Exercise 1: Øvelse i comparable-interfacet.
Implementer en klasse indeholdende en dato:

```
public class Dato implements Comparable<Dato> { ... }
```

datoerne skal kunne sammenlignes, så tidligere datoer er mindre end senere datoer.
Lav en konstruktør, der tager tallene år, måned og dag som parameter.
Lav også en toString() funktion, der skriver datoen på formen

```
"24/12 2018"
```

End of Exercise 1

Exercise 2: Øvelse i comparable-interfacet.
Lav en klasse Barn indeholdende barnets navn og fødselsdato

```
public class Barn implements Comparable<Barn> { ... }
```

børnene skal kunne sammenlignes, så yngre børn er mindre end ældre børn. Er børnene præcis lige gamle, sorteres de alfabetisk.
Lav en konstruktør, der tager navn, fødselsår, -måned og -dag som parameter.
Lav også en toString() funktion, der skriver barnet Brian født juleaften 2008 på formen

```
"Brian born 24/12 2008"
```

Hint: løs opgaven "datoer kan sammenlignes" først.
End of Exercise 2

Exercise 3: Øvelse i at anvende *datastrukturer*, specifikt ArrayList.
Complete the a class `ArrayListX` on CodeJudge. It handles ArrayLists of integers and which has the following two static methods.

```
public static void append(ArrayList<Integer> list1,
        ArrayList<Integer> list2)
```

which takes two array lists of integers as input and appends the elemets of list2 to list1 in unchanged order. List list2 remains unchanged. The second method is

```
public static ArrayList<Integer> merge(ArrayList<Integer> list1,
ArrayList<Integer> list2)
```

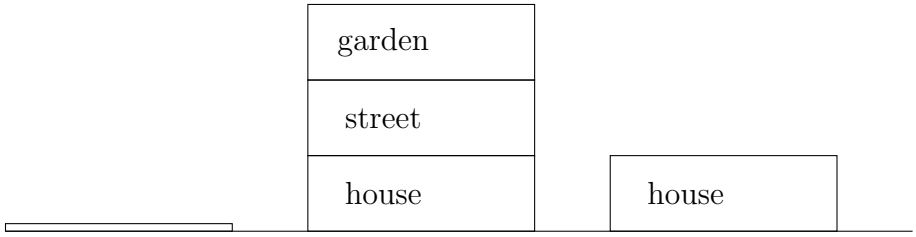
which takes two array lists of integers as input and merges them into one, which is returned. The merging is done as follows. The first element from list1 is the first in the returned list, then follows the first from list2, then the second from list1, then follows the second from list2 etc. If the list have different lengths, the the "tail" of the longer one is added at the end. Test your implementation with list1 consisting
End of Exercise 3

Exercise 4: Øvelse i at bygge *datastrukturer*.
Implement a **generic** class `Stack` which implements the data structure of a stack. A *stack* is a last-in-first-out (LIFO-) storage for objects of a certain type. The type of objects stored, is specified when the constructor is called, e.g., as
`Stack<Point> st = new Stack<Point>()`. A stack supports the following operations

- `public void push(<type> e)` This operation puts the element `e` on top of the stack.
- `public <type> pop()` This operation returns the top element of the stack and removes it from the stack.

- `empty` This returns true only when the stack does not contain any elements.

Example: Development of a stack for strings.



Empty stack

After:
`push("house")`
`push("street)`
`push("garden")`

After:
`pop()`
`pop()`

First implement the stack for type `String` and then make a generic implementation where the user can choose the type of objects stored in the stack. Make sure your implementation can handle boundary cases, e.g., a pop on an empty stack.

Hint: You might first want to implement the stack for a fixed type (e.g., String) and only then add the generics.

End of Exercise 4

Exercise 5: Øvelse i at bygge *datastrukturer* og lave *generiske klasser*
Lav en datastruktur for et voksende array i klassen `VoksendeArray<T>`

```
public class VoksendeArray<T> {
        private Object[] objekter;
        private int peger;
        ...
}
```

Klassen skal indeholde konstruktører for at initiere med og uden indhold:

```
public VoksendeArray(T[] ds) {
        ...
}

public VoksendeArray() {
        ...
}
```

Desuden skal den indeholde funktionerne

```
public boolean add(T e) { // tilføjer elementet e og returnerer true
}
public boolean contains(T d) { // svarer om elementet d findes
}
public T get(int index) { // returnerer elementet på plads nr index
}
public int indexOf(T d) { // returnerer index for første forekomst af d
}
```

```
public int lastIndexOf(T d) { // ditto sidste
}
public int size() { // returnerer antallet af elementer i datastrukturen
}
public T[] toArray() {
        // returnerer et nyt "trimmet" array indeholdende elementerne
}
```

Hint: løs evt først øvelsen for typen String, og løs den bagefter for T.

End of Exercise 5

Exercise 6: Øvelse i at bruge datastrukturer.
Løs opgave 6 fra uge 7, denne gang ved brug af enten ArrayList eller VoksendeArray. Bemærk at nu behøves tallet `maxNoOfWords` ikke længere.

"Write a class `TextAnalysis` which reads a text file and allows some text mining. The class has to have a constructor

```
public TextAnalysis(String sourceFileName, int maxNoOfWords)
```

The argument `sourceFileName` is the name of the source file, if necessary with the path. The parameter `maxNoOfWords` is an upper bound on the number of words in the file which might help in your implementation.
Implement the following instance methods: (A *word* is a non-empty string consisting of only of letters (a,...,z,A,...,Z), surrounded by blanks, punctuation, hyphenation, line start, or line end.)

```
public int wordCount() // returns the number of words in the file
public boolean contains(String word) // returns whether word is contained
public String mostFrequent() // returns the most frequently occuring word.
```

"

Overvej forskellige løsninger til at gemme ordene og deres frekvens på en smart måde, så programmet kan udviddes til at supportere metoden

```
public int frequenceOf(String word)
```

uden at skulle løbe hele ordlisten igennem og tælle, hver gang metoden bliver kaldt.

End of Exercise 6