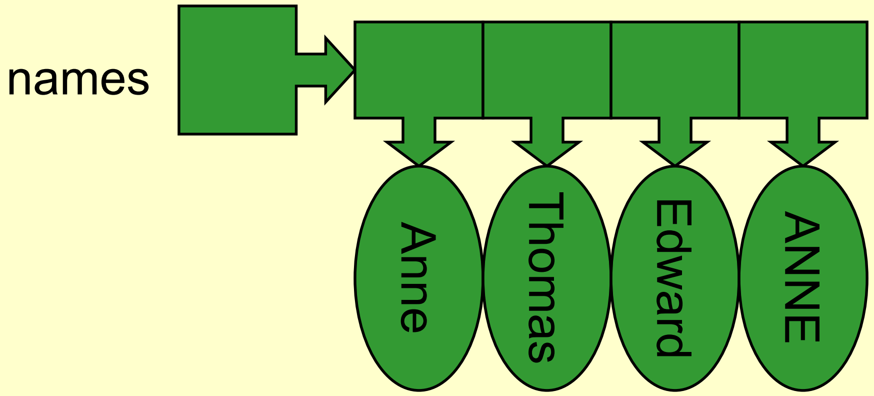
Java the UML Way

http://www.tisip.no/JavaTheUmlWay/

Arrays of Reference Types and Array Lists

An Array of References, Strings as an Example

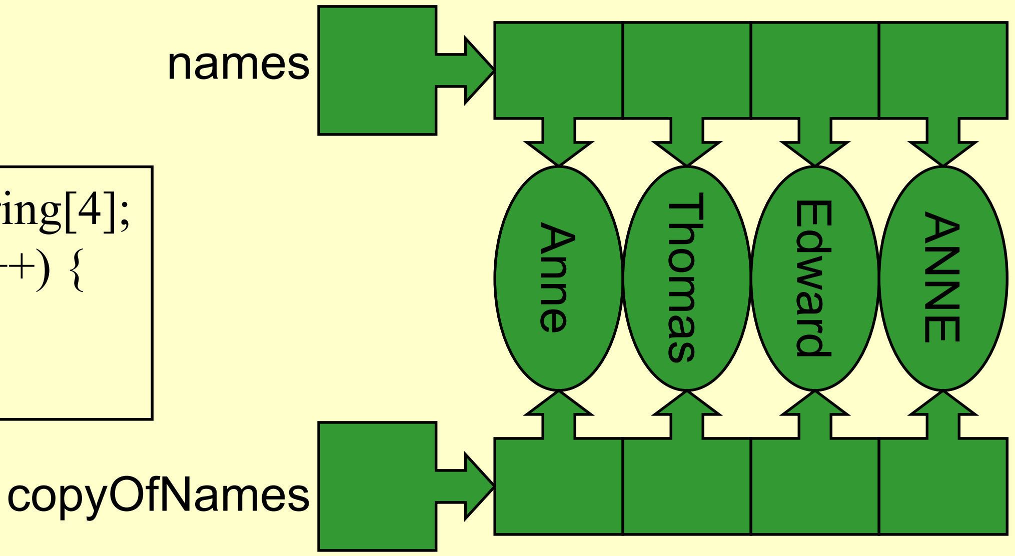
- Every element in the array is a reference to String:
  - String[] names = new String[4];
- This is an array of references. Every one of these has to point to an instance of the String class:
  - names[0] = new String(”Anne”);
  - names[1] = ”Thomas”; // short form is ok
- The reference may be a return value from a method creating a String object:
  - names[2] = JOptionPane.showInputDialog("Write a name: ");
  - names[3] = names[0].toUpperCase();



- Or we may write:
  - String[] names = {”Anne”, ”Thomas”, ”Edward”, ”ANNE”};

Copying an Array of References Does Not Copy the Objects

```
String[] copyOfNames = new String[4];
for (int i = 0; i < names.length; i++) {
  copyOfNames[i] = names[i];
}
```



Array Elements of Primitive Data Type vs. Reference Type

- An array of a primitive data type:
  - The elements in the array contain the data values.
  - Comparisons and assignments between elements directly affect the data values. The data values are compared and copied.
  - The elements in the array are initialized to 0 (or false) if nothing else is said in the declaration.
- An array of a reference data type:
  - The elements in an array of reference type are references, not objects.
  - The references in an array get the initial value null.
  - Every individual reference in the array has to be set to refer to an object before it can be used. If we try to use an array element that does not refer to any object, a NullPointerException is thrown.
  - We can't use comparison operators other than == and != on elements like these. The operator == returns true if the elements refer to the same objects.
  - Assignment means the reference changes value, in other words it is set to refer to another object. The objects remain untouched. They are neither moved nor copied.

Solve the problems 1-4, pp. 274-275.

What is an ArrayList?

- Ordinary arrays are not very flexible. The size cannot be changed after the array is created.
- An instance of the java.util.ArrayList class
  - has a built-in array of references.
  - maintains the size of this array as required, by itself.
  - knows how much of the built-in array is in use.
- Methods:
  - boolean add(Object obj) – insert a reference into the array list
  - Object get(int indeks) – get a reference from the array list
  - Object remove(int indeks) – remove a reference from the array list
  - int size() – the number of references stored in the array list
- If invalid index in the get() or remove() methods, an IndexOutOfBoundsException is thrown
- If a parameter is of the Object type (java.lang.Object), the argument can be from any reference type whatsoever.
- If the return type from a method is Object, we have to cast the return object to the right class before we can send messages to it.

An Example

```
import java.util.*;
class TestArrayList {
  public static void main(String[] args) {
    String name1 = new String("Edward");
    String name2 = new String("John");
    String name3 = new String("Elizabeth");
    ArrayList name = new ArrayList();
    name.add(name1);
    name.add(name2);
    name.add(name3);
    for (int i = 0; i < name.size(); i++) {
      String aName = (String) name.get(i);
      System.out.println(aName);
    }
  }
}
/* Printout:
Edward
John
Elizabeth
*/
```

Problem: Draw a figure showing all references and objects in this program.

Solve all problems, page 278.

What About Storing Numbers in an ArrayList?

- The elements in an array list have to be references.
- The classes Integer, Double, etc *wrap* a value of a primitive data type into an object. These classes are *wrapper classes* for the primitive data types.
- References to these *wrapper objects* may be stored in an array list.
- Examples:
  - Integer integerObject = new Integer(50); // creates an object with value 50
  - int number = integerObject .intValue(); // fetches the number value out of the object
- We input positive numbers from the user and store them in an array list:

```
ArrayList numbers = new ArrayList();
int aNumber = InputReader.inputInteger("Write a positive number: "); // see chap. 6.8
while (aNumber > 0) {
  Integer integerObject = new Integer(aNumber);
  numbers.add(integerObject);
  aNumber = InputReader.inputInteger("Write a positive number: ");
}
```

The Wrapper Classes for the Primitive Data Types

- Class names: Integer, Long, Float, Double, Byte, Character and Boolean
- The classes belong to the java.lang package.
- All the classes are immutable.
- Every class has a constructor taking a value of the corresponding primitive data type as an argument.
- Every class, except for Character, has a constructor taking a string as argument. The constructor throws a NumberFormatException if conversion is not possible.
- An example, the Integer class:
  - Integer(int value)
  - Integer(String s) throws NumberFormatException
  - String toString()
  - static int parseInt(String s) throws NumberFormatException
  - static String toString(int i)

- An example:

```
String t = "234_567";
try {
  int number = Integer.parseInt(t);
  System.out.println("The number is " +
      number);
} catch (NumberFormatException e) {
   System.out.println("Cannot convert "
              + t+ " into number");
   System.out.println(e);
}
```

- Printout:

```
Cannot convert 234_567 into number
java.lang.NumberFormatException: 234_567
```

Solve problem 1, page 281.

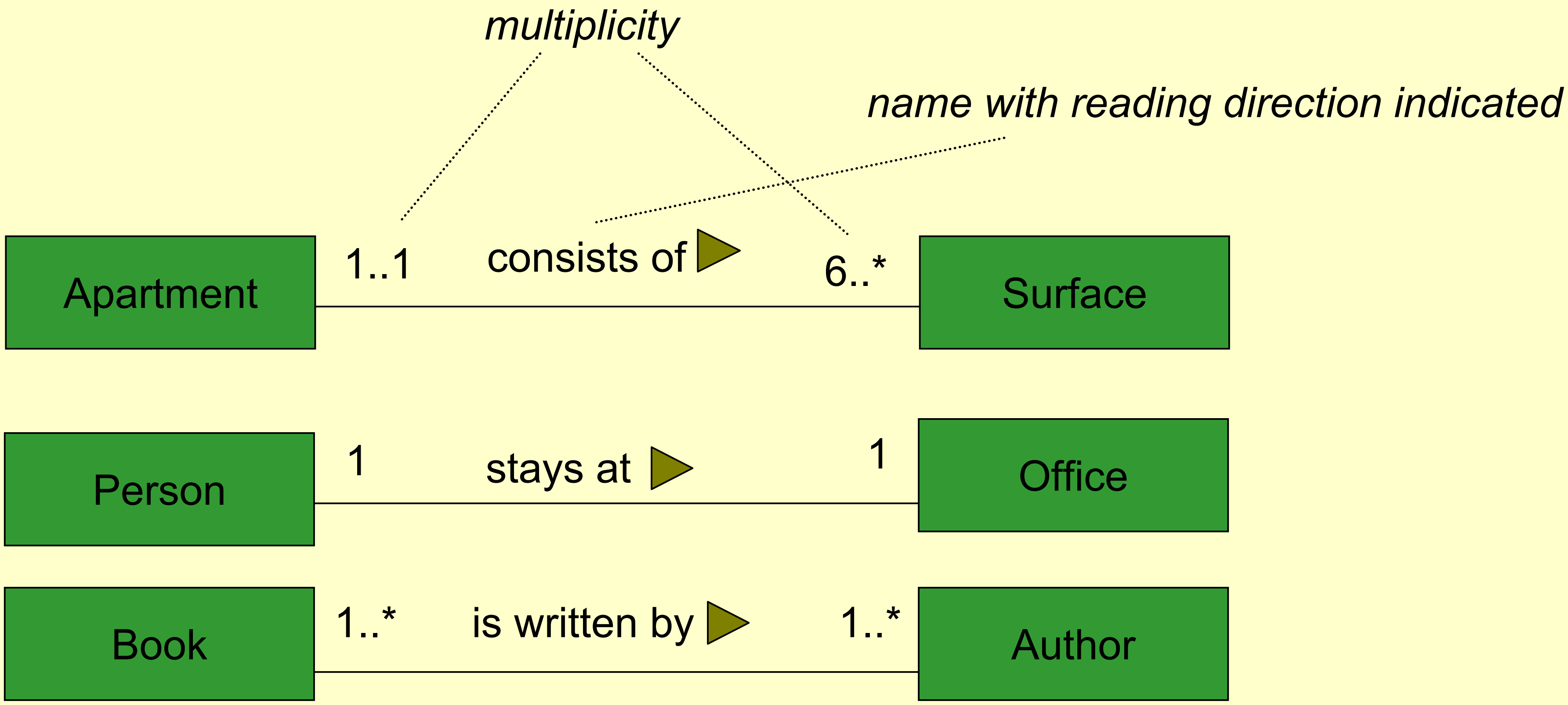
The equals() and toString() Methods

- All classes are subclasses of the java.lang.Object class
- All classes inherit the methods of the Object class, among others:
  - public String toString()
    - returns the name of the class the object is an instance of, followed by @ and a number code
  - public boolean equals(Object obj)
    - compares the reference this with the reference obj, returns true if they are equal
- *But* – a class can (and should!) have its own versions of these methods. If it has, it is these versions which apply for instances of the class.
- The methods should be programmed with this functionality:
  - toString() should return a string which the client could use to get a printout of the contents of the object.
  - equals() should compare the contents of the objects
- The toString() method is implied
  - if we link an object to a string with the + operator, e.g.:
    ```
    string result = "The account: " + acconut1;
    ```
  - if we use an object as argument to System.out.println(), e.g.:
    ```
    System.out.println(account1);
    ```

Solve the problem, page 283.

Associations

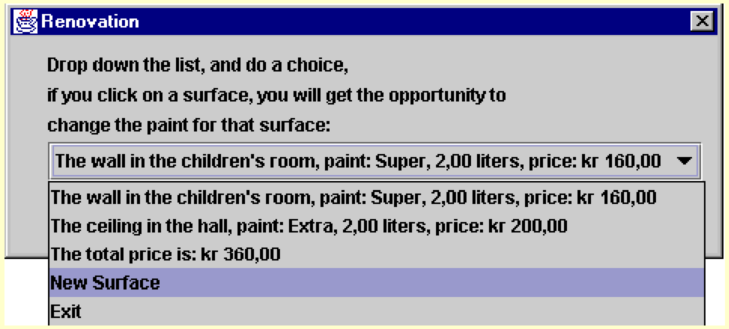
- An association between classes means that there is a connection between instances of the classes. Examples:



The Renovation Case, One More Time

Input about surface: name, length, width, type of paint.
Input about type of paint: name, no. of coats, no. of sq.m/l, price per liter

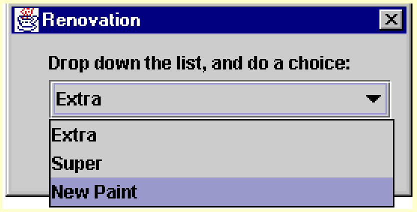
A wall and a ceiling with different type of paint



Renovation

Drop down the list, and do a choice,
if you click on a surface, you will get the opportunity to
change the paint for that surface:

```
The wall in the children's room, paint: Super, 2,00 liters, price: kr 160,00
The wall in the children's room, paint: Super, 2,00 liters, price: kr 160,00
The ceiling in the hall, paint: Extra, 2,00 liters, price: kr 200,00
The total price is: kr 360,00
New Surface
Exit
```

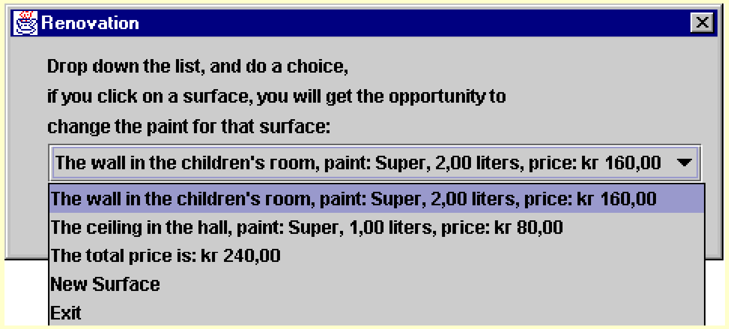


Renovation

Drop down the list, and do a choice:

```
Extra
Extra
Super
New Paint
```

Now we use the same paint on both surfaces:



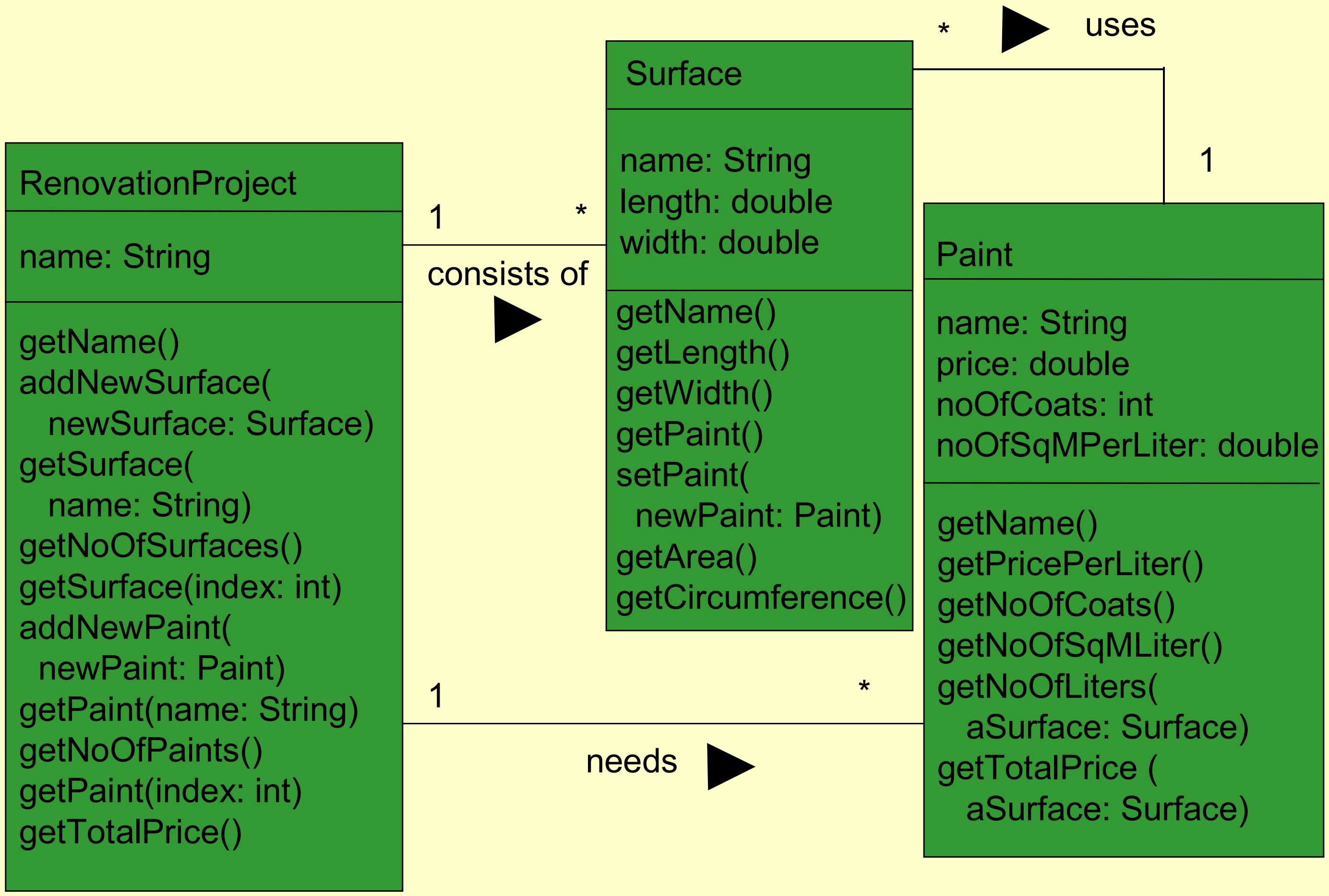
Renovation

Drop down the list, and do a choice,
if you click on a surface, you will get the opportunity to
change the paint for that surface:

```
The wall in the children's room, paint: Super, 2,00 liters, price: kr 160,00
The wall in the children's room, paint: Super, 2,00 liters, price: kr 160,00
The ceiling in the hall, paint: Super, 1,00 liters, price: kr 80,00
The total price is: kr 240,00
New Surface
Exit
```

The Renovation Case, Associations

We separate classes describing the user interface from classes describing the problem we're solving. This makes it easy to change between different kinds of user interfaces. Here we look at classes describing the problem:



Programming Associations

- A register emerges as a one-to-many association between the class that is responsible for maintaining the register and the class that is describing the objects that are included in the register.
  - An instance of the RenovationProject class maintains two registers, one containing information about the surfaces, the other containing information about the different types of paint.
- We create a register by creating an array list. This array list will contain references to the objects that are stored in the register.

```
class RenovationProject {
  private String name;
  private ArrayList allSurfaces = new ArrayList();
  private ArrayList allPaints = new ArrayList();
```

- The association between Surface and Paint should be programmed in another way. An instance of the Paint class doesn't need to know about all the Surface instances. The opposite is true: A Surface object needs to know with which type of paint it will be covered. We program this by creating a reference to this object in the Surface class:

```
class Surface {
  private String name;  // for identification purposes
  private double length;
  private double width;
  private Paint paint;
```

Which Tasks Should a Register Object be Able to Perform?

- In general, a client would ask for the following services:
  - Adding data to the register.
  - Looking at all data already stored in the register.
  - Searching in the register, given e.g. a name.
  - Removing data.
  - Editing data.
- We'll look at the registers in the renovation case as an example. (Removing and editing data are not implemented, but see the problems on page 295.)

Show Program Listing 10.1, pp. 289-291.

- *Problems:*
  - In a client program we have an object:
    - RenovationProject project = new RenovationProject (”MyHouse”);
  - Several surfaces and several types of paint are already registered.
  - Expand the client program with the following:
    1. Create and register a new surface object. You'll need constructors:
       - public Surface(String initName, double initLength, double initWidth)
       - public Paint(String initName, double initPrice, int initNoOfCoats, double initNoOfSqMPerLiter)
    2. Find out if a paint with name”Huldra 8” exist?
    3. Print the names of all the registered surfaces.
    4. Expand the loop with the name of the paint that is used for every surface.

Interface

- A Java *interface*, simply put, is a collection of method heads.
- A class can choose to implement an interface. Then all the methods in the interface have to be programmed.
- An example from the java.lang package is:

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

- Example of a class implementing this interface:

```
class Surface implements Comparable {
public int compareTo(Object obj) {  // comparing areas
      Surface theOtherSurface = (Surface) obj;
      double area1 = getArea();  // the area of this
      double area2 = theOtherSurface.getArea();
      if (area1 < area2 - 0.0001) return -1;     // comparing decimal numerals
      else if (area1 > area2 + 0.0001) return 1;
      else return 0;
}
```

Solve the problem, page 297.

How is it Possible to Create Generic Methods for Sorting and Searching?

- A generic method for sorting or searching will have to compare objects.
- Different kinds of objects should be compared in different ways.
- An instance of one class should be compared by another instance of the same class by sending a message to the first instance:

```
name1.compareTo(name2)
```

- The class which the instances belong to, should offer the compareTo() method.
- A generic method for sorting or searching is possible if the head of the comparing method is the same for all types of objects.
- We create an interface containing the method head, and then we require that all the objects which should be sorted (or searched in) belongs to classes that implement this interface.
- Comparable is just this interface:

```
public interface Comparable {   // belongs to the java.lang package
  public int compareTo(Object obj);
}
```

Sorting and Searching Methods from the Java API, Using Comparable

- The objects should belong to classes implementing the Comparable interface.
- The java.util.Arrays offers methods for arrays:
  - static int binarySearch(Object[] array, Object[] value)
  - static void sort(Object[] array)
  - static void sort(Object[] array, int fromIndex, int toIndex)
    - Exceptions may be thrown: ArrayIndexOutOfBoundsException, IllegalArgumentException, ClassCastException
- The java.util.Collections offers methods for array lists:
  - static int binarySearch(List arrayList, Object value)
  - static void sort(List arrayList)
    - The ArrayList class implements the List interface, therefore it is ok that the first parameter is of the List type.

An Alternative to Comparable

- We create an object containing complete information about the sorting order of "the main objects".
- Such an object has to belong to a class implementing the Comparator interface:

```
public interface Comparator {   // belongs to the java.util package
  public int compare(Object o1,Object o2)
  public boolean equals(Object obj)
}
```

- The java.text.Collator is a useful example of a class implementing the Comparator interface
  - An instance of this class contains information about the sorting order of strings according to a given location.
    - public static Collator getInstance()
    - public static Collator getInstance(Locale givenLocation)
    - public int compare(String text1, String text2)
    - public void setStrength(int newStrength) // PRIMARY, SECONDARY, TERTIARY

Sorting Strings According to the Default Settings

```
public static void sortIntegerArray(int[] array) {
  for (int start = 0; start < array.length; start++) {
    int smallestToNow = start;
    for (int i = start + 1; i < array.length; i++) {
      if (array[i] < array[smallestToNow]) smallestToNow = i;
    }
    int help = array[smallestToNow];
    array[smallestToNow] = array[start];
    array[start] = help;
  }
}
```

Sorting integers, pp. 253-254.

```
public static void sortTextLocale(String[] array) {
  Collator collator = Collator.getInstance();
  for (int start = 0; start < array.length; start++) {
    int smallestToNow = start;
    for (int i = start + 1; i < array.length; i++) {
      if (collator.compare(array[i], array[smallestToNow]) < 0) smallestToNow = i;
    }
    String help = array[smallestToNow];
    array[smallestToNow] = array[start];
    array[start] = help;
  }
}
```

Sorting strings, pp. 300-301.

Sorting and Searching Methods from the Java API, using Comparator

- These methods use a ”Comparator” to compare the objects.
- The java.util.Arrays offers methods for arrays:
  - static int binarySearch(Object[] array, Object[] value, Comparator comp)
  - static void sort(Object[] array, Comparator comp)
  - static void sort(Object[] array, int fromIndex, int toIndex, Comparator comp)
    - Exceptions may be thrown: ArrayIndexOutOfBoundsException, IllegalArgumentException, ClassCastException
- The java.util.Collections offers methods for array lists:
  - static int binarySearch(List arrayList, Object value, Comparator comp)
  - static void sort(List arrayList, Comparator comp)
    - The ArrayList class implements the List interface, therefore it is ok that the first parameter is of the List type.

Solve all problems, page 303-304.