CHAPTER 5: INTERFACES

LEARNING OBJECTIVES

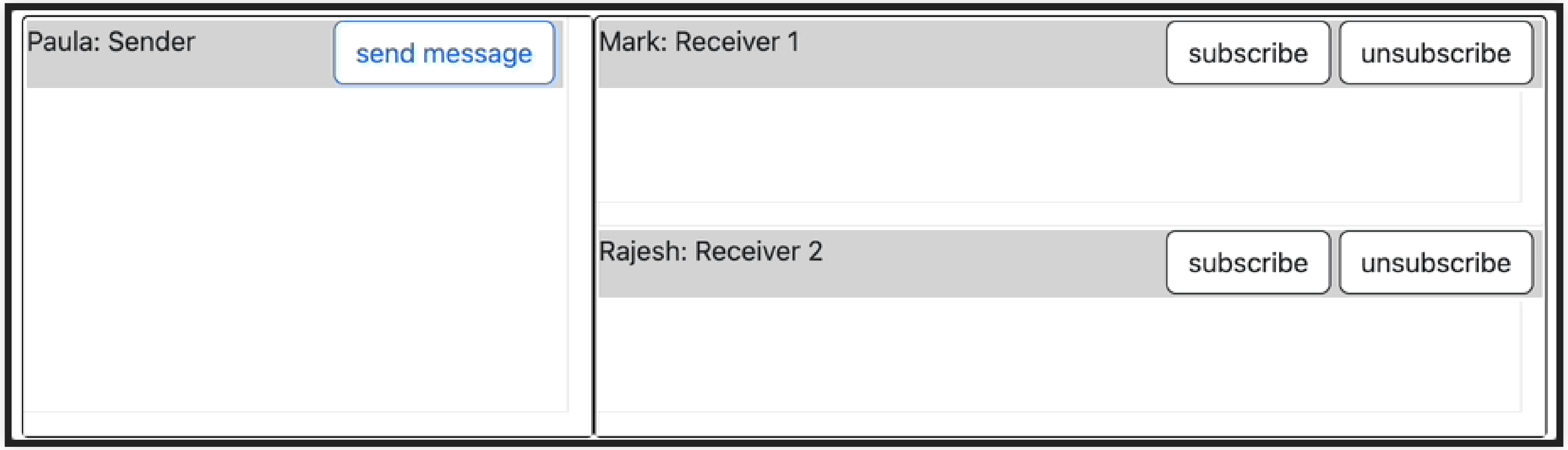
Being able to ...

- explain what a Java interface is and what it may contain
- use interface syntax to define and use interfaces
- use the Comparable interface
- use the Comparator interface
- use the UML representation of interfaces in class models

5.1 INTRODUCTION

INTERFACE CASE STUDY

There are many situations where an update on a data value leads to different reactions in other parts of a program. Look at the following example where the two receivers Mark and Rajesh react to a new message in different ways:



Mark just prints out the new message whereas Rajesh counts the messages and their length in bytes.

Of course, our previous example can be implemented with one Sender class and two receiver classes, Receiver1 and Receiver2. All classes know each other and we end up with a pretty "hard wired" solution!

However, in situations where the number of receivers may vary, we need a more general approach.

The question arises: how can we implement various receivers that can just subscribe for data changes without the changing data object knowing the receivers?

The advantage of this approach is clear: whenever a new receiver comes into play, there is no need to change the data object!

Solution: Contract between Sender and each Receiver

The connecting element of the contract is a method called `dataChanged` which is provided by each Receiver object and will be called by the Sender when a data change occurs.

- The Sender class offers a method `attach()` to register a receiver object
- Each receiver class fullfills the contract, i.e. implements method `dataChanged`
- When the Sender object's data are changed, it calls all attached receivers `r`: `r.dataChanged(newData)`

Java offers the **interface concept** to realize contracts which help to decouple communicating objects. In our example the sender does not know who the receivers are!

We define an interface named **IReceiver** which must be implemented by every receiver object which is interested in data changes.

```
interface IReceiver {
    void dataChanged(String newData);
}
```

- Interface `IReceiver` defines a method called `dataChanged`
- Classes may implement `IReceiver` by providing an implementation of `dataChanged`

Example Receiver implementation: ChangeLogger

A ChangeLogger object simply logs changes to the console.

```
class ChangeLogger implements IReceiver {
    public ChangeLogger(Sender s) {
        s.attach(this);
    }

    public void dataChanged(String newData) {
        System.out.println("[ChangeLogger] New data: " + newData);
    }
}
```

- ChangeLogger implements the interface - the compiler checks, whether `dataChanged` is implemented
- a receiver object knows the sender (constructor parameter) which is used to register the receiver object with the sender.
- when the data in the sender change, the dataChanged method is called, see class `Sender`

Example Receiver implementation: MessageCounter

A MessageCounter object counts messages and message sizes and prints them to the console.

```
class MessageCounter implements IReceiver {
    private int noMsgs = 0;
    private int totalSize = 0;

    public MessageCounter(Sender s) {
        s.attach(this);
    }

    public void dataChanged(String newData) {
      noMsgs++;
      totalSize += newData.length();
      System.out.printf("[MessageCounter] #msgs=%d, total size=%d bytes%n",
        noMsgs, totalSize);
    }
}
```

- just like ChangeLogger, MessageCounter registers with the sender object to receive changes in method `dataChanged`
- instead of printing new messages, simple statistics are printed

Sender implementation: (part 1)

```
class Sender {
    private String data = "";
    private IReceiver[] receivers = new IReceiver[10];

    public void attach(IReceiver r) {
        for (int i = 0; i < receivers.length; ++i) {
            if (receivers[i] == null) {
                receivers[i] = r;
                break;
            }
        }
    }
    ...
}
```

- data is the attribute that is subject to change, see method setData
- every attached receiver is stored in array receivers
  important: the actual receiver class is unknown!
- the first free position in the array is used to store the new receiver object

Sender implementation: (part 2)

```
class Sender {
    ...
    public void setData(String newData) {
        data = newData;
        for (int i = 0; i < receivers.length; ++i) {
            if (receivers[i] != null) {
                receivers[i].dataChanged(newData);
            }
        }
    }
}
```

- setData is called from outside the sender to store new data
- when new data arrive, all registered receivers are called using the "contract", i.e. the method dataChanged is called transporting the new data to the receiver

JAVA INTERFACE CONCEPT

Java, like other languages, offers the interface concept, which allows a strict separation of interface and implementation:

An interface defines a call interface for objects. If an object is known to offer a certain interface, other objects can use it, i.e. call its methods.

Objects can therefore call other objects of which they only know the interface (i.e. the methods), but not the class of the called object. This opens up freedom in the implementation of classes, as their implementation details (attributes or other methods outside the interface) do not need to be known to the caller.

In principle, an interface corresponds to an abstract class, i.e:

- Methods are without implementation (prototypes)
- Methods are always public
- No static properties
- No constructors
- Only constant attributes (corresponds to static public final declaration in a class).

But: as of Java 8, interfaces can also provide default implementations for methods, so that these do not necessarily have to be implemented .

A interface may be used as a type just like a class.

Object references of an interface type must reference an object whose class implements the interface.

Example:

```
class Printer {
    public static void printArea(Shape s) {
        System.out.println("Area: " + s.area());
    }
}

Shape s = new Circle(3.0);
Printer.printArea(s);
```

The above code only compiles without errors if class Circle implements interface Shape.

- This hold for assignments
- as well as for method parameters

Interfaces are the answer to many questions which arise during the development of larger programs:

- Which operations should an application provide to other applications?
- How can several development teams develop in parallel?
- How can frameworks (e.g. JavaFX) be elegantly integrated into your own classes?
- How can multiple inheritance be implemented in Java (single inheritance) ? Examples:
    - "An amphibious vehicle is a land vehicle and a water vehicle"
    - "An omnivore is a carnivore and a herbivore"
- How can software be designed to be easily changeable, i.e. flexible in the event of changes?

Simple interface example: every shape can calculate its area

```java
interface Shape {
    double area();
}
```

Let's use Shape to define a class Circle:

```java
interface Shape {
    double area();
}
```

Class Circle may be defined as follows:

```java
class Circle implements Shape {
    public double area() {
        return Math.PI * radius * radius;
    }
    public double addArea(Shape s) {
        return area() + s.area();
    }
}
```

Class Circle claims to implement interface Shape. The compiler checks whether all Shape methods are actually implemented!

The method `addArea(Shape s)` returns the sum of the areas of the circle and any object that also has an area. By using the Shape interface as a parameter, the method can be used flexibly! Let's extend the example to include a rectangle and then apply it:

```
class Rectangle implements Shape { ... }
```

```
Circle circle1 = new Circle(2.9)
Circle circle2 = new Circle(2.3);
Rectangle rect = new Rectangle(2.9, 3.0);
System.out.println("Circle1 + Rectangle: " + circle1.addArea(rect));
System.out.println("Circle1 + Circle2: " + circle1.addArea(circle2));
```

With `circle1.addArea()` the area of any object that implements the Shape interface can be added. So no specific class is required for the parameter, which results in great flexibility.

The example shows that interfaces are used as data types in the same way as classes.

```
Shape s1 = new Circle(2.0)
Shape s2 = new Rectangle(1.0, 1.5);
```

The declarations are correct if circle and rectangle both implement the Shape interface. So variable s1 is a reference to an object that implements the Shape interface.

Please be aware: With the above declarations, the call `s1.area()` *is correct, but the call* `s1.addArea(s2)` *is not because s1 is not a* `Circle` *reference! s1 is of type* `Shape`*, regardless of what it refers to.*

EXERCISE: INTRODUCTION TO INTERFACES

Which statements regarding interfaces are correct?

- ☐ Interfaces need a constructor
- ☐ Methods are without implementation (usually)
- ☐ Classes can inherit and implement an interface
- ☐ Classes can either inherit or implement an interface
- ☐ Interface methods need the public keyword
- ☐ Interfaces can have attributes

5.2 THE INTERFACE COMPARABLE

We already know how to check the equality of two objects:
the `equals()` method.

But how can we check whether one object is smaller or larger than another?

Objects often need to be compared with each other, for example to display a list in a sorted way. The Java standard library provides the Comparable interface, which provides a method for comparing objects:

```
interface Comparable {
  public int compareTo(Object o);
}
```

Class `String` implements the Comparable interface. The following example shows how to use it:

```java
String s1 = "Hello", s2 = "Shah Rukh";
int result = s1.compareTo(s2);
```

Calling `s1.compareTo(s2)` should yield the following results:

- $< 0 \rightarrow$ s1 is smaller than s2
- $= 0 \rightarrow$ s1 equals s2
- $> 0 \rightarrow$ s1 is greater than s2

Based on the above definition, objects can be compared and, for example, sorted, as sorting methods are based on the pairwise comparison of objects. The `Arrays` class provides very efficient sorting methods.

Here's an example. Output will be -7, 3, 9.

```
int[] list = { 3, 9, -7 };
java.util.Arrays.sort(list);

for (int value : list) {
    System.out.println(wert);
}
```

COMPARING SQUARES

The following example compares squares based on their area. The square class implements Comparable, so squares can be compared using the compareTo method.

```java
public class Square implements Comparable, Shape {
    public double area() {
        return sideLength * sideLength;
    }

    public int compareTo(Object o) {
        Square s = (Square) o; // only squares can be compared!
        double area1 = area(), area2 = s.area();

        if (area1 < area2)
            return -1; // called square < passed sq.
        else if (area1 > area2)
            return 1; // called square > passed sq.
        else
            return 0; // called square equals passed sq.
        }
}
```

Use of the example in a main method:

```
public static void main(String args[]) {
  Square s1 = new Square(2.01), s2 = new Square(2.0);
  int comparison = s1.compareTo(s2);
  if (comparison < 0)
    System.out.println("s1 < s2!");
  else if (comparison > 0)
    System.out.println("s1 > s2!");
  else
    System.out.println("s1 equals s2!");
}
```

REMARKS

When comparing float or double values, the comparison ("==") should be avoided as this can cause accuracy problems. In the example on the last page, smaller and larger conditions are therefore checked first.

A comparison with a defined threshold THRESHOLD would be optimal.

```
final double THRESHOLD = .0001;
if (Math.abs(float1 - float2) < THRESHOLD)
    System.out.println("f1 and f2 are equal using threshold\n")
else
    System.out.println("f1 and f2 are not equal using threshold\n")
```

Comparing integer values

When comparing two integers (types "int", "long", "short", "char"), it is sufficient to subtract the two values to produce a correct compareTo result.

Standard comparison functions

Comparisons between many data types already exist in Java:

- Integer.valueOf(9).compareTo(10)
- Double.valueOf(1.12).compareTo(2.34)
- "Hello".compareTo("Hugo")

The square example throws a *typecast exception* if `compareTo` is not called with a Square reference (which is allowed since the parameter type is Object!).

In the Generics chapter, we discuss ways to parameterise interfaces with data types (classes or interfaces). Accordingly, you would need to pass a square parameter instead of an object parameter. In that case the compiler would check if `compareTo` is used correctly ("compile-time checking versus runtime error").

5.3 THE COMPARATOR INTERFACE

The `Comparable` interface can be used to compare one instance with another - the comparison criterion is predefined by the implementation of `compareTo`.

However, sometimes you need to choose between a number of different comparison criteria - how can we achieve this?

Example: sorting an array of squares

```
Square[] list = { new Square(1.), new Square(0.1), new Square(3.0) };
Arrays.sort(list);
for (square s : list)
  System.out.println("Side length: " + s.sideLength)
```

The solution is to use a separate Comparator object with a method for comparing two objects for each use case. The comparison method is specified in the Comparator interface:

```
interface Comparator {
    int compareTo(object o1, object o2);
}
```

For example, if you want to sort squares by their perimeter and side length, you could define two Comparator classes, which are then passed to a special sort method of the Arrays class to achieve the desired sorting. A sample implementation can be found on the following page.

EXAMPLE: COMPARING SQUARES BY THEIR PERIMETER

```java
class PerimeterComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        double perimeter1 = ((Square) o1).perimeter();
        double perimeter2 = ((Square) o2).perimeter();

        if (perimeter1 < perimeter2)
            return -1;
        else if (perimeter1 > perimeter2)
            return 1;
        else
            return 0;
    }
}
```

EXAMPLE: COMPARING SQUARES BY THEIR SIDE LENGTH

```
class SideLengthComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        double sideLength1 = ((Square) o1).getSeitenlaenge();
        double sideLength2 = ((Square) o2).getSeitenlaenge();

        if (sideLength1 < sideLength2)
            return -1;
        else if (sideLength1 > sideLength2)
            return 1;
        else
            return 0;
    }
}
```

EXAMPLE: USING THE COMPARATOR

The sort method sorts the array of squares first by side length and then by perimeter. This is achieved by passing an appropriate comparator object. The compare method of the corresponding comparator object is used during the sorting process.

```java
// Sort the list of squares first by side length, then by perimeter
Square[] squares = { new Square(1), new Square(0.1), new Square(3) };
Arrays.sort(squares, new SideLengthComparator())
Arrays.sort(squares, new PerimeterComparator())
```

5.4 FURTHER ASPECTS

IMPLEMENTATION OF MULTIPLE INTERFACES

A class can implement any number of interfaces, but can only inherit from one class:

```
class ColouredSquare extends Square implements Shape, Comparable {
  // ...
}
```

The ColouredSquare class must implement each method of the two interfaces. If it does not, it must be defined as an abstract class, which forces the concrete subclasses to define the missing implementations.

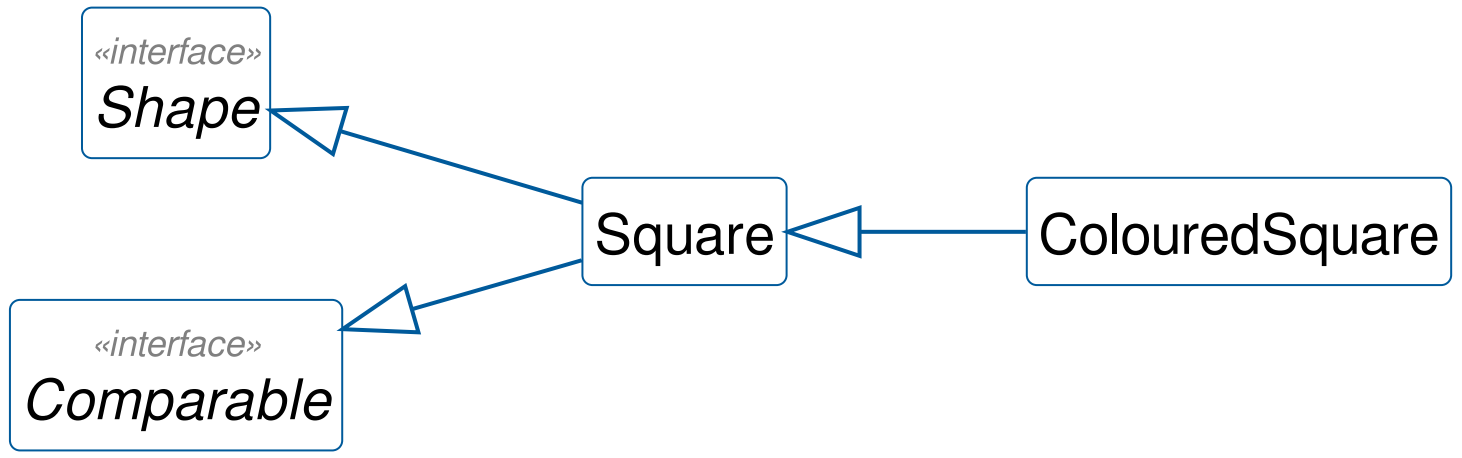
DERIVATION OF INTERFACES

An interface can "inherit" any number of interfaces using the `extends` keyword known from class inheritance. Interface inheritance means that the inheriting interface simply adds the methods (and other properties) of the base interface.

If the same method prototypes are defined in base interfaces, this is not a problem. The implementation of the required method simultaneously fulfils the requirements of all interfaces.

UML REPRESENTATION

In UML, an implementation relationship is represented by an inheritance arrow. If the arrow leads from a class to an interface, the arrow means `implements`. If it leads from an interface to another interface, the arrow means `extends`.



EXERCISE: UML CLASS MODEL AND INTERFACES

Given a ColouredSquare instance based on the previous UML class model.

```
ColouredSquare b = new ColouredSquare();
```

Which statements are correct?

- ☐ b instanceof Comparator
- ☐ b instanceof Square
- ☐ b instanceof Comparable
- ☐ b instanceof Object
- ☐ b instanceof Figure

5.5 EXERCISE: SENDER-RECEIVER CASE STUDY

EXERCISE: DETACHING THE CHANGELOGGER

Take a close look at the given Java implementation of the case study from the beginning of this section. In Messenger.main a sender and two receivers are created. Your task is to implement the stopLogging() method in class ChangeLogger which should unregister the logger from the sender. To accomplish this, call the corresponding method in class sender.

If everything works fine, your program output should be:

```
**data changed: '' -> 'Hello!'
New data: Hello!
#msgs=1, total size=6 bytes
**data changed: 'Hello!' -> 'What's up?'
#msgs=2, total size=16 bytes
```

EXERCISE: ADDING ANOTHER LOGGER

This last task is about adding another logger: a logger called "VerboseLogger". Use the empty file and define a Logger that acts quite similarly to the ChangeLogger. See the output below to identity the differences between the two loggers. Finally, activate line 14 to test your new logger.

If everything works fine, your program output should be:

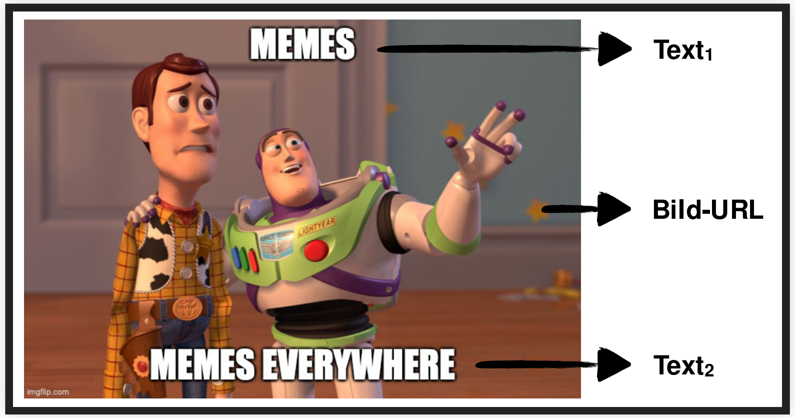
```
**data changed: '' -> 'Hello!
New data: Hello
#msgs=1, total size=6 byte
**data changed: 'Hello!' -> 'What's up?
New data: What's up
#msgs=2, total size=16 byte
**data changed: 'What's up?' -> 'Last message...
New data: Last message..
#msgs=3, total size=31 byte
Verbose: message was 'Last message...'
```

5.6 EXERCISE: MEMES

MEME INTERFACE

The following is an exercise on the interfaces Comparable and Comparator.

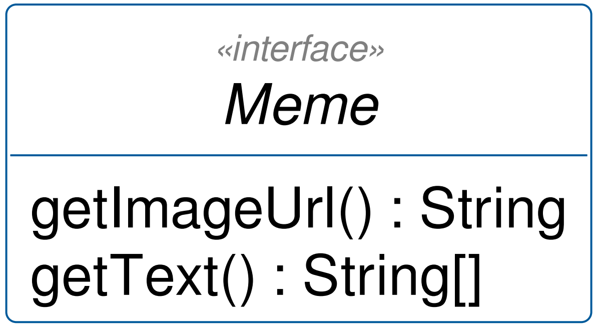
A meme is a graphic (*Image URL*) with a top (*Text$_1$*) and a bottom Text (*Text$_2$*). You can design your own memes at https://makeameme.org/!



Unzip meme-interface.zip and import the folder as "Existing Maven Project".

STEP 1: DEFINE INTERFACE

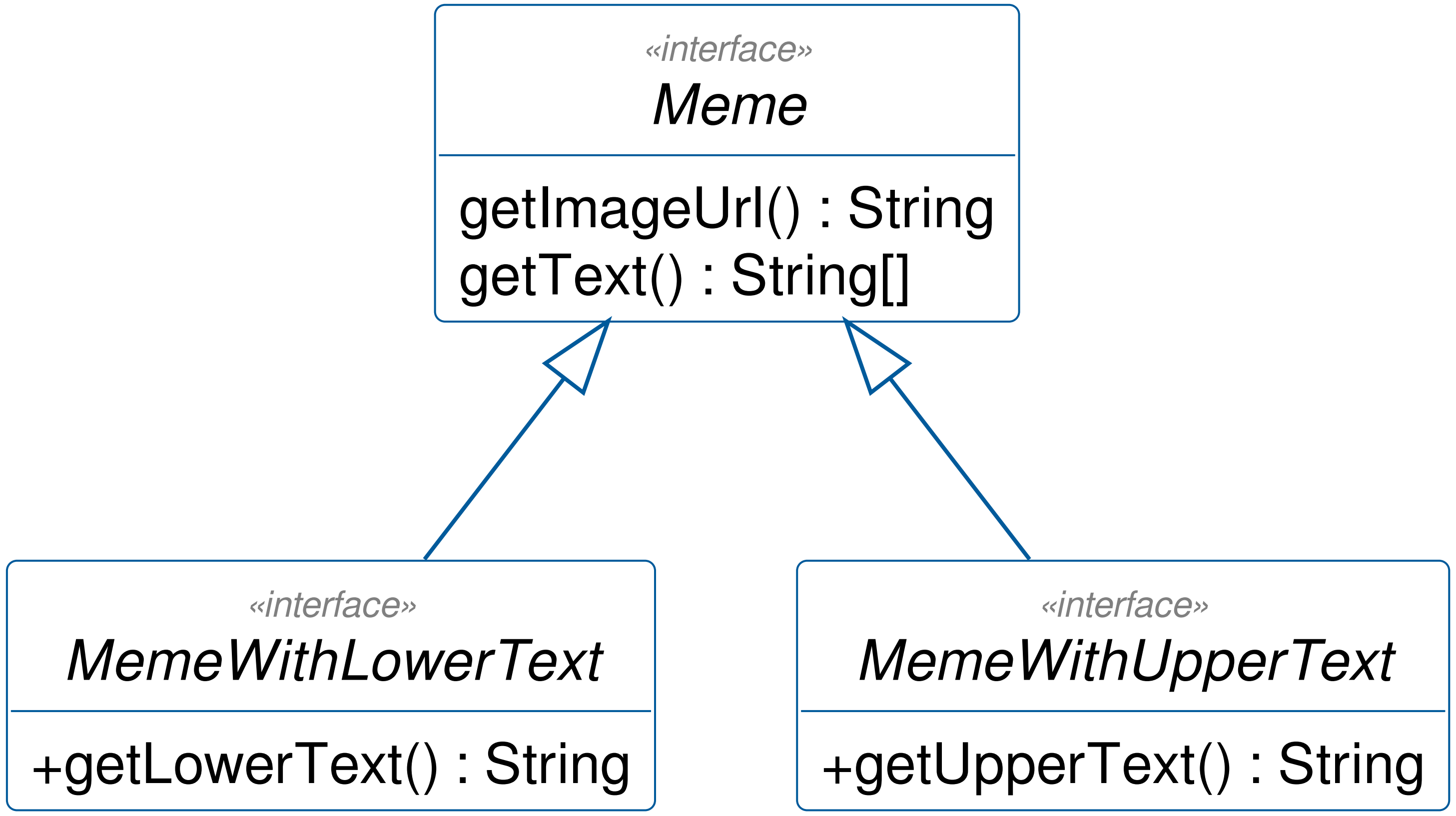
Define an interface named `Meme`.



- Add a method `getImageUrl()` which returns a string.
- Add a `getText()` method that returns a string[].

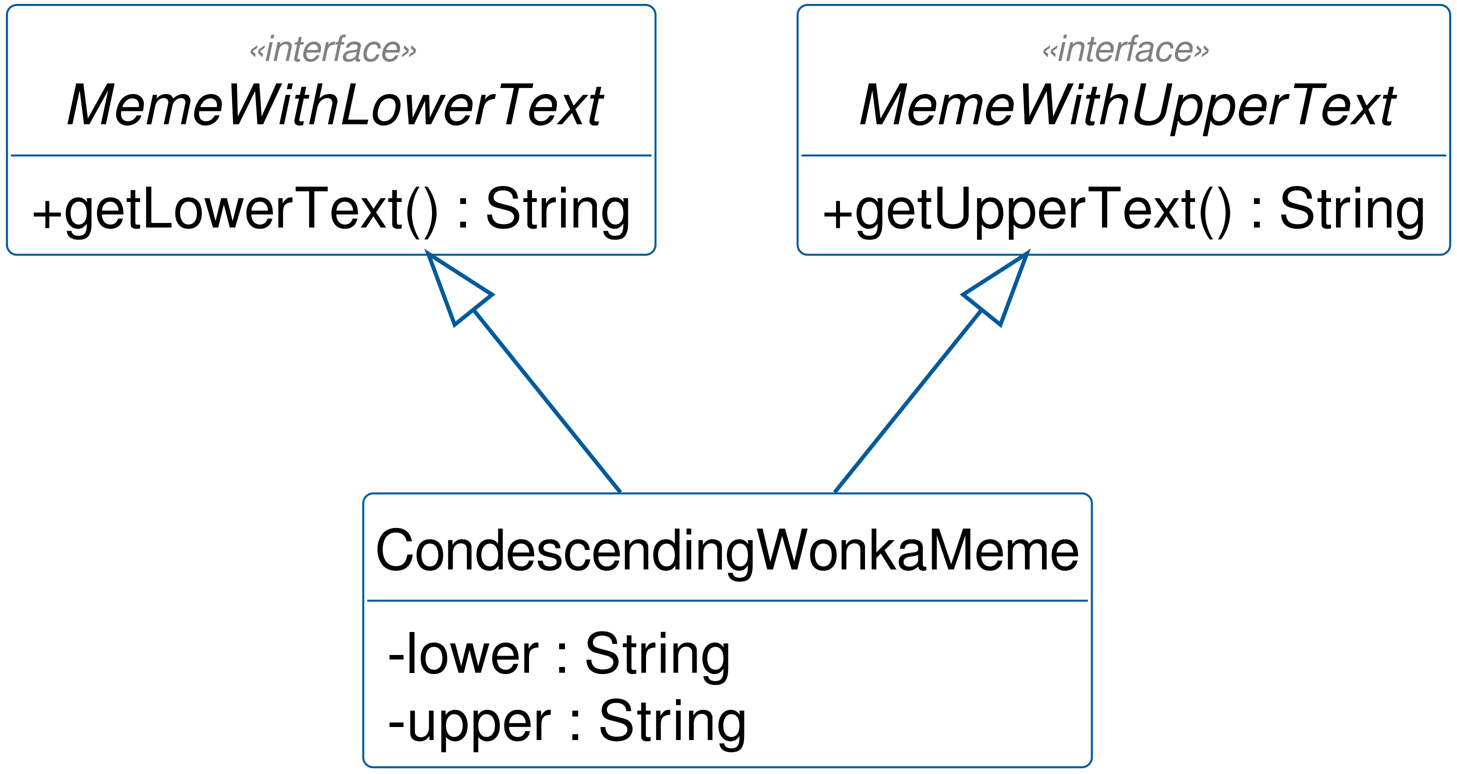
STEP 2: INTERFACE INHERITANCE

Implement the two interfaces inheriting from Meme.



STEP 3: IMPLEMENTING INTERFACES

Create the class `CondescendingWonkaMeme` which implements both `MemeWithLowerText` as well as `MemeWithUpperText`.



Implement the required methods in the class `CondescendingWonkaMeme`:

- Add a constructor that expects two strings as parameters: `upper` and `lower`
- Assign the constructor parameters to corresponding attributes
- Return the attribute `upper` in the method `getUpperText()`.
- Return the `lower` attribute in the `getLowerText()` method
- Use `getUpperText()` and `getLowerText()` and return their results as an array in the `getText()` method.
- Return the following string in the `getImageUrl()` method: `https://makeameme.org/media/templates/250/condescending_wonka.jpg`

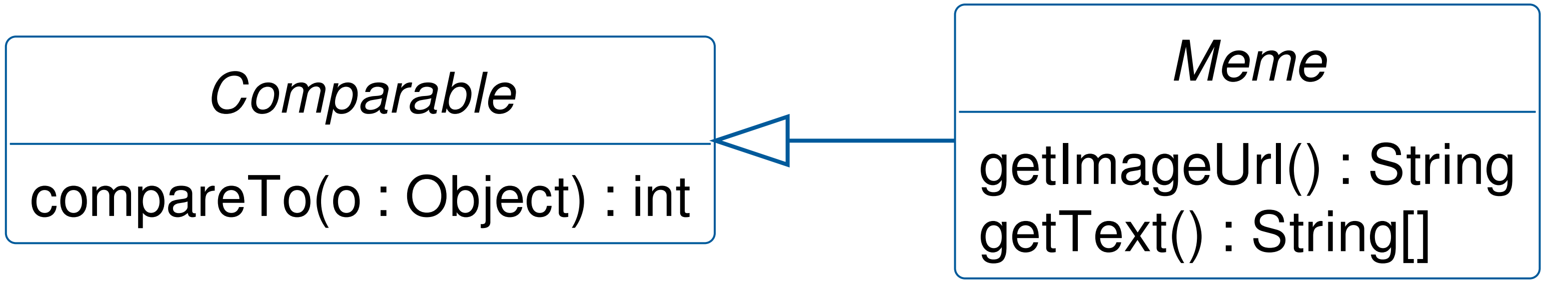
WHICH STATEMENTS ARE CORRECT?

```
Meme meme = new CondescendingWonkaMeme(
  "Oh you create memes in exercises?", "That's very unique");
```

- ☐ meme.getText() works
- ☐ meme.getLowerText() works
- ☐ ((MemeWithUpperText)meme).getLowerText() works
- ☐ ((CondescendingWonkaMeme)meme).getUpperText() works

STEP 4: MEME-COMPARABLE

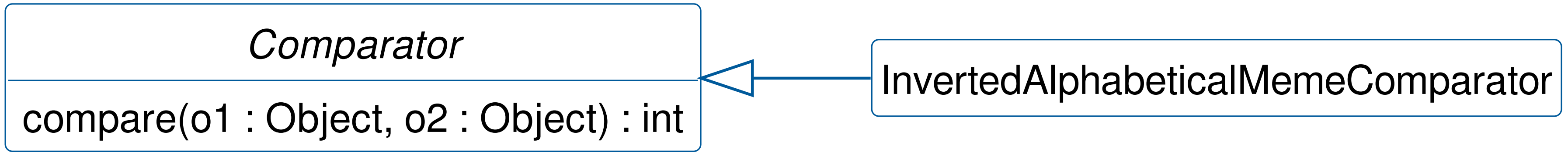
Extend `Meme` so that it inherits from `Comparable`.



Add an implementation so that memes are sorted alphabetically using `getText()`. Use `String.join` to convert the array into a string.

STEP 5: MEME-COMPARATOR

Add a comparator that sorts in reverse alphabetical order.


Comparator
compare(o1 : Object, o2 : Object) : int
InvertedAlphabeticalMemeComparator

Add an implementation so that memes are sorted in reverse alphabetical order using `getText()`. Use `String.join` to convert the array to a string.

5.7 EXERCISE: PEOPLE, ANIMALS AND CONSOLES

On the one hand there is a console application and on the other the two classes Person and Animal.

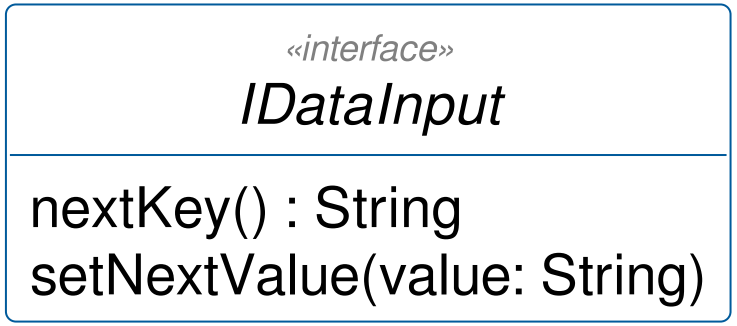
To the console application, the Person and Animal classes are unknown. Moreover, the Person and Animal classes do not inherit from a common parent class (with the exception of Object).

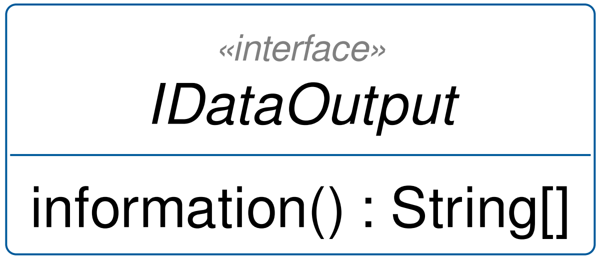
Idea: Use interfaces so that messages can be exchanged between the console application and persons / animals.

Unzip person-animal-console.zip and import the folder as "Existing Maven Project".

STEP 1: DATAINPUT AND DATAOUTPUT

Define the interfaces `IDataInput` and `IDataOutput`.





- an object of type `IDataInput` stores key/value pairs
- after creation all values are undefined
- `nextkey` returns a key of the next undefined value or null if all values are defined
- `setNextValue` sets the next currently undefined value

STEP 2: CLASS CONSOLEAPPLICATION

Implement the methods of class `ConsoleApplication`.

- `inputValuesFromConsole` takes a parameter of type `IDataInput`
  - Call the method `nextKey` as long as it does not return `null`
  - Use each return value as prompt to input a string from the console
  - Then store the input value using `setNextValue`
- `outputValuesToConsole` takes a parameter of type `IDataOutput`
  - Read all information using the `IDataOutput` interface
  - iterate over it and output the values line by line to the console

STEP 3: IMPLEMENT INTERFACES IN CLASS PERSON

Implement the interfaces `IDataInput` and `IDateOuput` in class Person.

- Method `nextKey` should return:
  - ... "Name", when attribute `name` is either empty or `null`
  - ... or "E-Mail", when attribute `email` is either empty or `null`
  - ... `null` otherwise
- Method `setNextValue` takes a parameter called `value` which should be:
  - ... assigned to attribute `name`, if it is either empty or `null`
  - ... or assigned to attribute `email`, if it is either empty or `null`
- Method `information` should return an array containing two Strings:
  - the first string should contain the value of the name attribute
  - and the second element should contain the value of the email attribute

Use class `Main` to read a person's attributes and output them to the console:

```
public static void main(String[] args) {
    Person person = new Person();
    ConsoleApplication controller = new ConsoleApplication();
    controller.inputFromConsole(person);
    controller.outputToConsole(person);
}
```

STEP 4: IMPLEMENT INTERFACES IN CLASS ANIMAL

Implement the interfaces `IDataInput` and `IDataOutput` in class `Animal`.

Use the Person class as a guide here. Note that the `age` attribute is of type `int`.

Use class `Main` to read a animal's attributes and output them to the console:

```
public static void main(String[] args) {
    Animal animal = new Animal();
    ConsoleApplication controller = new ConsoleApplication();
    controller.inputFromConsole(animal);
    controller.outputToConsole(animal);
}
```

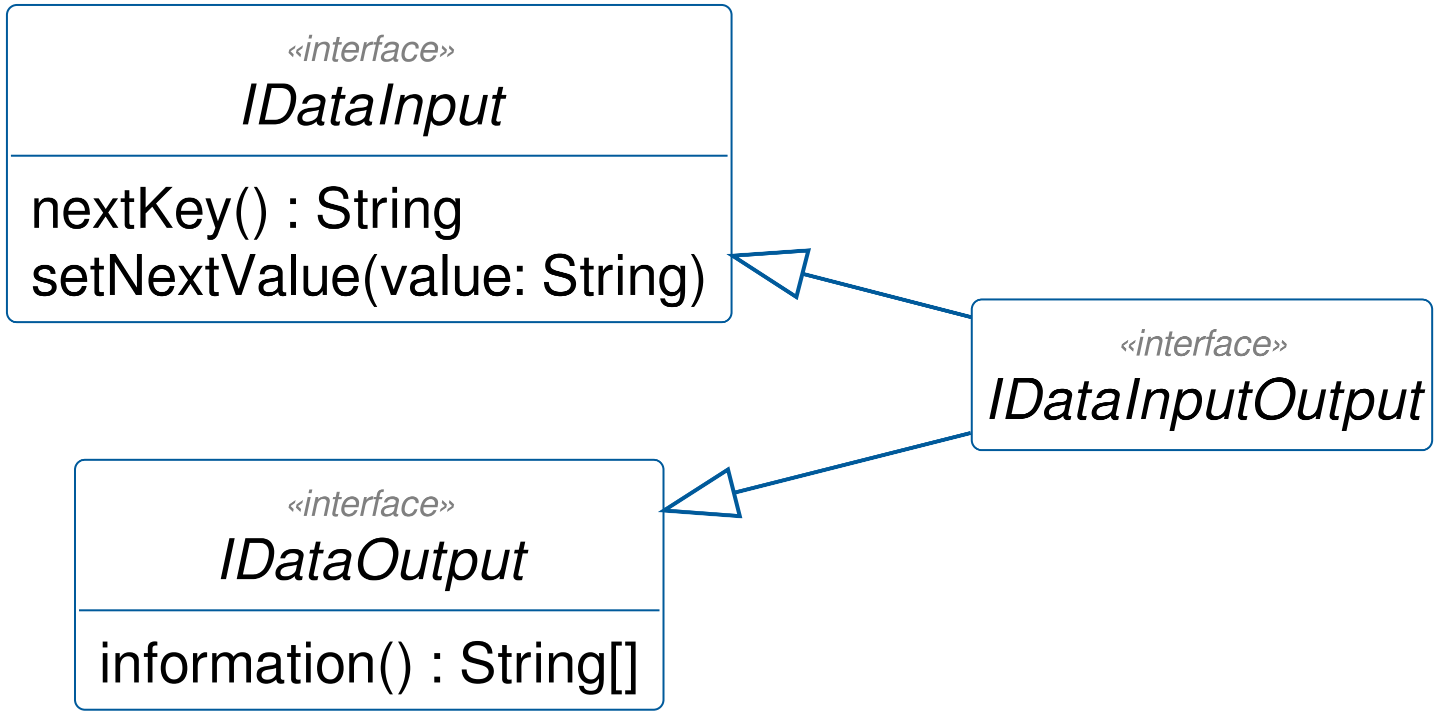
EXERCISE: ANIMAL, PERSON AND CONSOLEAPPLICATION

Which statements are correct?

- ☐ `ConsoleApplication` can be implemented without the classes `Tier` and `Person`
- ☐ `Person` can only be implemented after the `ConsoleApplication` class has been implemented
- ☐ `ConsoleApplication` shows compiler errors as long as the class `Person` has not yet implemented the interface `IDataInput`.
- ☐ `Person` and `Animal` inherit from a common parent class (not Object)

STEP 5: INTERFACE INHERITANCE HIERARCHY

Define a new interface `IDataInputOutput` specialising the interfaces `IDataInput` and `IDataOutput`.



Change the class `Animal` so that it only uses the interface `IDataInputOutput`.

What other changes are necessary?

- ☐ No changes necessary
- ☐ Changes to 'ConsoleApplication' necessary
- ☐ Changes to `Person` necessary
- ☐ Changes to `Animal` necessary

EXERCISE: COMPARABLE PERSON

Implement the interface `Comparable` in class `Person`, so that persons are compared by their `name` attribute when using `Arrays.sort` with an array of type Person.

What needs to be taken into account?

- ☐ `this` could equal `null`
- ☐ `other.name` could equal `null`
- ☐ `this.name` could equal `null`
- ☐ Parameter `other` could equal `null`

Sort the array of persons in `Main`

```
Arrays.sort(list);
```

EXERCISE: E-MAIL COMPARATOR

Create a class `EMailComparator` that implements the `Comparator` interface. Use the method `compare` to realise a sort order according to the attribute `email` for instances of the class `Person`.

What needs to be taken into account?

- ☐ `o1.getName()` could yield `null`
- ☐ Parameter `o2` could be `null`
- ☐ Parameter `o1` could be `null`
- ☐ `o2.getName()` could yield `null`
- ☐ A getter for `name` should be used
- ☐ `name` could have public visibility

Use the comparator in class `Main`

```java
Arrays.sort(list, new EMailComparator());
```