Computer Science for Engineers
Exercise 9 and 10


Prof. Dr. Dr.-Ing. J. Ovtcharova
Dipl. Wi.-Ing. Dan Gutu
12th and 13th of February 2009
Outline of Exercise 9 and 10

Content of Exercise 9 and 10

1. Threads
2. Packages
3. Collections. Generic Collections
4. Data Structures
5. Graphical User Interface
6. UML
What is the problem which is solved by Multithreading?

- People can do several things simultaneously.

  **Example:** driving a car
  - How many people can drive a car?
  - How many people can amuse themselves while driving?

  **Example:** walking
  - How many people can amuse themselves while walking?
  - ... Thinking while walking?
  - ... Moving the arms while walking?

- It would be difficult to achieve anything, without doing things simultaneously.
  - People are always doing different things simultaneously.
Multithreading - example

• Racing numbers problem
  Two columns should be shown on the board simultaneously with numbers

Possible solutions:

• A person with one piece of chalk at first writes the left column and then the right column.
  - Problem not solved.
  - The columns „race“, and the left column always „wins“.

• A person with two pieces of chalk fills the columns line by line.
  - Problem not solved.
  - The column which was begun first always wins.

• Another person writes the other column.
  - This solves the problem!
  - It can not be forecast who is more rapid.
Lifecycle of a thread (1)

• A new thread emerges by creating a new instance of the class `java.lang.Thread`.

• The `thread`-object represents a real thread in the Java interpreter and serves to control and synchronize its execution.

• With this `thread`-object, the thread can be started, stopped or interrupted temporarily.

• The constructor of the class `Thread` contains information about the starting position of the thread.
Lifecycle of a thread (2)

- An object that wants to be the aim of a thread must declare that it has an appropriate run method by implementing the interface `java.lang.Runnable`.

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

- A newly created thread remains idle until its `start()` method is applied.

- The `start()`-method does some initialization and then calls the method `run()` of the associated object of the thread.

- Any thread having been started continues executing as long as the method `run()` is finished or the method `stop()` is invoked to irrevocably destroy the thread.
The class Thread

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>start()</code></td>
<td>Start or stop thread. Both methods can only be called once for each thread</td>
</tr>
<tr>
<td><code>stop()</code></td>
<td>Interrupt or resume the execution of the thread. Both methods may be called arbitrarily often.</td>
</tr>
<tr>
<td><code>suspend()</code></td>
<td>Make a thread sleep for a certain period of time.</td>
</tr>
<tr>
<td><code>resume()</code></td>
<td>Modify or ask priority of the thread.</td>
</tr>
<tr>
<td><code>sleep(long)</code></td>
<td></td>
</tr>
<tr>
<td><code>setPriority(int)</code></td>
<td></td>
</tr>
<tr>
<td><code>getPriority():int</code></td>
<td></td>
</tr>
</tbody>
</table>
Multithreading (1)

- A thread is the strand of a program
- A thread is NO PROCESS! A process has its own memory space, its own resources! Several threads can run inside a process
- Situation without threads:
  - It is possible to let drive one car at a time, but it is not possible to let drive several cars simultaneously
- Situation with threads:
  - Several cars can drive simultaneously
- Threads permit branching inside a program. The program flow is not linear (top-down), but contains several branches that are performed in parallel.
- Inside a thread, new objects can be created
- Without threads, the entire execution of the program would wait for a method to terminate. With threads, methods belonging to different objects can run simultaneously.
Multithreading (2)

• The thread needs an object that implements the interface **Runnable**
  - The interface **Runnable** already exists in Java, it must not be implemented

• A thread runs as long as its method **run()** implemented through the interface **Runnable** is processed

• Threads can “sleep”. A sleeping thread waits until it is woken up or until its sleeping time is over

• Threads can be synchronized between each other and can communicate with each other.

• **Advantages:**
  - Several methods can run simultaneously
  - System resources can be optimally used, avoiding processor idle time
  - On multiprocessor systems, threads can run on different processor cores

• **Disadvantages:**
  - Errors occurring due to parallel processing by threads are very hard to be traced and understood; debugging is complicated, too.
Multithreading – parallel executions

- A thread corresponds to a control flow within a program.

- Using threads one has the possibility to realize quasi-parallel execution in the same address space, i.e. they share instance variables, but not local variables (ex., of the method executed several times in parallel).

- Having different threads within one single program, it has to be guaranteed that the access to variables and methods can be synchronized.

\[
\begin{align*}
  &a = 1; \\
  &a = a + 1;
\end{align*}
\]
assuming thread 1 and 2 manipulate the variable a quasi simultaneously.

Result is not definite:

- Threads carry out sum in sequence: Result = 3
- Threads carry out sum simultaneously: Result = 2
Multithreading - monitors

- The critical blocks are those whose statements manipulate common data. In Java, they can be provided with so called monitors that enable the execution of critical blocks by only one thread at a time.

**Example:**
- Program consists of blocks a, b, c.
- Block b is critical.
- 2 threads carry out a program simultaneously.

At entry of thread 1 in block b, the monitor gets blocked.

After ending of the critical block, the monitor is released.

Thread 2 can not enter blocked critical block b.

At entry of thread 2 in block b, monitor gets blocked again (this time by thread 2).

After ending of the critical block, the monitor is released.
Synchronization of threads

• Synchronization is needed to serialize the access to a certain resource, i.e. an object.

• By synchronizing access to common resources it is ensured, that certain actions can be performed only by one single thread at any time.

• In Java, each class and each instance of a class possesses its own monitor.

• Using the key word `synchronized` the critical blocks are determined that have to protected from simultaneous execution by different threads.
Synchronization of threads - example

- Implementation of a class `SpeechSynthesizer` that contains a method `say()`.

- This method should not be called by many threads simultaneously because this could cause problems with understanding what is said.

```java
class SpeechSynthesizer {
    synchronized void say( String words ) {
        // speaking
    }
}
```

- Having finished the execution of the method `say()` the next thread can apply for the access to this method.
Synchronization of methods of the same class

- Only one of many synchronized methods is executed simultaneously.

- All static methods of a class use the same monitor, belonging to the class. And all instance methods of a class use the same monitor, belonging to the respective instance of the class.

```java
class SpreadSheet {
    int cellA1, cellA2, cellA3;

    synchronized int sumRow() {
        return cellA1 + cellA2 + cellA3;
    }

    synchronized void setRow( int a1, int a2, int a3 ) {
        cellA1= a1; cellA2= a2; cellA3= a3;
    }
}
```

Both methods must be marked as synchronized.
Synchronization of blocks of code

• The key word `synchronized` can not only be used for synchronizing methods but for supervising arbitrary blocks of code. This can be more efficient, since objects don’t have to be blocked for the whole execution time of the method.

• In this version, an argument is passed to `synchronized` explicitly representing the object for which a monitor is to be assigned.

```java
synchronized (myObject) {
    // actions to be synchronized,
    ...
}
```

For the object indicated as parameter of `synchronized`, threads must obtain a monitor before manipulation.
public class Animation implements Runnable {
    
    public void run() {
        while (true) {
            // draw the different images 
            
        }
    }
}

The class Animation implements the method run() to control the loop to draw images.

public class test {
    public static void main(String args[]) {
        Animation anim = new Animation();
        Thread myThread1 = new Thread(anim);
        Thread myThread2 = new Thread(anim);
        myThread1.start();
        myThread2.start();
        
    }
}

Two thread objects with an instance of the class Animation as target object are created and their methods start() are invoked. The start method automatically invokes the run method of the associated object.
Example of threads in Java (2)

• The same example in other variants:

```java
class Animation implements Runnable {
    Thread myThread1, myThread2;
    Animation() {
        myThread1 = new Thread(this);
        myThread2 = new Thread(this);
        myThread1.start();
        myThread2.start();
    }
    ...
}
```

Object controls is owned thread.

```java
class Animation extends Thread {
    ...
    public void run() {
        while ( true ) {
            // draw the different Images
            ...
        }
    }
}
```

Definition of the target class as subclass of the class `Thread`.

```java
public class test {
    public static void main(String args[]) {
        Animation anim1 = new Animation();
        Animation anim2 = new Animation();
        anim1.start();
        anim2.start();
        ...
    }
}
```
Outline of Exercise 9 and 10

Content of Exercise 9 and 10

1. Threads
2. Packages
3. Collections. Generic Collections
4. Data Structures
5. Graphical User Interface
6. UML
• Name conflicts are a main problem with the development of re-usable code.

• The standard solution for name collisions is in many programming languages the use of a package prefix in front of each class.

• However, the problem of name collisions is only shifted from class names to package names

• Thus in Java a more formal packet concept was defined.
• A package consists of several classes and subpackages.

• Packages are named and can be imported.

• Package names are built up hierarchically; the components are separated by dots.

• If a part of a package is used, the full-qualified name is given or the whole package (or a part of it) is imported.
• Packages also earn a share of the structure in a class’s library

• Packages contain
  – classes
  – sub-packages

• Packages
  – can be labeled
  – and can be imported

• Package names are
  - systematically given based on a hierarchy
  - components are separated by periods (.)

• If a part of the package is being used, then either the entire package, the class from the package with the fully given corresponding name, or a part of the package will be imported.

Example

java.util
class Date1 {
    public static void main ( String[] args ) {
        java.util.Date now= new java.util.Date();
        System.out.println(now);
    }
}

import java.util.Date;

class Date2 {
    public static void main ( String[] args ) {
        Date now= new Date();
        System.out.println(now);
    }
}
The access to variables and methods in classes is regulated by so-called access modifiers:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public</code></td>
<td>Access for everyone</td>
</tr>
<tr>
<td><code>private</code></td>
<td>Only within the same class</td>
</tr>
<tr>
<td><code>protected</code></td>
<td>Only within same class, package and subclasses</td>
</tr>
<tr>
<td><code>default</code></td>
<td>Only within same class and package, but <strong>not</strong> in the subclasses, if they are placed in another package</td>
</tr>
</tbody>
</table>
### Java Method and Attribute Visibility

<table>
<thead>
<tr>
<th>Situation</th>
<th>public</th>
<th>default</th>
<th>protected</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessible for non-subclass from the same package?</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Accessible for subclass from the same package?</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Accessible for non-subclass from another package?</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Accessible for subclass from another package?</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Inherited from superclass in the same package?</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Inherited from superclass in another package?</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>
Outline of Exercise 9 and 10

Content of Exercise 9 and 10

1. Threads
2. Packages
3. Collections. Generic Collections
4. Data Structures
5. Graphical User Interface
6. UML
Method chaining

- Methods can be chained to each other directly, without catching the return value in a variable. In a method call, another method can be inserted.

```
World.getCollection().add(anObject);
```

*Returns a Collection-object*  
*Calls a method on the previously called object*
Collections

• Motivation
  - Mapping in UML

• What are Collections?
  - Collections types

• How are Collections used?

• How can Collections be accessed?
Motivation

- Relations between classes can be of multiple cardinality
  - Several students attend a lecture
  - A professor reads several lectures
- Collections allow the mapping of these circumstances in Java
  - An object of the class Lecture (Vorlesung) has a collection of students
Options for m:n relations

• Define several attributes of same data type
  - Bad solution, because:
    it is not possible to access the attributes over a loop (the name of the object is available only in the source code, no loop can be made over different attributes)

• Alternatives: an object shall contain other objects
  - A list or a table contains a number of elements of similar data type

StudentsList

<table>
<thead>
<tr>
<th>Name</th>
<th>Surname</th>
<th>MatrNr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rodriguez</td>
<td>Bender, B</td>
<td>21</td>
</tr>
<tr>
<td>Wong</td>
<td>Amy</td>
<td>23</td>
</tr>
<tr>
<td>Fry</td>
<td>Phillip J.</td>
<td>42</td>
</tr>
</tbody>
</table>
The class `Student`

- `public class Student()` {
  
  private String name;
  
  private String surname;
  
  private int MatrNr;
  
  // ..... 

  }

`public class Lecture` {

  // contains a Container with several Student -Objects.
  
  private Vector students;

  }

• An object containing a number of other objects
  - A Collection is a **container** for other objects
  - It contains objects and offers methods to add or remove objects
• **Collection** is a Java interface that is integrated in several implementations
  - Ex. Map, List, Set, Vector
  - It is contained in the package java.util

• In the Java Standard API there is a Collection Framework containing different interfaces, as well as their implementations

• These implementations have advantages and disadvantages:
  - Ex. random access, sequential access, sorting/ not sorting, inserting objects, indexing, …

• A Collection-object can contain several other (arbitrary) objects.
Collections (3)

Sun Collections
A Vector implements the interface Collection and summarizes several objects.

The interface defines access possibilities on the Collection-objects.

Java offers several Collection – implementations that can be used, for ex. Vector, ArrayList. These objects implement the methods of the interface.

A Collection offers methods to allocate other objects from the Vector or to remove them:
- Ex. `add(...)`, `addAll(...)`

In a Collection, the elements are stored as instances of the class `Object`. 
Access the Collections

- Objects of a Collection can be accessed by an iterator.
- An iterator is an object that can be interpreted as a pointer on the elements. With the iterator, navigation is made possible through a Collection.
The iterator is placed in front of the first element and is made available by the Collection:

```java
Iterator iter = collectionObject.iterator();
```

Calling `hasNext()` on the iterator delivers `true`, if there exists another object in the Collection. If the iterator points on the last element, the method `hasNext()` returns `false`.

With the method `next()`, the iterator points to the next element.
Iterators

- The iterator belongs to the Collection and can be inquired by the Collection

  - `Iterator iter = collectionObject.iterator();`

- If an iterator is fetched from a Collection, it points on the position in front of the first object. If the method `next()` is called for the first time, the first object is returned

  - `Iterator iter = collectionObject.iterator();
    Object o = iter.next() // o is the first object of the Collection`

- The class `Iterator` is a standard class in Java and is contained in the package `java.util`. It must be imported:

  - `import java.util.Iterator;`
Accessing objects in a Collection

- Collections contain elements of type *Object*

- *All classes in Java inherit from the upper class Object!*

- If objects are extracted from a Vector, they are of type Object and must be transformed (type-casted) into their initial data type
  - `Student stud = (Student) objectfromtheVector;`
  - `Ship s = (Ship) iterator.next();`
Example Iterators

// returns an iterator for the Collection ships
Iterator iter = ship.iterator();

// Loop “while next element existent”
while (iter.hasNext()) {
    // fetch next element from the vector
    // from the Vector, the ship is returned as instance of the class
    // object. It must be type-casted back to an object of the class
    // Ship.
    Ship s = (Ship) iter.next();
    // a simple output
    System.out.println(s.getName());
}
Generic Collections (1)

• Since Java 1.5, Collection can be parametrised
  - A Vector is created for a certain type (class)
  - The Vector can contain only object of type of that class or subclasses of it

```java
private Vector<Car> cars;
cars = new Vector<Car>();
// permitted
cars.add(new Car());
// not permitted
cars.add(new Ship());
```
- The parameter that determines which object type can be stored in the Vector is declared and initialized between \(<\>\).

- Declaration:
  - `Vector<Parameter_Class> vector;`

- Initialisation
  - `vector = new Vector<Parameter_Class>();`

- Iterators must be allocated with parameters, too. Type-casting is not necessary any more.
  - `Iterator<Parameter_Class> iter = vector.iterator();`
  - `Parameter_class object = iter.next();`
The big advantage of generic Collections is the type security

- Without a parameter, objects of different classes could be inserted in the Collection. When extracted with an iterator, the type casting might than fail.

```java
// fetch the next element from Vector
Object o = iter.next();
// type-casting; if the instance o is not an instance of Ship, a ClassCastException
// is thrown, the conversion fails.
```
// fetch an iterator for Cars from the Collection
// the iterator is parameterized, it will return an instance of Car

Iterator<Car> carIter = cars.iterator();

// no type-casting is required, the iterator does not return an instance of
// class object, but an instance of the class Car
For loop

- In Java 1.5, a new for loop type was introduced for generic Collections.
- It means „for all objects o in the Collection“. Ex.: „for all Car cars on street“
- The Car car is an object that is fetched from the Collection during a pass.

```java
Vector<Car> cars = street.getCars();
for (Car car : cars) {
    car.drive();
}
```
For loop – step by step

Vector<Car> cars = street.getCars();

for (Car car : cars) {
    car.drive();
}

• We have a generic Vector `cars`, received in each loop passage from the object `street`. In this example, the class `street` contains an attribute of type `Vector<Car>`

• `for (Auto auto : autos)`

The loop traverses the Vector `cars`. At each passage, the actual object from the Vector is assigned to the variable `car`. The instance `car` can be used inside the loop.
Iterators and for loops

Vector<Car> cars = street.getCars();

for(Car car : cars ) {
    car.drive();
}

Corresponds:

Vector<Car> cars = street.getCars();
Iterator<Car> iter = cars.iterator();
while(iter.hasNext()) {
    Car car = iter.next()
    car.drive();
}
Literature for Collections

• http://www.jeckle.de/vorlesung/java/script.html#collectionAPI
• http://java.sun.com/j2se/1.4.2/docs/api/java/util/Vector.html
• http://www.dpunkt.de/java/Die_Sprache_Java/Objektorientierte_Programmierung_mit_Java/50.html
• http://www.dpunkt.de/java/index.html
Outline of Exercise 9 and 10

Content of Exercise 9 and 10

1. Threads
2. Packages
3. Collections. Generic Collections
4. Data Structures
   4.1. Lists
   4.2. ArrayList
   4.3. Stacks and Queues
   4.4. Hash-Tables
   4.5. Tree Structures
   4.6. Net-like Data Structures
   4.7. Objects
5. Graphical User Interface
6. UML
Important implementations of list in Java:

- Interface `java.util.List`
  - Class `java.util.Vector`
  - Class `java.util.Stack`
  - Class `java.util.LinkedList`

Elementary methods for the administration of lists:
- Adding of new elements to the list
- Removing elements from the list
- Reading elements from the list
- Replacing elements from the list
• Lists are created mainly with the aid of the class `java.util.Vector`.

• The class `Vector` creates a dynamic list and can contain any JAVA-object.
  (Dynamic list means, that the list has a variable length during the runtime of the program)

• User can add, read and remove elements at any position.

• Every element in a vector can be accessed by its index (the actual position of the element in the list).

• The size of the list is available at any time.

• A set of useful method are also available, for example, building a sub-list, checking the appearance of an element in the list, transforming the list to an `Array` etc.
Lists – the most important methods of the class Vector

- **void addElement(Object o)**: Adds the specified component to the end of this list.

- **boolean contains(Object o)**: Tests if the specified object is a component in this list.

- **Object get(int index)**: Returns the element at the specified position to its caller.

- **Enumeration elements()**: Returns an enumeration of the components of this list.

- **int indexOf(Object o)**: Searches for the first occurrence of the given object.

- **void insertElementAt(Object o, int index)**: Inserts the specified object as a component of this list at the specified position (index).

- **void remove(int index)**: Removes the first occurrence of the specified element in this list.
// create elements of the list
String color1 = new String(“red”);
String color2 = new String(“yellow”);
String color3 = new String(“blue”);

// create Vector
Vector colorlist = new Vector();

// add element to the end of the list
colorlist.addElement(color1);
colorlist.addElement(color2);
colorlist.addElement(color3);

// get the index of an element
int index = colorlist.indexOf(color2);

// remove an element at the specified position
colorlist.removeElementAt(index);

// insert an element at the beginning of the list
colorlist.insertElementAt(color2, 0);

List: red
List: red, yellow
List: red, yellow, blue

index = 1
List: red, blue
List: yellow, red, blue
### With lists

```java
// create a list element
Circle circle1 = new Circle(1.0);
Circle circle2 = new Circle(2.0);
Circle circle3 = new Circle(3.0);

// create a vector
Vector liste = new Vector();

// add an element at the end of the list
liste.addElement(circle1);
liste.addElement(circle2);
liste.addElement(circle3);

// determine the index of an element
int index = liste.indexOf(circle2);

// remove the element at the given index
liste.removeElementAt(index);

// insert an element at the beginning of the list
liste.insertElementAt(circle2, 0);
```

### With arrays

```java
Circle[] liste = new Circle[3];

// insert an element at the beginning of the list
liste[0] = circle1;

// determine the index of an element
int index = -1;
for (int i = 0; i < 3; i++) {
    if (liste[i] == circle2)
        index = i;
}

// delete an element from the list
for (int i = index; i < 2; i++) {
    liste[index] = liste[index + 1];
}
liste[2] = null;

// insert at the beginning (list big enough?)
for (int i = 2; i > 0; i--)
    liste[i] = liste[i - 1];
liste[0] = circle2;
```
Sequential Enumerating of a Linear Data Structure in Java

Enumerating interface **Enumeration**:

- Defined in JAVA-package „java.util“.
- Interface defines methods by which objects in a collection of objects can be enumerated (obtain one at a time).
- Realising the abstract pattern: "enumerate all elements"

**Example**: print all elements of object “list”

Vector aVector;
...

```java
Enumeration e = aVector.elements();
while ( e.hasMoreElements()) {
    System.out.println(e.nextElement());
}
```

Vector entries:
Rose, Clove, Crocus, Tulip

Output:
Rose
Clove
Crocus
Tulip
Outline of Exercise 9 and 10

Content of Exercise 9 and 10

1. Threads
2. Packages
3. Collections. Generic Collections
4. Data Structures
   4.1. Lists
   4.2. ArrayList
   4.3. Stacks and Queues
   4.4. Hash-Tables
   4.5. Tree Structures
   4.6. Net-like Data Structures
   4.7. Objects
5. Graphical User Interface
6. UML
The class ArrayList can be found in `java.util.ArrayList`.
Difference between Vector and ArrayList: the class Vector can be synchronized (multithread-safe), the class ArrayList not.
The APIs of the classes are very similar.

Example of an operation on the ArrayList:

```java
// create the ArrayList
ArrayList e = new ArrayList();

// create and add elements
Circle c1 = new Circle(1.0);
Circle c2 = new Circle(3.0);
e.add(c1);
e.add(c2);
```
Type-casting

Object[] elemente = e.toArray();

for (int i=0; i < elemente.length; i++) {
    Circle k = (Circle)elemente[i];
    k.setRadius(2.0);
    ...
}

The method „Cast“ can be called after setRadius()
Creating an Array from an ArrayList (2)

- Read the elements as array from the object and perform type-casting:
  - Overload the method `toArray()`: `toArray(Object[])`
  - Deliver an (empty) array as casting type

```java
Circle[] t = new Circle[1];
Circle[] allElements = (Circle[])e.toArray(t);
for (int i=0; i < allElements.length; i++) {
    allElements[i].setRadius(2.0);
}
```

Create array of desired type and pass it to the method `toArray()`

Type-casting
Outline of Exercise 9 and 10

Content of Exercise 9 and 10

1. Threads
2. Packages
3. Collections. Generic Collections
4. Data Structures
   4.1. Lists
   4.2. ArrayList
   4.3. Stacks and Queues
   4.4. Hash-Tables
   4.5. Tree Structures
   4.6. Net-like Data Structures
   4.7. Objects
5. Graphical User Interface
6. UML
- A stack is realised by the class `java.util.Stack`

- A stack is a linear list with the restriction that only the element at the top can be accessed (Last-In-First-Out: LIFO):

- Inserting and removing an element are often called **Push** and **Pop** respectively.
Stack - important methods

- **boolean empty()**: Tests if this stack is empty.
- **Object peek()**: Returns the object at the top of this stack without removing it from the stack.
- **Object pop()**: Removes the object at the top of this stack and returns that object as the value of this function.
- **void push(Object item)**: Pushes an item onto the top of this stack.
- **int search(Object o)**: Returns the position where an object is on this stack.
Queues

- Queues are realised by the class `java.util.LinkedList`
- Queues are linear list with the restriction that objects can only be added at the end and only be removed at the beginning of the list (First-In-First-Out: FIFO):
- JAVA does not provide for a queue class but the class `java.util.LinkedList` can serve as substitute.
Queues - important methods

- `void addFirst(Object o)`: Inserts the given element at the beginning of this list.
- `Object getLast()`: Returns the last element in this list.
- `Object removeLast()`: Removes and returns the last element from this list.
- `int size()`: Returns the number of elements in this list.
- `int indexOf(Object o)`: Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
Outline of Exercise 9 and 10

Content of Exercise 9 and 10

1. Threads
2. Packages
3. Collections. Generic Collections
4. Data Structures
   4.1. Lists
   4.2. ArrayList
   4.3. Stacks and Queues
   4.4. Hash-Tables
   4.5. Tree Structures
   4.6. Net-like Data Structures
   4.7. Objects
5. Graphical User Interface
6. UML
• A Hashtable is realised by the class `java.util.Hashtable`

• Entries in hashtables are key-value-pairs.

• Each value-object in Hashtable is addressed with a key-object.

• Any JAVA-Object can be used as a key and as a value.

• Instead of searching for an element with the index, the element in Hashtable can be directly accessed through the key.
Hashtables

- A special type of linear fields, in which an unique key has to be addressed to each element of the linear field.
- With the Hash-function the key is converted into an address of the table. The element is stored at this address.
- The access on a data element is direct by the key.
Hashtable - Example

<table>
<thead>
<tr>
<th>Name of person</th>
<th>eMail-address</th>
</tr>
</thead>
<tbody>
<tr>
<td>„Fred“</td>
<td><a href="mailto:fred@bedrock.com">fred@bedrock.com</a></td>
</tr>
<tr>
<td>„Barney“</td>
<td><a href="mailto:barney@bedrock.com">barney@bedrock.com</a></td>
</tr>
<tr>
<td>„Wilma“</td>
<td><a href="mailto:wilma@bedrock.com">wilma@bedrock.com</a></td>
</tr>
</tbody>
</table>

// create key objects
String[] name = new String[3];
name[0] = new String(“Fred”);
name[1] = new String(“Barney”);
name[2] = new String(“Wilma”);

// create list elements
String[] email = new String[3];
email[0] = new String(“fred@bedrock.com”);
email[1] = new String(“barney@bedrock.com”);
email[2] = new String(“wilma@bedrock.com”);

// create Hashtable
Hashtable hash = new Hashtable();

// fill the Hashtable
for ( int i = 0; i < 3; i++ ) {
    hash.put(name[i], email[i]);
}

// print out Wilmas eMail-address
System.out.println(hash.get(“Wilma”));
HashMap in Java: Examples of operations

```java
import java.util.HashMap;
...
HashMap h = new HashMap();

// create and add elements
Circle k = new Circle(1.0);
Cylinder z = new Cylinder(k,3.0);
h.put(k,z);

if (h.containsKey(k)) {...}
if (h.containsValue(z)) {...}

Cylinder[] d = new Cylinder[1];
Cylinder[] allCyl;
allCyl = (Cylinder[])(h.values()).toArray(d);
```

- Import HashMap
- Create HashMap
- Add elements to the HashMap
- Verify, if Key oder Value are in the HashMap
- Read elements as Array
Outline of Exercise 9 and 10

Content of Exercise 9 and 10

1. Threads
2. Packages
3. Collections. Generic Collections
4. Data Structures
   4.1. Lists
   4.2. ArrayList
   4.3. Stacks and Queues
   4.4. Hash-Tables
   4.5. Tree Structures
   4.6. Net-like Data Structures
   4.7. Objects
5. Graphical User Interface
6. UML
Tree Structures (1)

- **Tree (class javax.swing.tree.DefaultTreeModel):**
  - Each instance of this class administrates a tree structure
  - Methods to add, remove, get a node of the tree structure are supplied
  - It is suitable to administrate the tree structure with any degree
  - Additional methods, such as to get the path from a specified node to the root is supplied

- **Node (class javax.swing.tree.DefaultMutableTreeNode):**
  - Each instance of this class represents a node of a tree structure.
  - Each node can be associated to a UserObject.
  - Any JAVA-Object can be used as an UserObject.
  - In order to administrate any data object in a tree structure, the data object should be referenced by a node at first, the node will then be added in the tree.
2. Tree Structures

Instance of class DefaultTreeModel

Instance of class DefaultMutableTreeNode

Instance of class DefaultMutableTreeNode

Instance of class DefaultMutableTreeNode

object

object
public class Book {
    private DefaultTreeModel tree;
    private DefaultMutableTreeNode root;

    public Book (String titel) {
        root = new DefaultMutableTreeNode(titel);
        tree = new DefaultTreeModel(root);
    }

    public void addChapter (File newchapter) {
        DefaultMutableTreeNode node = new DefaultMutableTreeNode(newchapter);
        root.add(node);
    }

    public void addParagraph(File newparagraph, File ofChapter) {
        DefaultMutableTreeNode node = new DefaultMutableTreeNode(newparagraph);
        DefaultMutableTreeNode chapterNode = findNode(ofChapter);
        chapterNode.add(node);
    }

    private DefaultMutableTreeNode findNode(Object userObject) {
        ...
        ...
    }
}
public class Book {
    ...

    private DefaultMutableTreeNode findNode(Object userObject) {
        DefaultMutableTreeNode ret;
        Enumeration en = root.breadthFirstEnumeration();
        while ( en.hasMoreElements() ) {
            DefaultMutableTreeNode aktnode = (DefaultMutableTreeNode)en.nextElement();
            if ( aktnode.getUserObject() == userObject)
                ret = aktnode;
        }
        return ret;
    }

    // look-out: graphic representation of the tree structure
    public JTree getJTree() {return new JTree(tree);}
}
```java
public class Start {
    public static void main(String[] args) {
        ...
        Book mybook = new Book("Buch1");
        mybook.addChapter(chapter1);
        mybook.addParagraph(paragraph1, chapter1);
        mybook.addParagraph(paragraph2, chapter1);
        mybook.addChapter(chapter2);
        ...
        // it is supposed, the "File"-Objects
        // for chapter and paragraph were previously defined.
    }
}
```
Outline of Exercise 9 and 10

Content of Exercise 9 and 10

1. Threads
2. Packages
3. Collections. Generic Collections
4. Data Structures
   4.1. Lists
   4.2. ArrayList
   4.3. Stacks and Queues
   4.4. Hash-Tables
   4.5. Tree Structures
   4.6. Net-like Data Structures
   4.7. Objects
5. Graphical User Interface
6. UML
Net-like Data Structures

• **Net-like data structure**
  - An element can have an arbitrary number of successors (like the tree structure).
  - It can also have an arbitrary number of predecessors in contrast to tree structure where only one predecessor is allowed.

There is no class in JAVA available to map a net-like data structure. Therefore own implementations of suitable classes are necessary.
Net-like Data Structures: Example: the Tetrahedron

Volume $V_1$: tetrahedron

Data structure of the tetrahedron

- Points: $P_1, P_2, P_3, P_4$
- Faces: $F_1, F_2, F_3, F_4$
- Edges: $e_1, e_2, e_3, e_4, e_5, e_6$
// Implementation of a point
// with x, y and z-coordinate

public class Point {
    public float x;
    public float y;
    public float z;

    // constructor
    public Point(float x,
                 float y,
                 float z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}

// Implementation of an edge
// consisting of start point
// p1 and end point p2

public class Edge {
    public Point p1;
    public Point p2;

    // constructor
    public Edge(Point p1,
                Point p2) {
        this.p1 = p1;
        this.p2 = p2;
    }
}
Example – the Tetrahedron - Implementation (2)

Triangle area:

// Implementation of a triangle area
// it consists of
// three edges

public class TriangleArea {
    public Edge e1;
    public Edge e2;
    public Edge e3;

    // constructor
    public TriangleArea(Edge e1,
                        Edge e2,
                        Edge e3) {
        this.e1 = e1;
        this.e2 = e2;
        this.e3 = e3;
    }
}

Tetrahedron:

// Implementation of a tetrahedron
// limited by tree triangle areas

public class Tetrahedron {
    public TriangleArea f1;
    public TriangleArea f2;
    public TriangleArea f3;
    public TriangleArea f4;

    // constructor
    public Tetrahedron(TriangleArea f1,
                        TriangleArea f2,
                        TriangleArea f3,
                        TriangleArea f4) {
        this.f1 = f1;
        this.f2 = f2;
        this.f3 = f3;
        this.f4 = f4;
    }
}
Instance of a tetrahedron:

// Definition of the points
Point p1 = new Point (0.0, 0.0, 0.0);
Point p2 = new Point (2.0, 0.0, 0.0);
Point p3 = new Point (1.0, 2.0, 0.0);
Point p4 = new Point (1.0, 1.0, 2.0);

// Definition of the edges
Edge e1 = new Edge (p1, p2);
Edge e2 = new Edge (p2, p3);
Edge e3 = new Edge (p3, p4);
Edge e4 = new Edge (p1, p4);
Edge e5 = new Edge (p2, p4);
Edge e6 = new Edge (p1, p3);

// Definition of the faces
TriangleArea f1 = new TriangleArea (e1, e2, e6);
TriangleArea f2 = new TriangleArea (e1, e4, e5);
TriangleArea f3 = new TriangleArea (e2, e3, e5);
TriangleArea f4 = new TriangleArea (e3, e4, e6);

// Definition of the volume
Tetrahedron V1 = new Tetrahedron (f1, f2, f3, f4);
Outline of Exercise 9 and 10

Content of Exercise 9 and 10

1. Threads
2. Packages
3. Collections. Generic Collections
4. Data Structures
   4.1. Lists
   4.2. ArrayList
   4.3. Stacks and Queues
   4.4. Hash-Tables
   4.5. Tree Structures
   4.6. Net-like Data Structures
   4.7. Objects
5. Graphical User Interface
6. UML
Comparing Objects

Question: when are 2 objects “equal” to each other?

• Object identity: ==
  - If the all variables of the objects are equal (both objects contain these variables and the values are equal to each other), comparing the objects will give a “false” result.
  - == is true, if an object is compared with itself.
  - Example: 2 variables reference the same object.

• Equality due to the same state: equals (Object o)
  - Checks, if 2 objects can be considered to be equal due to the value of their attributes.
  - The method equals is defined in the class Object.
  - Each class must define (overwrite) the method equals.
public class Circle {
    private int x = 0;
    private int y = 0;
    private int r = 0;
}

public Circle c = new Circle();

public class Sphere {
    private int x = 0;
    private int y = 0;
    private int z = 0;
    private int r = 0;
    public Circle t = new Circle();
}

public Sphere s = new Sphere();
public class Circle {
    private int x = 0;
    private int y = 0;
    private int r = 0;
    public Circle(int x, int y, int r) {
        this.x = x;
        this.y = y;
        this.r = r;
    }

    public Circle c = new Circle(2,3,4);
    ...
}
public class Circle {
    private int x = 0;
    private int y = 0;
    public int r = 0;
    public Circle(int x, int y, int r) {
        this.x = x;
        this.y = y;
        this.r = r;
    }
}

public Circle c = new Circle(2, 3, 4);
public Circle k = c;
k.r = 7;

// c.r = 7 and k.r = 7
Example

```java
public boolean equals (Object o) {
    if (o == null) return false;
    if (this.getClass() != o.getClass()) return false;
    Circle c = (Circle) o;
    if (c.r != this.r) return false;
    if (c.x != this.x) return false;
    if (c.y != this.y) return false;
    return true;
}
```

Circle c1 = new Circle(1.0, 0.0, 0.0);
Circle c2 = new Circle(1.0, 0.0, 0.0);
Circle c3 = new Circle(1.0, 1.0, 1.0);
Circle c4 = c1;

In this example:
- c1.equals(c2) is true
- c1.equals(c3) is false
- c1.equals(c4) is true

4. Objects
Outline of Exercise 9 and 10

Content of Exercise 9 and 10

1. Threads
2. Packages
3. Collections. Generic Collections
4. Data Structures
5. Graphical User Interface
6. UML
Motivation

http://www.fotocommunity.de/
In the Beginning...

- Xerox Palo Alto Research Center
  - Prototyp, 1971

- Apple Lisa
  - 1983/1984
Java-AWT (Abstract Window Toolkit)
All components (except for menus) are subclasses of the class `Component`. From `Component`, they inherit a wide range of functionalities:

- **Basic displaying support:**

  `Component` provides methods `paint`, `update` and others that allow an individually adapted component to display itself on the screen.

- **Event handling:**

  `Component` defines a range of methods to handle certain types of events (the user’s activity on a component)

- **Influence on displaying mode:**

  `Component` provides methods to specify or determine current typeface (font) and colour.
Layout Manager in Java-AWT

- **BorderLayout**
  - Provides five areas for reception of components

- **FlowLayout**
  - Orders the elements from left to right opening a new line if necessary.

- **CardLayout**
  - Allows for only one component of a container to be visible – like in a stack of playing cards. With the call of certain statements it’s possible to switch between components.
• **GridLayout**

  gives all the components equal dimensions and shows them in the given number of rows and columns.

• **GridBagLayout**

  Components are shown in a grid of rows and columns, certain components may occupy several rows/columns.
• **Canvas**

Canvas is a rectangular window with basic functions for drawing areas or for the processing of events, e.g. a displaying area for pictures or graphics.

• **Button**

A button is a simple form for interaction creating an action event as soon as the user triggers it.

• **Choice**

A menu like list of options that are accessed by choice of the user. Choice is used if a set of alternatives shall be shown in a limited area.
• **Checkbox**

  Checkbox provides two states for each element: „on“ and „off“. If a user chooses a box, it changes its current state and triggers an action event.

• **List**

  List provides a moveable area containing different text entries, one per row. The user chooses an entry by clicking once and triggers an action event by double-clicking.
• **Label**
  
  Label offers the possibility to put (unselectable) text into the GUI.

• **TextArea and TextField**
  
  - TextField is used for the handling of single-line text.
  
  - TextArea is used for the handling of multi-line text.

• **Scrollbar**
  
  Scrollbars are used to display a part of a bigger area that is -as a whole- too big for the given displaying space.
Menus in Java-AWT

- Menubar
- MenuItem
- Submenu
- Popup-Menu
- ClickboxMenuItem
- Separating line
Java Swing, the new Toolkit

• In 1996, Java-AWT was offered with the Java Development Toolkit (JDK) in version 1.0.
  Week point of the AWT:
  - Uses platform dependent library
  - Programs are displayed differently on different platforms

• In late 1997 a new product (Java Swing) was offered as a toolkit for creating user interfaces with JDK Version 1.2.
  Important differences between AWT and Swing:
  - Swing never uses platform dependent code, appears exactly the same on all platforms
  - Swing-components fully contain the AWT-components, plus a set of higher-level-components.
  - Swing increases the quality and makes programming easier
Classical Components in Java Swing

- Dialog
- Choice
- Button
- List
- Menu
- Label
- Textarea and Textfield
Additional Components in Java Swing

- Progress Bar
- Table
- Tabbed Pane
- Tree
- Slider
Outline of Exercise 9 and 10

Content of Exercise 9 and 10

1. Threads
2. Packages
3. Collections. Generic Collections
4. Data Structures
5. Graphical User Interface
6. UML
UML-diagram types

- Use-Case diagrams
- Class and instance diagrams
- Sequence diagrams
- State diagrams
- Activity diagrams
<table>
<thead>
<tr>
<th>Term</th>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Boundary</td>
<td><img src="image" alt="SystemBoundary" /></td>
<td>The <strong>System boundary</strong> is the border between the system about to be developed and the environment.</td>
</tr>
<tr>
<td>Use Case</td>
<td><img src="image" alt="UseCase" /></td>
<td>A <strong>Use Case</strong> is a group of functions that come from a class or components and are set at the user's disposal.</td>
</tr>
<tr>
<td>Actor</td>
<td><img src="image" alt="Actor" /></td>
<td><strong>Actors</strong> represent everything that lies outside of the system and communicates and interacts with the system. Actors can be people, users, external hardware, or software systems.</td>
</tr>
<tr>
<td>Communication association link</td>
<td><img src="image" alt="Communication association link" /></td>
<td>A <strong>Communication Link</strong> represents the interaction between an instance of the case and an instance of the actor.</td>
</tr>
</tbody>
</table>
Use-Case Diagrams – Example

- Describes the service that the "ordering system" offers,

- especially the requirements of the customer for the system.
Class diagrams (1)

- **Class:**
  - Name of the class
  - Attribute:
    - `<Term>`<Attributename>:<Type of value>
  - Methods:
    - `<Term>`<Methodname >: <Type the return value>

- **Association:**
  - Labelling the types (with a rhombus):
    - Association ("no rhombus")
    - Aggregation ("empty rhombus")
    - Composition ("filled rhombus")
  - Assigning the cardinality (c₁, c₂):
    - How many objects of the class are involved with the association?
      - 0..1, 1, 0..* or 1..*

Term indicator:
- + = public
- - = private
<table>
<thead>
<tr>
<th>Term</th>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><img src="image" alt="Class Diagram" /></td>
<td>Groups a bunch of objects with similar behavior</td>
</tr>
<tr>
<td>Interface</td>
<td><img src="image" alt="Interface Diagram" /></td>
<td>Defines methods that must be implemented in a class</td>
</tr>
<tr>
<td>Association</td>
<td><img src="image" alt="Association Diagram" /></td>
<td>Relationship between classes. The associations can be qualified through its cardinality. (c_1, c_2) can be 0, 1 or more than 1 (0, 1, 0..1, 0..<em>, 1..</em>)</td>
</tr>
<tr>
<td>Association (directed)</td>
<td><img src="image" alt="Directed Association Diagram" /></td>
<td>Objects of the class Class2 can not be navigated through this association to an object of the class Class1.</td>
</tr>
</tbody>
</table>
### Class diagrams (3)

<table>
<thead>
<tr>
<th>Term</th>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregation</td>
<td><img src="image" alt="Aggregation Diagram" /></td>
<td>Special association that fulfils a “is composed of” relationship</td>
</tr>
<tr>
<td>Composition</td>
<td><img src="image" alt="Composition Diagram" /></td>
<td>Strengthening of the “Aggregation”: objects in Class1 can not exist without objects of class Class2.</td>
</tr>
<tr>
<td>Inheritance</td>
<td><img src="image" alt="Inheritance Diagram" /></td>
<td>The inheritance describes a “is one” relationship between one or more classes (ex: a car is a vehicle)</td>
</tr>
<tr>
<td>Term</td>
<td>Symbol</td>
<td>Definition</td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Object</td>
<td><img src="image" alt="Object Symbol" /></td>
<td>An object becomes part of the scenario through the reception of messages (calling up a method)</td>
</tr>
<tr>
<td>Message</td>
<td><img src="image" alt="Message Symbol" /></td>
<td>Calling up a method with continuous numbering and surrendering of the parameters.</td>
</tr>
<tr>
<td>Life line of a process step</td>
<td><img src="image" alt="Life Line Symbol" /></td>
<td>Represents the execution of the process-steps</td>
</tr>
<tr>
<td>Self advice</td>
<td><img src="image" alt="Self Advice Symbol" /></td>
<td>Calling up the same object</td>
</tr>
</tbody>
</table>
• Continual numbering

• Nesting of the call through a clear numbering, for ex: 1.1.3

• Life line of the process steps clarify the adaptation of the methods

• Dashed lines signal the end of execution of the method and ends the life line of the process steps.

**Object1**

**initial**
Circle

**middlePoint**
Point

1: GetXCoordinateOfMP():double

1.1: GetX():double
### State diagrams

<table>
<thead>
<tr>
<th>Term</th>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start state</td>
<td><img src="image" alt="Start State Symbol" /></td>
<td>Start point</td>
</tr>
<tr>
<td>End state</td>
<td><img src="image" alt="End State Symbol" /></td>
<td>End point</td>
</tr>
<tr>
<td>State</td>
<td><img src="image" alt="State Symbol" /></td>
<td>state (of an object)</td>
</tr>
<tr>
<td>State transition</td>
<td><img src="image" alt="State Transition Diagram" /></td>
<td>The activating events will be written on the shaft.</td>
</tr>
<tr>
<td>Conditioned state transition</td>
<td><img src="image" alt="Conditioned State Transition Diagram" /></td>
<td>Prerequisite of the condition transition will be written with either [] or {} (not uniformly though!)</td>
</tr>
<tr>
<td>Term</td>
<td>Symbol</td>
<td>Definition</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Activity</td>
<td><img src="Activity1" alt="Activity" /></td>
<td>Represents activities that will be performed by the system or by the actors.</td>
</tr>
<tr>
<td>Start state</td>
<td>![Start state](Start state)</td>
<td>Entrance point</td>
</tr>
<tr>
<td>End state</td>
<td>![End state](End state)</td>
<td>End point</td>
</tr>
<tr>
<td>Decision</td>
<td><img src="Decision" alt="Decision" /></td>
<td>Represents a point in an activity diagram, where a workflow (based on definite conditions) is ramified on multiple alternatives.</td>
</tr>
<tr>
<td>Split/Join flow</td>
<td>![Split/Join flow](Split/Join flow)</td>
<td>This join supports the synchronisation - each workflow (thread) waits with the execution until all other workflows have achieved this synchronisation point.</td>
</tr>
<tr>
<td>Term</td>
<td>Symbol</td>
<td>Definition</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Transition link</strong></td>
<td><img src="image" alt="Transition link" /></td>
<td>Signals the transition from an activity condition into the next</td>
</tr>
<tr>
<td><strong>Swim lane</strong></td>
<td><img src="image" alt="Swim lane diagram" /></td>
<td>A swim line shows the action and its activity, which is carried out by one item (object, class)</td>
</tr>
</tbody>
</table>
Activity diagrams - example

- Conditional workflow
  - Forking out of the workflow
  - Synchronization of the workflow

- SwimLane
  - Activity 1
  - Activity 2

- Activity 1
  - Fill Order
  - Overnight Delivery
  - Receive Order
  - Receive Payment

- Activity 2
  - Send Invoice
  - Regular Delivery
  - Close Order