Lambda expressions in Java

Again our example application

```
public interface ActionListener extends EventListener {
   public void actionPerformed(ActionEvent e);
}

public class AppWithAnonymousClass {
    private void run() {
        JFrame frame=new JFrame("Hello");
        [...]

        JButton button=new JButton("Press me!");
        button.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(frame,"Thank you! "+e.getModifiers());
            }
        });
        frame.add(button);

        frame.setVisible(true);
    }

    public static void main(String[] args) {
        [...]
    }
}
```

The ActionEvent object contains information about what has happened. For example, I can see whether the Shift-Key or the Control-Key have been pressed.

What do we notice?

- Let's look again at the code of our ActionListener object:

```
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(frame,"Thank you!"+e.getModifiers());
    }
});
```

- We can notice something interesting:
  - Our ActionListener object is a very stupid object. It only has one method and it doesn't have members.
  - It's only there because we wanted to tell the button to execute this line of code if the button is pressed:

```
JOptionPane.showMessageDialog(frame,"Thank you!"+e.getModifiers());
```

- Is there an easier way?

Lambda Expressions

- Since Java 8, we can write

```
button.addActionListener(new ActionListener() {
     @Override
     public void actionPerformed(ActionEvent e) {
         JOptionPane.showMessageDialog(frame,"Thank you!"+e.getModifiers());
     }
});
```

simply like this:

```
JButton button=new JButton("Press me!");
button.addActionListener(
  (ActionEvent e) -> JOptionPane.showMessageDialog(frame,"Thank you!")
);
```

This is called a lambda expression

- Again, this is syntactic sugar. But the idea behind an ActionListener becomes much clearer by using lambda expressions:

“An ActionListener is a function $f: ActionEvent \rightarrow Void$ that is executed when a button is pressed”

Lambda Expressions

- A Lambda expression consists of two parts:

```
(ActionEvent e) -> JOptionPane.showMessageDialog(frame,"Thank you!")
```

One or more parameters

An expression

- A Lambda expression can also have a result. This lambda expression is equivalent to the function $f(x, y) = x + y$ with $x \in \mathbb{Z}$:

```
(int x, int y) -> x+y
```

- You can also have a block of statements on the right side. In that case, you have to use “return” for the result:

```
(int i) -> { int j=i*2; return j+1; }
```

Functions as objects

- Lambda expressions can be used whenever there is an interface with one abstract method (called a *Functional Interface*)

```
interface MyFunction {
    public double calculate(double d);
}
```

This is a functional interface. It has only one abstract method.

```
public class FunctionExample {
    public static double execute(MyFunction function, double d) {
        return function.calculate(d);
    }

    public static void main(String[] args) {
        MyFunction mySquareRootFunction = (double d) -> Math.sqrt(d);
        double rootOfFive = mySquareRootFunction.calculate(5.0);
        double rootOfSeven = mySquareRootFunction.calculate(7.0);

        MyFunction myIncrementFunction = (d) -> d+1.0;
        double fivePlusOne = execute(myIncrementFunction, 5.0);
    }
}
```

A method that takes a function as parameter

We don’t have to specify the type of the parameter explicitly if it is clear from the interface definition

Package java.util.function

- The JDK has already defined some useful functional interfaces:
  - Interface Function<T,R> for functions of type $T \to R$

    We can rewrite our example

    ```
    MyFunction mySquareRootFunction = (d) -> Math.sqrt(d);
    double rootOfFive = mySquareRootFunction.calculate(5.0);
    ```

    as

    ```
    Function<Double,Double> f = (d) -> Math.sqrt(d);
    double r = mySquareRootFunction.apply(5.0);
    ```

  - Interface BiFunction<T,U,R> for functions of type $T \times U \to R$

    ```
    BiFunction<Integer,Integer,Integer> sum = (i1,i2) -> i1+i2;
    int r = sum.apply(3,5);
    ```

  - Interface Predicate<T> for functions of type $T \to boolean$

    ```
    Predicate<Integer> testZero = (i) -> i==0;
    boolean isZero = testZero.test(5);
    ```

Function composition

- We can combine functions to create new functions
  - Let $f: T \rightarrow R$ and $g: V \rightarrow T$
  - We can define a new function $h: V \rightarrow R$ with $h(x) = f(g(x))$
- The Java interface Function<T,R> has already a compose method that does exactly this:

```
Function<V, R> compose(Function<V,T> g) {
    return (V v) -> apply(g.apply(v));
}
```

- Example:

```
Function<Double, Double> f = (d) -> d/2.5;
Function<Integer, Double> g = (i) -> Math.sqrt(i);
Function<Integer, Double> h = f.compose(g);
double r = h.apply(25);
```

The Comparator Interface (in java.util)

- Interface Comparator<T> for functions $T \times T \rightarrow int$ that

  return an integer $\begin{cases} < 0, & if\ first\ argument < second\ argument \\ 0, & if\ first\ argument = second\ argument \\ > 0, & if\ first\ argument > second\ argument \end{cases}$

- Example (from stackoverflow):

```
Comparator<Duck> byWeight = (d1,d2) -> d1.getWeight() - d2.getWeight();
```

- The `Comparator` interface is used in a lot of Java libraries. For example, there is the static method `sort` from the class java.util.Arrays:

```
Duck[] ducks = new Duck[]{ duck1, duck2, duck3 };
Arrays.sort(ducks, byWeight);
```

- Thanks to the `Comparator` interface it's very easy to change the sort order simply by defining the comparator function differently:

```
Comparator<Duck> byWeight = (d1,d2) -> d2.getWeight() – d1.getWeight();
```

- There is also a similar `sort` method for lists in java.util.Collections

Immutable data structures

Programming without side effects

- Remember that lambda expressions in Java are implemented as anonymous inner classes. They are allowed to access members of the outer class:

```
public class SideEffect {
    int sum=0;

    public void run() {
        Function<Integer,Integer> add=(i) -> { sum++; return i+sum; };

        System.out.println(add.apply(3));
        System.out.println(add.apply(3));
    }
}
```

Side effect

- Howver, this kind of code should be avoided. In mathematics, we expect that a function always gives the same result for the same argument.
- A good function should not have *side effects*. A function should not change existing objects and variables. The code of a function or method is easier to understand if the result <u>only</u> depends on its arguments.

Programming without side effects (2)

- Is it possible to write programs only with code that has no side effects?
- For example, how can we write the method addElement of a class List as a function?

```
class List {
    ...
    void addElement(Element e);
}
```

- Is it possible to add an element to a list without modifying the list???

A list without side effects

- We can implement a list without side effects as a linked list:

```
class Cons {
    public final int value;     // value of the element
    public final Cons next;     // next element, null if end of list

    public Cons(int value, Cons next) {
        this.value = value;
        this.next = next;
    }
}
```

- We can add an element to the head of a list without changing the list:

```
        Cons list1 = new Cons(3,null);   // this is the list [3]
        Cons list2 = new Cons(5,list1);  // this is the list [5,3]
        Cons list3 = new Cons(1,list2);  // this is the list [1,5,3]
```

  Note that list1 and list2 are still the same lists!

- Such a data structure is called *immutable*. It cannot be changed after creation. By the way, String objects in Java are immutable, too!

Working with an immutable list

- Since an immutable list cannot be changed, we have to create a new list if we want to change list content
- Here is a recursive method that increments all elements of a list by three:

```
public static Cons increment(Cons list) {
    if(list==null)
        return null;
    else
        return new Cons(list.value+3, increment(list.next));
}
```

Note: This is not very efficient code because it's not tail recursive.

- Let's what happens if we use it on the list [1,5]:

  increment(Cons(1, Cons(5, null)))

  → Cons(4, increment(Cons(5, null)))

  → Cons(4, Cons(8, increment(null)))

  → Cons(4, Cons(8, null))

Cons(x,y) means here: a Cons object with value=x and next=y

Exercise on INGInious

- In the INGInious exercises, we ask you to implement a map method and a filter method for an immutable Cons list:
  - The map method takes a function $f: int \rightarrow int$ and applies it to all elements of a list. That means map with a function $f(x) = x + 1$ on the list [1,2,3] should give as result a new list [2,3,4]
  - The filter method takes a predicate $p: int \rightarrow boolean$ and applies it to all elements of a list. The result is a new list with all elements for which $p(x) = true$

    Example: filter with $p(x) = x < 3$ on the list [7,2,1,8] should give a new list [2,1]

- First do the Map/Filter/Cons exercise for lists only containing int values. Then do the exercise with Generics for lists with other value types.