## 5 Classes and Objects

- Introduction
- Data encapsulation
- Example: Rational Numbers
- Wrapper classes
- Standard datatypes Queue, Stack, List
- Inheritance
- Chains of constructors
- Referencing superclasses and subclasses
- Override annotation
- Abstract methods & abstract classes
- Interfaces
- Inner Classes

It is possible to define classes *within other classes*:

- *Nested Static Classes*
  - Interfaces and static classes as elements of other classes
  - used to structure the code
- *Member Classes*
  - non-static classes as elements of other classes
- *Local Classes*
  - classes locally defined within methods
- *Anonymous Classes*
  - local classes without an explicit name

**Nested Static Classes** are used to structure the code; definitions of interfaces and classes are embedded in another class.

Example:

```
1  class TopLevel {
2
3   interface NestedIF {
4      void method1 ();
5      int method2 ();
6   }
7
8   static class NestedSC {
9       ...
10  }
11 }
```

```
1  class User1 implements
2                    TopLevel.NestedIF {
3   void method1 () {...}
4   int method2 () {...}
5   TopLevel.NestedSC foo
6              = new TopLevel.NestedSC();
7     ...
8  }
```

```
1  import TopLevel.NestedSC;
2  class User2 {
3   NestedSC foo = new NestedSC();
4   ...
5  }
```

Example: '**`interface`** `Enumeration`' – implementation via inner classes:

- `java.util.Enumeration` defines a simple interface with two core methods for accessing collections like lists, queues, etc.:

```
1 public boolean hasMoreElements();
2
3 public Object nextElement () throws NoSuchElementException;
```

- We will implement this interface in different ways, namely as:
  - class (normal 'top-level class')
  - member class
  - local class
  - anonymous class
- We will later see other typical examples for the use of inner classes when we discuss events.

Basis for examples on `inner classes`: class `Elem`
(exactly like in the previous examples `Queue` and `Stack`!)

used to create example class `SimpleList` similar to `Queue`
i.e., simple linkage, unsorted, but references for `start`, `end`:

```
1  import java.util.Enumeration;
2  public class SimpleList {
3    SimpleList () {}
4    private Element start, end;
5
6    public static int getKey(Object obj) { // sorting key
7        return  (Integer)obj;    // only for Integer objects
8    }
9
10   public Element getStart () { return start;}
11
12   public void addToEnd (Elem newElement) {
13     if (start == null) { start= newElement; end= newElement; }
14     else { end.setNext (newElement);  end= newElement; }
15   }
16 ...
```

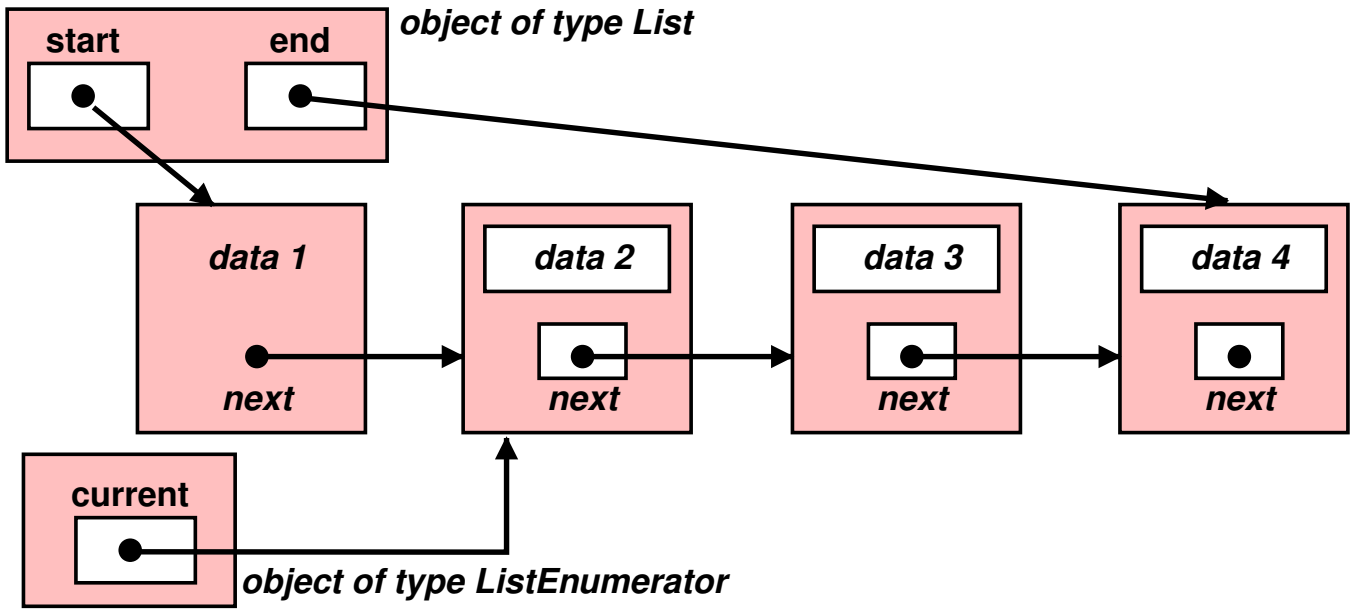
```
1   public void delete (Elem toDelete) {
2     if (toDelete == start) start = start.getNext();
3     else { Elem current = start;
4        while (current.getNext() != null)
5           if (current.getNext() == toDelete)
6              current.setNext(current.getNext().getNext());
7           else current = current.getNext();
8     }
9   }
10  @Override public String toString () {
11    Element current = start; String output = "";
12    if (start != null)
13       while (current != null) {
14          output += current.toString() + "␣";
15          current = current.getNext(); }
16    return output;
17  }
18
19 // new: Enumerator:
20   public Enumeration enumerate() {
21     return new ListEnumerator (this);
22   }
23 }
```

(what happens at `delete`, if `toDelete == start`?
Correct the method!)

# Enumerator as normal ‘top-level class’:

```
1  import java.util.Enumeration;
2  import java.util.NoSuchElementException;
3
4  public class ListEnumerator implements Enumeration {
5
6     private Elem current;
7
8     public ListEnumerator (SimpleList li) {
9        current = li.getStart(); }
10
11    @Override public boolean hasMoreElements () {
12       return (current != null); }
13
14    @Override public Object nextElement () {
15       if (current == null)
16          throw new NoSuchElementException("List");
17       Object value = current;
18       current = current.getNext();
19       return value;
20    }
21 }
```

(A list can have multiple enumerators, thus implementation in separate class!)


start
end
object of type List
data 1
next
data 2
next
data 3
next
data 4
next
current
object of type ListEnumerator

# Test example `K5B15E_TopLevelClass` for Enumerator as 'top-level class'

```
1  import java.util.*;
2  public class ListTest {
3    public static void main(String[] args) {
4      Scanner sc = new Scanner(System.in);
5
6      SimpleList liste = new SimpleList();
7      Enumeration listEnum;
8
9      while ( true ) {
10       System.out.print("Input [key|-key|0]: ");
11       String str = sc.nextLine();
12       if (str.equals("0")) {
13         return;
14       } else {
15         int key = Integer.parseInt(str);
16         if (key > 0) {
17           liste.addToEnd(new Elem(new Integer(key)));
18           System.out.println("AddtoEnd: " + key);
19           System.out.println("List:     " + liste);
20         } else {
21 ...
```

```
1  ...
2              key = - key;
3              listEnum = liste.enumerate();
4              while (listEnum.hasMoreElements()) {
5                Elem current = (Elem) listEnum.nextElement();
6                if ( SimpleList.getKey(current.getObject()) == key)
7                        liste.delete(current);
8              }
9              System.out.println("Delete:␣" + key);
10             System.out.println("List:␣␣␣" + liste);
11           }
12         }
13       }
14     }
15 }
```

**Member Classes**:

- Member Classes are defined at the level of method and variable declarations, but they do not have the modifier '`static`'.
- Every instance of a member class is associated with an instance of the surrounding class.
- Methods of a member class can access elements of the member class and elements (incl. private) of the surrounding class.

Member classes can be used instead of nested top-level classes,

- if the inner class needs to access instance elements of the outer class.
- if every instance of the inner class needs to refer to an instance of the outer class.

# Enumerator as 'member class' in class `SimpleList`:

```
1  public class SimpleList {
2    SimpleList () {}
3    ...
4    @Override public String toString () { ... }
5    public Enumeration enumerate() {return new ListEnumerator ();}
6  //begin of the member class:
7    private class ListEnumerator implements Enumeration {
8      private Elem current;
9      public ListEnumerator () { current = start; }
10     @Override public boolean hasMoreElements(){
11        return (current != null);
12     }
13     @Override public Object nextElement () {
14        if (current == null)
15           throw new NoSuchElementException("List");
16        Object value = current;  current = current.getNext();
17        return value;
18     }
19   }
20 //end of the member class
21 }
```

Rest in `K5B16_MemberClass` exactly as in `K5B15E_TopLevelClass`!

## Local Classes

- Local Classes are declared locally within a code block (method, constructor, initialization block).
- Visible and useable only in the block containing the declaration. access to local variables only possible if they are ‘final’.

# Enumerator as ‘local class’ in class `SimpleList`:

```
1  public class SimpleList {
2    SimpleList () {}
3    ...
4    @Override public String toString () { ... }
5    public Enumeration enumerate() {
6     //begin of the 'local class'
7     class ListEnumerator implements Enumeration {
8       private Elem current;
9       public ListEnumerator () { current = start; }
10      @Override public boolean hasMoreElements(){
11        return (current != null);
12      }
13      @Override public Object nextElement () {
14        if (current == null)
15          throw new NoSuchElementException("List");
16        Object value = current;
17        current = current.getNext();
18        return value; }
19    } //end of the `local class'
20     return new ListEnumerator ();
21   } //end of the method 'enumerate'
22 }
```

Rest and `K5B18E_ListTest_localclass` exactly as before!

**Anonymous Classes**:

- Anonymous classes are like local classes *without name*.
- Combination of the two steps “definition of a local class” and “instantiation of a local class”.
- Anonymous classes do not have a constructor.
- Only one instance of the class is created.
- The created object is used immediately, e.g., in an expression.

# Enumerator as ‘anonymous class’ in class `SimpleList`:

```
1  public class SimpleList {
2    SimpleList () {}
3    ...
4    public String toString () { ... }
5
6    public Enumeration enumerate() {
7     return new Enumeration () {
8     //begin of the 'anonymous class'
9      private Elem current = start;
10
11       @Override public boolean hasMoreElements(){
12          return (current != null);
13       }
14       @Override public Object nextElement () {
15          if (current == null)
16             throw new NoSuchElementException("List");
17          Object value = current;
18          current = current.getNext();
19          return value; }
20      //end of the `anonymous class'
21      };
22   } //end of the method 'enumerate'
23 }
```

Rest and `K5B19E_ListTest_anonymousclass` exactly as before!

**Implementation of ‘inner classes’:**

- Inner classes are a concept at the level of the compiler.
- The Java Virtual Machine does not use inner classes.
- For member classes and local classes, the compiler creates for the previous example the `.class` files `SimpleList.class` and `SimpleList$1ListEnumerator.class`
- For the anonymous class, `SimpleList.class` and `SimpleList$1.class` are generated.
- For the JVM, there is no difference to ‘normal’ top-level classes .
- Inner classes often allow for a better handling of specific concepts like enumerators, event handlers, etc.

## **Packages**:

- Packages are collections of associated classes.
- Goal: topic-driven structuring of class libraries.
- Java comes with a large library of predefined classes, subdivided into more than 200 packages, `http://docs.oracle.com/javase/8/docs/api/overview-summary.html` for example
  - `java.lang` (core language classes, always imported)
  - `java.util` (general services)
  - `java.io` (basic services for input and output)
  - `java.math` (includes computation with large numbers)
- Every newly developed class can be assigned to a user-defined package, e.g.:

```
1  package myPackage;
2  import java.util.*
3  class xyz { ... }
```

- Packages need to be 'imported'
- Classes without an assigned package are members of the 'default package' and do not need to be imported.

Packages need to be stored in a directory of the same name:

```
1 package K5B19E_Packages.Geometry;
2 public class Rectangle {
3     int wi, he;
4     public Rectangle (int w, int h) { wi =  w; he = h; }
5     public int area () { return wi * he;  }
6 }
```

with application:

```
1 package K5B19E_Packages;
2 import K5B19E_Packages.Geometry.Rectangle;
3 class RectangleTestA {
4    public static void main(String[] args) {
5       Rectangle r = new Rectangle (4,5);
6       System.out.println( "Area: " + r.area());
7 }  }
```

or (without import)

```
1 package K5B19E_Packages;
2 class RectangleTestB {
3    public static void main(String[] args) {
4       K5B19E_Packages.Geometry.Rectangle r
5         = new K5B19E_Packages.Geometry.Rectangle (4,5);
6       System.out.println( "Area: " + r.area());
7 }  }
```