quiz.generateQuestion(5, new Addition());
    quiz.generateQuestion(5, new Subtraction());
}
```

```java
interface Operator {
    char getOperationSymbol();
    int doOperation(int num1, int num2);
}
```

```java
class Addition implements Operator {
    public char getOperationSymbol() { return '+'; }
    public int doOperation(int num1, int num2) { return num1 + num2; }
}

class Subtraction implements Operator {
    public char getOperationSymbol() { return '-'; }
    public int doOperation(int num1, int num2) { return num1 - num2; }
}
```

Ulike typer – samme kode: MatteQuiz

```
private void generateQuestion(int n, Operator operator) {
    for (int i = 0; i < n; i++) {
        int num1 = random.nextInt(100);
        int num2 = random.nextInt(100);
        int ans = operator.doOperation(num1, num2);
        char op = operator.getOperationSymbol();

        String q = num1 + " " + op + " " + num2 + " = ";
        String a = "" + ans;
        Question question = new Question(q, a);
        questions.add(question);
    }
}
```

Polymorfisme

- Når det samme metodekallet gjør ulike ting
  Eksempel:

  ```
  operator.doOperation(num1, num2)
  ```

  gir ulik oppførsel avhengig av om operator er Addition eller Subtraction

- Polymorfisme gjør det mulig at koden “utenfor” (som utfører kallet) kan gjenbrukes til ulike formål
  Eksempel:

  ```
  generateQuestion
  ```

  brukes både når addisjon og subtraksjon –spørsmål utvikles.

Hvordan oppnå polymorfisme?

- Grensesnitt

```
class Addition implements Operator { ... }
class Subtraction implements Operator { ... }
```

- Arv

  (mer om det senere i kurset)

- (Statisk polymorfisme/overloading: ulike signaturer, samme navn)

```
static double square(double x) { return x * x; }
static int square(int x) { return x * x; }
```

Hva om vi ønsker å returnere samme type som vi fikk inn?

Finn det største tallet

```
def find_biggest(vals):
    largest_so_far = vals[0]
    for val in vals:
        if val > largest_so_far:
            largest_so_far = val
    return largest_so_far


print(find_biggest([3, 6, 4]))         # Printer 6
print(find_biggest([0.3, 0.6, 0.4]))   # Printer 0.6
print(find_biggest(["a", "d", "c"]))   # Printer d
```

Finn det største tallet

```java
public static Integer findLargest(List<Integer> vals) {
    Integer largestSoFar = vals.get(0);
    for (Integer val : vals) {
        if (num > largestSoFar) {
            largestSoFar = val;
        }
    }
    return largestSoFar;
}

public static void main(String[] args) {
    System.out.println(findLargest(Arrays.asList(3, 6, 4)));
}
```

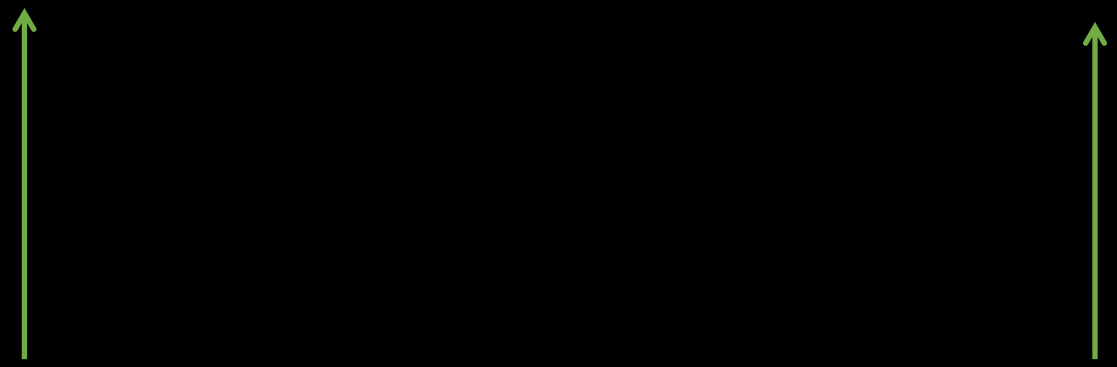
Finn det største tallet

```java
public static Double findLargest(List<Double> vals) {
    Double largestSoFar = vals.get(0);
    for (Double val : vals) {
        if (num > largestSoFar) {
            largestSoFar = val;
        }
    }
    return largestSoFar;
}

public static void main(String[] args) {
    System.out.println(findLargest(Arrays.asList(0.3, 0.6, 0.4)));
}
```

DRY - Don’t repeat yourself

Generiske typer

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



Generiske typer: eksempel

```
public static void main(String[] args) {
    Box<String> myStringBox = new Box<String>();
    myStringBox.put("Hello");
    String hello = myStringBox.pick();
    System.out.println(hello); // Prints 'hello’

    String helloAgain = myStringBox.pick();
    System.out.println(helloAgain); // Prints ‘null'
}
```

Generiske typer: eksempel

```
public static void main(String[] args) {
    Box<String> myStringBox = new Box<String>();
    myStringBox.put("Hello");
    String hello = myStringBox.pick();
    System.out.println(hello); //

    String helloAgain = myString
    System.out.println(helloAga
}
```

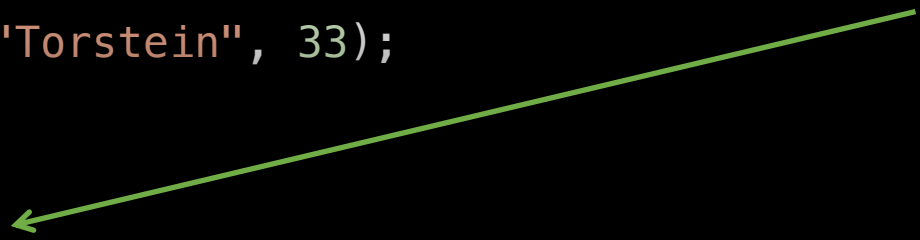
```
class Box <MyCustomType> {
    private MyCustomType element = null;

    public void put(MyCustomType element) {
        this.element = element;
    }

    public MyCustomType pick() {
        MyCustomType element = this.element;
        this.element = null;
        return element;
    }
}
```

Generiske typer: flere typer

```
public static void main(String[] args) {
    Pair<String, Integer> lecturer = new Pair<>("Torstein", 33);
    String theName = lecturer.getFirst();
    int theAge = lecturer.getSecond();
    ...
}
```

Generiske typer: flere typer

```
public static void main(String[] args) {
    Pair<String, Integer> lecturer = new Pair<>("Torstein", 33);
    String theName = lecturer.getFirst();
    int theAge = lecturer.getSecond();
    ...
}
```



```
public class Pair <A, B> {
    private final A first;
    private final B second;

    public Pair(A first, B second) {
        this.first = first;
        this.second = second;
    }

    public A getFirst() { return this.first; }
    public B getSecond() { return this.second; }
}
```

Generiske typer: oppsummering, del A

- Når vi ønsker å kunne bruke den samme koden for ulike typer
- Når det ikke er greit å miste informasjon om hvilken type vi har
- Plassholderne blir beskrevet mellom diamant-klammer: <>

```
public class Pair <A, B> { ... }
```

- Vi kan etterpå bruke plassholderne som vanlige typer
- For å bruke en generisk klasse, oppgi en reell type når variabelen defineres:

```
Pair<String, Integer> lecturer = new Pair<>("Torstein", 33);
```

Generiske typer: også for metoder

```
public static void main(String[] args) {
    Pair<String, Integer> lecturerA = new Pair("Torstein", 33);
    Pair<Integer, String> lecturerB = new Pair(33, "Torstein");
    System.out.println(areSwapped(lecturerA, lecturerB));
}
```

```
private static <K, V> boolean areSwapped(Pair<K, V> a, Pair<V, K> b) {
    K a1 = a.getFirst();
    V a2 = a.getSecond();
    V b1 = b.getFirst();
    K b2 = b.getSecond();
    return Objects.equals(a1, b2) && Objects.equals(a2, b1);
}
```

Generiske typer: men ikke hva som helst

```
interface Valuable {
    public double getValue();
}


class House implements Valuable {
    int value;

    House(int value) {
        this.value = value;
    }

    @Override
    public double getValue() {
        return (double) this.value;
    }
}
```

Generiske typer: men ikke hva som helst

```
public static void main(String[] args) {
    List<House> allHouses = Arrays.asList(new House(1), new House(6), new House(4));
    House mostValuedHouse = getMostValuable(allHouses);
}
```

```
public static <K extends Valuable> K getMostValuable(Iterable<K> objects) {
    K mostValuableSoFar = null;
    double currentBestValue = Double.MIN_VALUE;
    for (K obj : objects) {
        double value = obj.getValue();
        if (value < currentBestValue) {
            mostValuableSoFar = obj;
            currentBestValue = value;
        }
    }
    return mostValuableSoFar;
}
```

Generiske typer: også for grensesnitt

```
interface Comparable <T> {
/**
* Compares this object with another, and returns a numerical result based
* on the comparison. If the result is negative, this object sorts less
* than the other; if 0, the two are equal, and if positive, this object
* sorts greater than the other.
* @param other the object to be compared
* @return an integer describing the comparison
*/
int compareTo(T other);
}
```

Generiske typer: også for grensesnitt

```
class House implements Comparable<House> {
    int value;

    House(int value) {
        this.value = value;
    }

    @Override
    public int compareTo(House other) {
        if (this.value < other.value) return -1;
        if (this.value > other.value) return 1;
        return 0;
    }
}
```

Finn det største… huset!

```
public static void main(String[] args) {
    List<House> allHouses = Arrays.asList(new House(1), new House(6), new House(4));
    House mostValuedHouse = findLargest(allHouses);
}

public static <K extends Comparable<K>> K findLargest(Iterable<K> objects) {
    K largestSoFar = null;
    for (K obj : objects) {
        if (largestSoFar == null || largestSoFar.compareTo(obj) < 0) {
            largestSoFar = obj;
        }
    }
    return largestSoFar;
}
```

Generiske typer: oppsummering, del B

- Når vi ønsker å kunne bruke den samme koden for ulike typer
- Kan brukes i
  - Klasser
    ```
    class Box <MyCustomType> { ... }
    ```
  - Grensesnitt (interface)
    ```
    interface Comparable <T> { ... }
    ```
  - Individuelle metoder
    ```
    private <K, V> boolean areSwapped(Pair<K, V> a, Pair<V, K> b) { ... }
    ```
- Vi kan kreve at typen implementerer et interface
  ```
  public <K extends Valuable> K getMostValuable(Iterable<K> objects) { ... }
  ```
- Vi kan kreve at typen implementerer et generisk interface
  ```
  public <K extends Comparable<K>> K findLargest(Iterable<K> objects) { ... }
  ```

Generics i Java sitt standardbibliotek

- List og ArrayList
- Set og Map
- Collections.sort()

```
List<String> list = new ArrayList<String>();
```

Viktige grensesnitt: samlinger (collections)

Grensesnitt	**Eksempler:**
Iterable<E>	*Collection<E>*
Collection<E>	*Set<E>, List<E>*
Set<E>	HashSet<E>, *SortedSet<E>*
SortedSet<E>	TreeSet<E>
List<E>	ArrayList<E>, LinkedList<E>, Vector<E>
Map<K, V>	HashMap<K, V>, TreeMap<K, V>

Viktige grensesnitt: funksjonelle prinsipper

Grensesnitt	**Krever metode:**
Runnable	`public void run()`
Consumer<T>	`public void accept(T)`
BiConsumer<T, U>	`public void accept(T, U)`
Supplier<R>	`public R get()`
Function<T, R>	`public R apply(T)`
BiFunction<T, U, R>	`public R apply(T, U)`