Praktische Softwaretechnologie

Lecture 9.

Károly Bósa
(Karoly.Bosa@jku.at)

Research Institute for Symbolic Computation
(RISC)

The Map Interface

- A Map is an object that maps keys to values.
- A Map cannot contain duplicate keys.
- Each key can map to at most one value.
- It models the mathematical *function* abstraction.

The Map Interface

```
public interface Map<K,V> {
    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk operations
    void putAll(Map<? extends K, ? extends V>
    m);
    void clear();

    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();
```

```
// Interface for entrySet elements
    public interface Entry<K, V> {
        K getKey();
        V getValue();
        V setValue(V value); }
}
```

Map Implementation

- The Java platform contains three general-purpose Map implementations: **HashMap, TreeMap, and LinkedHashMap.** Their behavior and performance are precisely analogous to HashSet, TreeSet, and LinkedHashSet (see before).

- **HashTable** is in the language only because of historical reason and backward compatibility. It is a generic data type, too.

- Each implementation has no argument (**HashMap(), TreeMap(), LinkedHashMap()**) constructor:

```
Map <Type1, Type2> l = new TreeMap<Type1, Type2>();
```

- **No conversion constructors(!):**

Examples for Collection Views

```
for (KeyType key : m.keySet())
    System.out.println(key);
```

with an iterator:

```
// Filter a map based on some property of its keys.
for (Iterator<Type> it = m.keySet().iterator(); it.hasNext(); )
    if (condition(it.next()))
                it.remove();
```

Iterating over key-value pairs (with the help of the interface Map.Entry<K,V>):

```
for (Map.Entry<KeyType,ValType> e : m.entrySet())
    System.out.println(e.getKey() + ": " + e.getValue());
```

Example: Frequency Table of Words

```
import java.util.*;
public class Freq {
        public static void main(String[] args) {
                Map<String, Integer> m = new HashMap<String, Integer>();
                // Initialize frequency table from command line
                for (String a : args) {
                  Integer freq = m.get(a); m.put(a, (freq == null) ? 1 : freq + 1);
                }
                System.out.println(m.size() + " distinct words:");
                System.out.println(m);
         }
}
```

The program generates a frequency table of the words found in its argument list. The frequency table maps each word to the number of its occurrence in the argument list.

Map Example Output

Assume running this program with the following command:

```
java Freq if it is to be it is up to me to delegate
```

The program yields the following output:

```
8 distinct words: {to=3, delegate=1, be=1, it=2, up=1, if=1, me=1, is=2}
```

Preferably the program should print out frequency table in alphabetical order → Change the implementation type of the Map from **HashMap** to **TreeMap**.

This time the program yields the following output:

```
8 distinct words: {be=1, delegate=1, if=1, is=2, it=2, me=1, to=3, up=1}
```

Similarly, if you prefer that the program prints the frequency table in the order the words given in the command line → Change the implementation type of the Map to **LinkedHashMap** Map:

```
8 distinct words: {if=1, it=2, is=2, to=3, be=1, up=1, me=1, delegate=1}
```

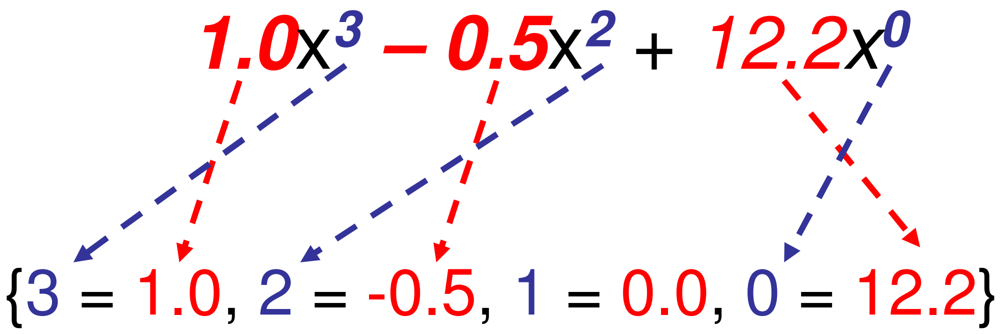
Example: Map Iterations

```
public static void dumpMap(Map mp) {
    Iterator it = mp.entrySet().iterator();
    while (it.hasNext()) {
            Map.Entry pairs = (Map.Entry)it.next();
            System.out.println(pairs.getKey() + " = " + pairs.getValue());
    }
}
```

Example: Representation of Polynomials

$$x^3 - \mathbf{0.5}x^2 + 12.2$$

- **PolynomMap<Integer,Double>:**



$1.0x^3 - 0.5x^2 + 12.2x^0$

{3 = 1.0, 2 = -0.5, 1 = 0.0, 0 = 12.2}

- Keys are the **powers.**
- Values are the **coefficients.**

```
...
List<Map<Integer, Double>> all_polynomials = new
                                        LinkedList<Map<Integer,Double>>();
Map<Integer, Double> pol1 = new TreeMap<Integer, Double>();
Map<Integer, Double> pol2 = new TreeMap<Integer, Double>();
pol1.put(2, 1.0); pol1.put(0, -1.0); all_polynomials.add(pol1);
pol2.put(1, 1.0); pol2.put(0, 1.0); all_polynomials.add(pol2);
```

Example: Converting Polynomial to String 1.

```
public static String toString(Map<Integer, Double> polynomial) {
        StringBuilder sb =new StringBuilder();
        Iterator<Map.Entry<Integer, Double>> it = polynomial.entrySet().iterator();
        while (it.hasNext()) {
            Map.Entry<Integer, Double> pairs = it.next();
            //sign of the coefficient or constant
            if (pairs.getValue().doubleValue() > 0.0) sb.append("+");
            else sb.append("-");
            //coefficient
            sb.append(pairs.getValue());
            //x with its power
            sb.append("x^"+pairs.getKey()+"\t");
        } //while
        return sb.toString();
} //toString
```

E.g.:

```
    pol1.put(2, 1.0); pol(1, -0.0); pol1.put(0, -1.0); System.out.println(toString(pol1));
```

Output:

```
        --1.0x^0  --0.0x^1  +1.0x^2
```

Example: Converting Polynomial to String 2.

```
public static String toString(Map<Integer, Double> polynomial) {
            StringBuilder sb =new StringBuilder();
            Iterator<Map.Entry<Integer, Double>> it = polynomial.entrySet().iterator();
            while (it.hasNext()) {
                Map.Entry<Integer, Double> pairs = it.next();
                //sign of the coefficient or constant
                if (pairs.getValue().doubleValue() > 0.0) sb.append("+");
                else sb.append("-");
                //coefficient
                sb.append(Math.abs(pairs.getValue()));
                //x with its power
                sb.append("x^"+pairs.getKey()+"\t");
            } //while
            return sb.toString();
} //toString
```

E.g.:

```
pol1.put(2, 1.0); pol(1, -0.0); pol1.put(0, -1.0); System.out.println(toString(pol1));
```

Output:

```
-1.0x^0   -0.0x^1   +1.0x^2
```

Example: Converting Polynomial to String 3.

```
public static String toString(Map<Integer, Double> polynomial) {
                StringBuilder sb =new StringBuilder();
                Iterator<Map.Entry<Integer, Double>> it = polynomial.entrySet().iterator();
                while (it.hasNext()) {
                      Map.Entry<Integer, Double> pairs = it.next();
                      //sign of the coefficient or constant
                      if (pairs.getValue().doubleValue() > 0.0) sb.append("+");
                      else sb.append("-");
                      //constant
                      if (pairs.getKey().intValue() == 0) sb.append(Math.abs(pairs.getValue())+"\t");
                      else {
                         //coefficient
                         sb.append(Math.abs(pairs.getValue()));
                         //x with its power
                         sb.append("x^"+pairs.getKey()+"\t");
                      } //else
                } //while
                return sb.toString();
} //toString
```

Output:

```
-1.0 -0.0x^1    +1.0x^2
```

Example: Converting Polynomial to String 4.

Karoly.Bosa@jku.at

```
public static String toString(Map<Integer, Double> polynomial) {
        StringBuilder sb =new StringBuilder();
        Iterator<Map.Entry<Integer, Double>> it = polynomial.entrySet().iterator();
        while (it.hasNext()) {
            Map.Entry<Integer, Double> pairs = it.next();
            //sign of the coefficient or constant
            if (pairs.getValue().doubleValue() > 0.0) sb.append("+");
            else sb.append("-");
            //constant
            if (pairs.getKey().intValue() == 0) sb.append(Math.abs(pairs.getValue())+"\t");
            else {
              //coefficient
              sb.append(Math.abs(pairs.getValue()));
              //x with its power
             if (pairs.getKey().intValue() == 1) sb.append("x"+"\t");
             else sb.append("x^"+pairs.getKey()+"\t");
            } //else
        } //while
        return sb.toString();
} //toString
```

Output:

```
      -1.0 -0.0x       +1.0x^2
```

Example: Converting Polynomial to String 5.

```
public static String toString(Map<Integer, Double> polynomial) {
            StringBuilder sb =new StringBuilder();
            Iterator<Map.Entry<Integer, Double>> it = polynomial.entrySet().iterator();
            while (it.hasNext()) {
                 Map.Entry<Integer, Double> pairs = it.next();
                 //sign of the coefficient or constant
                 if (pairs.getValue().doubleValue() > 0.0) sb.append("+");
                 else sb.append("-");
                 //constant
                 if (pairs.getKey().intValue() == 0) sb.append(Math.abs(pairs.getValue())+"\t");
                 else {
                    //coefficient
                    if (Math.abs(pairs.getValue().doubleValue()) != 1.0)
                         sb.append(Math.abs(pairs.getValue()));
                    //x with its power
                   if (pairs.getKey().intValue() == 1) sb.append("x"+"\t");
                   else sb.append("x^"+pairs.getKey()+"\t");
                 } //else
            } //while
            return sb.toString();
} //toString
```

Output:

```
       -1.0 -0.0x       +x^2
```

Example: Converting Polynomial to String 6.

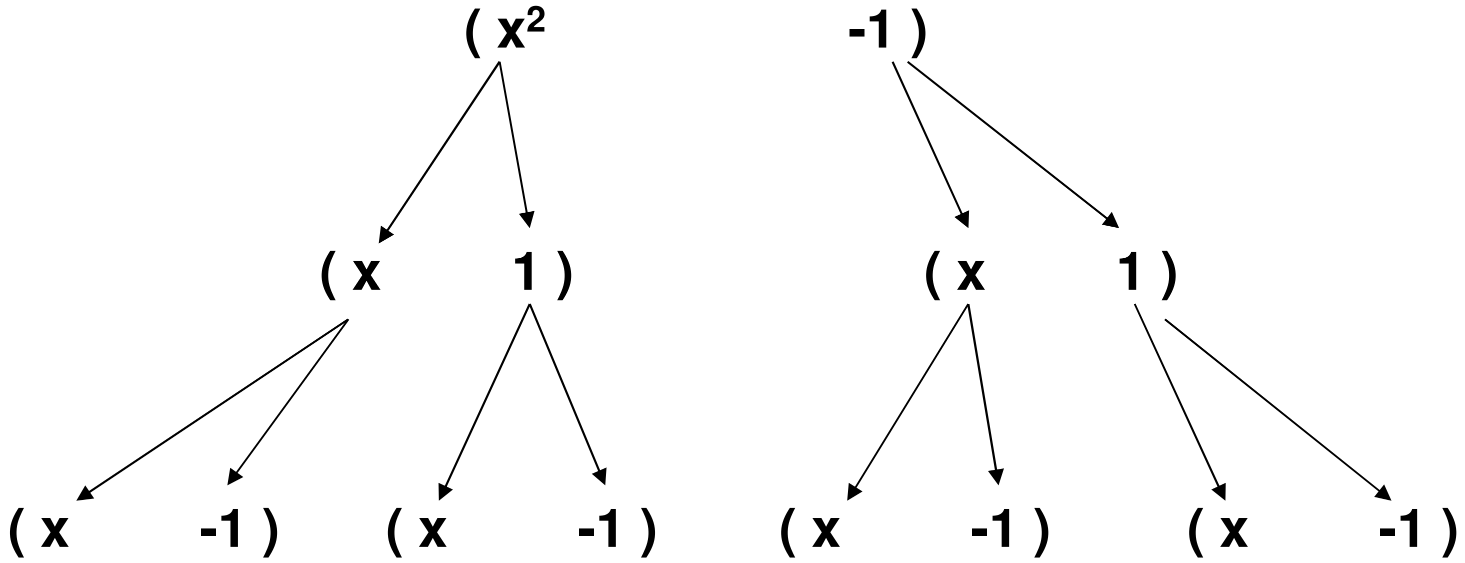
```
public static String toString(Map<Integer, Double> polynomial) {
          StringBuilder sb =new StringBuilder();
          Iterator<Map.Entry<Integer, Double>> it = polynomial.entrySet().iterator();
          while (it.hasNext()) {
              Map.Entry<Integer, Double> pairs = it.next();
              if (pairs.getValue().doubleValue()== 0.0) continue;
              //sign of the coefficient or constant
              if (pairs.getValue().doubleValue() > 0.0) sb.append("+");
              else sb.append("-");
              //constant
              if (pairs.getKey().intValue() == 0) sb.append(Math.abs(pairs.getValue())+"\t");
              else {
                //coefficient
                if (Math.abs(pairs.getValue().doubleValue()) != 1.0)
                    sb.append(Math.abs(pairs.getValue()));
                //x with its power
               if (pairs.getKey().intValue() == 1) sb.append("x"+"\t");
               else sb.append("x^"+pairs.getKey()+"\t");
              } //else
          } //while
          return sb.toString();
} //toString
```

Output:

```
     -1.0             +x^2
```

Example: Polynomial Multiplication 1.

$(x^2-1)(x+1)(x-1)$



$(x^2 \quad -1)$

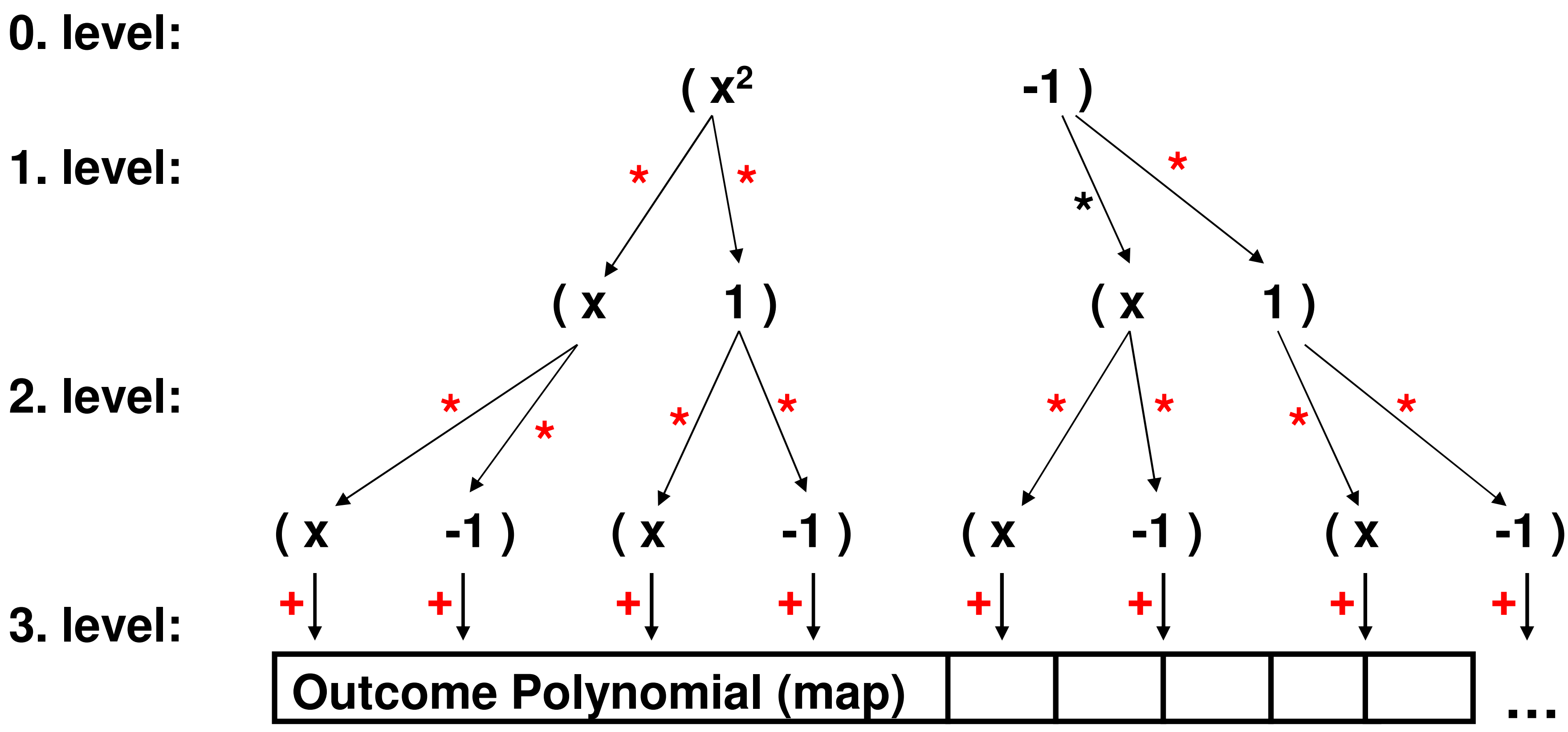
$(x \quad 1) \quad (x \quad 1)$

$(x \quad -1) \quad (x \quad -1) \quad (x \quad -1) \quad (x \quad -1)$

- Good Candidate for a Recursive Algorithm
- Outcome: $x^4 \; -2x^2 +1$

Example: Polynomial Multiplication 1.

$(x^2-1)(x+1)(x-1)$



0. level:

$(\ x^2 \quad -1\ )$

1. level:

$(\ x \quad 1\ )$ $\quad$ $(\ x \quad 1\ )$

2. level:

$(\ x \quad -1\ )$ $\quad$ $(\ x \quad -1\ )$ $\quad$ $(\ x \quad -1\ )$ $\quad$ $(\ x \quad -1\ )$

3. level:

Outcome Polynomial (map) …

- Good Candidate for a Recursive Algorithm
- Outcome: $x^4\ \ -2x^2 +1$

Example: Polynomial Multiplication 2.

```
private static Map<Integer, Double> outcome;

public static Map<Integer, Double> multiplication(List<Map<Integer, Double>> polynomials) {
        outcome = new TreeMap<Integer, Double>();
        recursivePolMult(polynomials, 0, 1, 0);
        return outcome;
}


private static void recursivePolMult(
                        List<Map<Integer, Double>> polynomials,
                        int indexOfNextPolynomial,
                        double curr_coeff,
                        int curr_power) {
        …
}
```

Example: Polynomial Multiplication 3.

```
private static void recursivePolMult(List<Map<Integer, Double>> polynomials,
  int indexOfNextPolynomial, double curr_coeff, int curr_power) {
  Iterator<Map<Integer, Double>>  polynomial_it = polynomials.listIterator(indexOfNextPolynomial);
  if (!polynomial_it.hasNext()) {
     Double current_value = outcome.get(curr_power);
     outcome.put(curr_power, (current_value == null) ? curr_coeff : current_value + curr_coeff);
     return;
  }
  else {
     Map<Integer, Double> polynomial = polynomial_it.next();
     Iterator<Map.Entry<Integer, Double>> coeff_it = polynomial.entrySet().iterator();
     while (coeff_it.hasNext()) {
             double tmp_coeff = curr_coeff;
             int tmp_power = curr_power;
             Map.Entry<Integer, Double> pairs = coeff_it.next();
             tmp_coeff *= pairs.getValue().doubleValue();
             tmp_power += pairs.getKey().intValue();
             recursivePolMult(polynomials, indexOfNextPolynomial+1, tmp_coeff, tmp_power);
      //while}
   } //else
}
```

Deadline: 21.05.2014

Exercise 11: Polynomial Addition

Karoly.Bosa@jku.at

- Download **polynomial.java** from the course web page.
- Your task is to implement the following method of the class ***Polynomial**:*

```
public static Map<Integer, Double> addition(List<Map<Integer, Double>> polynomials) {…}
```

- The method expects a list of polynomial and returns the sum of the given polynomial in the list.
- A skeleton of the solution:
    - Create a new Map (let’s call ***sum***) for the outcome,
    - Go through all the polynomials in the given list one by one,
        - Go through all the power products in the current polynomial,
            - Add the value of the coefficient of the current power product to the corresponding element of the ***sum*** (whose key is equal to the key/power of the current power product),
    - Return ***sum*.**

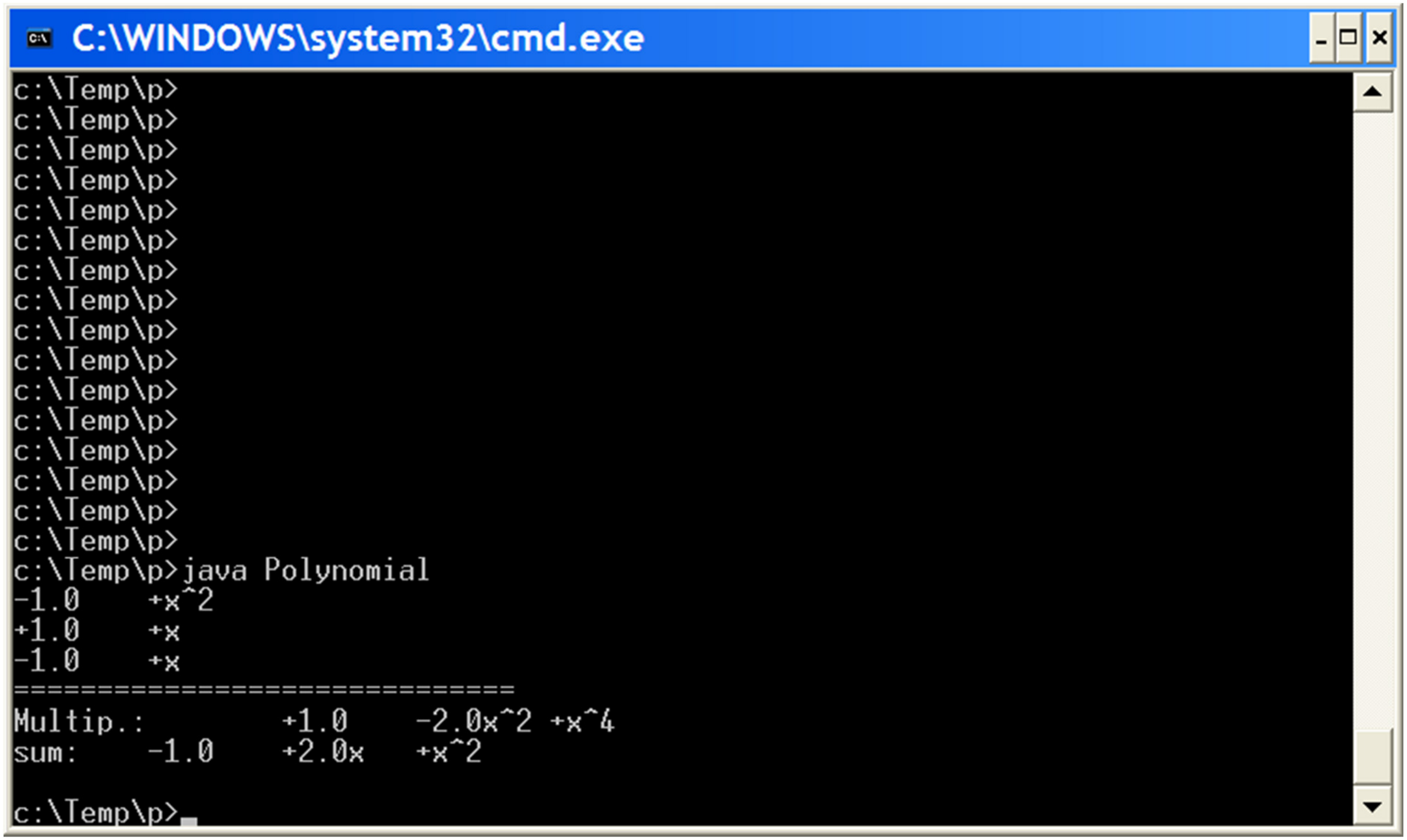
Example: Method main

Karoly.Bosa@jku.at

```
public static void main(String[] args) {
            List<Map<Integer, Double>> all_polynomials = new
                                                LinkedList<Map<Integer,Double>>();
            //key: power of x, value: coefficient
            Map<Integer, Double> pol1 = new TreeMap<Integer, Double>();
            Map<Integer, Double> pol2 = new TreeMap<Integer, Double>();
            Map<Integer, Double> pol3 = new TreeMap<Integer, Double>();
            //first polynomial
            pol1.put(2, 1.0); pol1.put(0, -1.0); all_polynomials.add(pol1);
            System.out.println(toString(pol1));
            //second polynomial
            pol2.put(1, 1.0); pol2.put(0, 1.0); all_polynomials.add(pol2);
            System.out.println(toString(pol2));
            //third polynomial
            pol3.put(1, 1.0); pol3.put(0, -1.0); all_polynomials.add(pol3);
            System.out.println(toString(pol3));

            System.out.println("=================================");
            System.out.println("Multip.:\t"+toString(multiplication(all_polynomials)));
            System.out.println("sum:\t"+toString(addition(all_polynomials)));
 }
```

Example: Output



```
c:\Temp\p>
c:\Temp\p>
c:\Temp\p>
c:\Temp\p>
c:\Temp\p>
c:\Temp\p>
c:\Temp\p>
c:\Temp\p>
c:\Temp\p>
c:\Temp\p>
c:\Temp\p>
c:\Temp\p>
c:\Temp\p>
c:\Temp\p>
c:\Temp\p>
c:\Temp\p>
c:\Temp\p>java Polynomial
-1.0    +x^2
+1.0    +x
-1.0    +x
==============================
Multip.:        +1.0    -2.0x^2 +x^4
sum:    -1.0    +2.0x   +x^2

c:\Temp\p>_
```

Formatting Float and Double Numbers

The ***DecimalFormat*** class can be uses to format decimal numbers into locale-specific strings.

```
Double value = …;

DecimalFormat myFormatter = new DecimalFormat(pattern);
String output = myFormatter.format(value);
System.out.println(value + " " + pattern + " " + output);
```

Patterns:

value	Pattern	Output
123456.789	###,###.###	123,456.789
123456.789	###.##	123456.79
123.78	000000.000	000123.780
12345.67	$###,###.##	$12,345.67
12345.67	\u00A5###,###.###	¥12,345.67

Recommended to Read

Reading and completing the course material from the online Java Tutorial:

http://java.sun.com/docs/books/tutorial/java/index.html

- Learning the Java Language: Numbers and Strings
- Collections

The Java Archive (JAR) file format

- The Java Archive (JAR) file format enables you to bundle multiple files into a single archive file.

- Typically a JAR file contains the class files and auxiliary resources.

- Advantages of the JAR file format (among others):

  - ***Security*: You can digitally sign the contents of a JAR file.

  - ***Compression*: The JAR format allows you to compress your files for efficient storage (lossless ZIP file format).

  - ***Decreased download time***: If your applet is bundled in a JAR file, the applet's class files and associated resources can be downloaded to a browser in a single HTTP transaction.

  - ***Package Versioning*: A JAR file can hold data about the files it contains, such as vendor and version information.

  - …

Creating a JAR File

- To perform basic tasks with JAR files, you use the **Java Archive Tool** provided as part of the JDK (executable: **jar**).

- The basic format of the command for creating a JAR file is:

jar cvf *jar-file input-file(s)*

- The options and arguments used in this command are:
  - The c option indicates that you want to *create* a JAR file.
  - The f option indicates that you want the output to go to a *file* rather than to stdout.
  - The v option produces *verbose* output on stdout while the JAR file is being built.
  - jar-file is the name of JAR file.
  - The input-file(s) argument is a space-separated list of one or more files.

Default Manifest File

- The manifest is a special file that can contain meta information about the files packaged in a JAR file.

- Via manifest the JAR file able to support a wide range of functionality (e.g.: electronic signing, version control, etc.).

- There can be only one manifest file in an archive, and it always has the pathname:

META-INF/MANIFEST.MF

- When you create a JAR file, a default manifest is created automatically with the following content:

```
Manifest-Version: 1.0
Created-By: 1.6.0 (Sun Microsystems Inc.)
```

Specifying the JAR File's Classpath

- You may need to refer classes located in other JAR files from some classes within a JAR file.

- You can specify classpath in the Class-Path header field in the manifest file.

Class-Path: ***jar1-file directory-name/*.class …***

Setting Package Version Information

- You may need to include other meta information (e.g.:package version) in a JAR file's manifest.

Headers	Description
Name	The name of the specification.
Specification-Title	The title of the specification.
Specification-Version	The version of the specification.
Specification-Vendor	The vendor of the specification.
Implementation-Title	The title of the implementation.
Implementation-Version	The build number of the implementation.
Implementation-Vendor	The vendor of the implementation.

Example for Package Version Information

META-INF/MANIFEST.MF:

```
Manifest-Version: 1.0
Created-By: 1.6.0 (Sun Microsystems Inc.)
Name: java/util/
Specification-Title: Java Utility Classes
Specification-Version: 1.2
Specification-Vendor: Sun Microsystems, Inc.
Implementation-Title: java.util
Implementation-Version: build57
Implementation-Vendor: Sun Microsystems, Inc.
```

Modifying a (Default) Manifest File

- You use the **m command-line option** to add custom information to the manifest **during creation of a JAR file.**

- To modify the manifest, you must first prepare a text file containing the information you wish to add to the manifest, e.g.:

```
Class-Path: jar1-file  directory-name/*.class   …
<empty_line>
```

- Then use the Jar tool's m option to add the information in your file to the manifest:

```
jar cfm jar-file manifest-addition.txt input-file(s)
```

- **Warning:** The text file from which you are creating the manifest must end with a new line or carriage return.

Viewing the Contents of a JAR File

- The basic format of the command for viewing the contents of a JAR file is:

jar tf *jar-file*

- The options and arguments used in this command are:
  - The t option indicates that you want to view the *table* of contents of the JAR file.
  - The f option indicates that the JAR file whose contents are to be viewed is specified on the command line.
  - The jar-file argument is the path and name of the JAR file whose contents you want to view.

Extracting the Contents of a JAR File

- The basic command to use for extracting the contents of a JAR file is:

```
jar xf jar-file [archived-file(s)]
```

- The options and arguments used in this command are:
  - The x option indicates that you want to *extract* files from the JAR archive.
  - The f options indicates that the JAR *file* from which files are to be extracted is specified on the command line, rather than through stdin.
  - The jar-file argument is the JAR file from which to extract files.
  - The archived-file(s) is an optional argument consisting of a space-separated list of the files to be extracted from the archive.

Updating a JAR File

- To update the contents of an existing JAR file by modifying its manifest or by adding files:

jar uf *jar-file input-file(s)*

- In this command:
  - The u option indicates that you want to *update* an existing JAR file.
  - The f option indicates that the JAR file to update is specified on the command line.
  - jar-file is the existing JAR file that's to be updated.
  - input-file(s) is a space-deliminated list of one or more files that you want to add to the Jar file.
- Any files already in the archive having the same pathname and name as a file being added will be **overwritten**.

Running JAR-Packaged Application

- You can run JAR-packaged applications with the Java virtual machine:

```
java -jar jar-file
```

- The -jar flag tells the interpreter that the application is packaged in the JAR file format.

- Before you execute this command make sure the runtime environment has an information of which class within the JAR file is **the application's entry point**.

- You must add a Main-Class header to the JAR file's manifest:

```
Manifest-Version: 1.0
Created-By: 1.6.0 (Sun Microsystems Inc.)
Main-Class: packageName.className
```

- The *className* is the name of the class that's the application's entry point(that contains a method *main*).

Setting an Entry Point with the JAR Tool

- The 'e' flag (for 'entrypoint'), introduced in JDK 6, creates or overrides the manifest's Main-Class attribute.

- It can be used while creating or updating a jar file:

jar cfe *jar-file*.jar *entry-point input-file(s)*

- Entry-point is always a class name not a file name (do not use .class file extension).

- If the entrypoint class name is in a package it may use a '.' (dot) character as the delimiter, e.g.:

jar cfe *jar-file*.jar *package.entry-point package*/*.class

Running JAR-Packaged Applet

- If the applet is bundled as a JAR file, it must be used the *ARCHIVE* parameter to specify the relative path to the JAR file:

```
<applet code=AppletClassName.class
             archive=“dir-path/jar-file.jar"
            width=120 height=120>
</applet>
```