GUI - lecture notes

with example programs from archive ***java_examples***

Tomasz R. Werner

Warsaw, May 6, 2024 13:32

Contents

Section 1

Bitwise operators

operator!bitwise

We can operate on variables of integral types (mainly **int**) treating them as "buckets" of single bits. In what follows, remember that operations of shifting, ANDing, ORing etc., that we discuss, do not modify their arguments: they return *new* values that we have to handle in some way (display it, assign to a variables, and so on).

As we know, data in a variable is stored as a sequence of bits, conventionally represented by 0 and 1. In particular an **int** consists of 32 bits. We can interpret individual bits as coefficients at powers of 2: the rightmost (least significant) bit is the coefficient at 2^0, the next, from the right, at 2^1, the next at 2^2 and so on, to the last (i.e., the leftmost, most significant) bit which stands at 2^{31}.

Shifting

Shift operators act on values of integral types: they yield another value, which corresponds to the original one but with all bits shifted by a specified number of positions to the left or to the right.

Left shift (<<) moves the bit pattern to the left: bits on the left which go out of the variable are lost, bits which enter from the right are all 0. The value to be shifted is given as the left-hand operand while number of positions to shift – by the right-hand operand. For example (we use only eight bits to simplify notation, in reality there are 32 bits in an **int**):

```
a           1 0 1 0 0 1 1 0
a << 3      0 0 1 1 0 0 0 0
```

The *unsigned* right shift operator (>>>) does the same but in the opposite direction

```
a           1 0 1 0 0 1 1 0
a >>> 3     0 0 0 1 0 1 0 0
```

The *signed* right shift operator (>>) behaves in a similar way, but what comes in from the left is the sign bit: if the leftmost bit is 0, zeros will come in, if it is 1, these will be ones

```
a           1 0 1 0 0 1 1 0
a >> 3      1 1 1 1 0 1 0 0
b           0 0 1 0 0 1 1 0
b >> 3      0 0 0 0 0 1 0 0
```

ANDing, ORing, etc.

Bit-wise operations work similarly to logical ones (ANDing, ORing, XORing, negating) but operate on individual bits of their operands which must be of an integral type. For example, when ANDing two values with bit-wise AND operator (&) we will get a new value, where on each position there is 1 if and only if in both operands there were 1s on this position, and 0 otherwise

```
a           1 0 1 0 0 1 0 0
b           1 0 0 0 0 1 1 0
a & b       1 0 0 0 0 1 0 0
```

For bit-wise OR operator (|), each bit of the result is 1 if there is at least one 1 at the corresponding position in operands, and 0 if both bits in the operands are 0

```
a           1 0 1 0 0 1 0 0
b           1 0 0 0 0 1 1 0
a | b       1 0 1 0 0 1 1 0
```

The bit-wise XOR operator (^), sets a bit of the result to 1 if bits at the corresponding position in operands are different, and to 0 if they are the same

```
a           1 0 1 0 0 1 0 0
b           1 0 0 0 0 1 1 0
a ^ b       0 0 1 0 0 0 1 0
```

As can be expected, negating operator (~) just reverses (flips) the bits

```
 a          1 0 1 0 0 1 0 0
~a          0 1 0 1 1 0 1 1
```

Let us notice here, that XORing has an interesting and useful feature. Let us see the result of XORing a bit sequence `a` with all ones:

```
a           1 0 1 0 0 1 0 0
b           1 1 1 1 1 1 1 1
a ^ b       0 1 0 1 1 0 1 1
```

We see that XORing the value `a` with all ones gives negation of `a` — wherever there was 1 in `a`, we get 0, and *vice versa*.
Now let us try to XOR our `a` with all zeros:

```
a           1 0 1 0 0 1 0 0
b           0 0 0 0 0 0 0 0
a ^ b       1 0 1 0 0 1 0 0
```

This time what we got is exactly the same as `a`! So, XORing a bit with 1 flips its value, XORing it with zero – reproduces the same value.

Listing 1 BAA-Bits/Bits.java

```java
public class Bits {
    public static void main (String[] args) {
        int a = 0b11111111;    // 255 or 0xFF
        System.out.println("a = " + a);
        int b = 0x7F;          // 127
        System.out.println("b = " + b);

        a = 3; // 00...011
        System.out.println(a + " " + (a << 1) + " " +
                           (a << 2) + " " + (a << 3));
        a = -1;
        int firstByte  = a & 255;
        int secondByte = (a >> 8) & 0xFF;
        System.out.println("-1: " + secondByte + " " +
                                    firstByte);
```

```
        a = 0b1001;
        b = 0b0101;
        System.out.println("AND: " + (a & b) + "; " +
                            "OR: " + (a | b) + "; " +
                           "XOR: " + (a ^ b) + "; " +
                           " ~a: " + (~a)    + "; " +
                           " ~b: " + (~b));
    }
}
```

Section 2

IO streams primer

2.1 Binary and text IO streams

An IO stream can be viewed as a sequence of data (ultimately, these are always just bytes) 'flowing' from a 'source' to a 'sink' (destination). All IO streams fall into two categories

- output streams: data from our programs (values of variables, objects, arrays etc.) are the "source" while the destination is a file, a socket, the console or even a region in memory;
- input streams: our programs (variables, arrays etc.) is now the destination, while the source might be a file, a socket, the keyboard, etc.

Java represents characters by their two-byte Unicode codes (the so called *code points*). Therefore, there is a problem with reading and writing texts: any text is written using some encoding, so Java must translate a byte or a sequence of bytes corresponding to a character into a two-byte code point and *vice versa*. Therefore, we have to distinguish byte (binary) streams, where bytes are treated just as bytes, without any modifications, and text (also called *character*) streams where bytes are subject to some transformations, dependent of the encoding in use. Hence, we end up with four types of IO streams:

- output byte (binary) streams: they all correspond to subclasses of the class **OutputStream** (e.g., **ByteArrayOutputStream**, **FileOutputStream**, **ObjectOutputStream**);
- output text streams: they correspond to subclasses of **Writer** (e.g., **BufferedWriter**, **CharArrayWriter**, **OutputStreamWriter**, **PrintWriter**, **StringWriter**);
- input byte streams: they correspond to subclasses of **InputStream** (e.g., **ByteArrayInputStream**, **FileInputStream**, **ObjectInputStream**, **StringBufferInputStream**);
- input character streams: they correspond to subclasses of **Reader** (e.g., **BufferedReader**, **CharArrayReader**, **InputStreamReader**, **StringReader**);

The main four classes mentioned above are *abstract*, what means that one cannot create objects of these types, only objects of their concrete subclasses.

Any Java process, as other processes, is given by the operating system three standard streams. They are represented by objects which are static fields of the class **System**

- System.out: representing the standard output stream (by default it is connected to the terminal's screen);
- System.in: representing the standard input stream (by default it is connected to the terminal's keyboard);
- System.err: representing the standard error stream (by default it is connected to the terminal's screen).

Both System.out and System.err are of type **PrintStream** which inherits (indirectly) from **OutputStream** (but is designed to handle text output) while System.in is of a type inheriting from **InputStream** (and is a binary stream). The difference between System.out and System.err is that the former is *buffered*, while the latter is not. When we write to a buffered stream, like System.out, we are in fact writing to a buffer which will be eventually flushed to its destination (by default, to the screen) in larger chunks. Sometimes such behavior is not desired, especially when we expect some exception handling or when we write to a socket. Printing to unbuffered streams (like System.err) is immediate — data is not stored in any buffer, it goes directly to the destination of the stream.

All I/O operations[1] may throw exceptions of types extending **IOException** and are *checked*; therefore, when using them, we have to handle possible failures somehow.

Let us consider the following example. In the simple program below, we first write, in a loop, individual bytes of a **long**, starting with the most significant one, to the file. Then we read these bytes back and reconstruct a **long** with the same value. Note the special form of the **try** clause, the so called *try-with-resources*. In the round parentheses, we create I/O streams and references to these streams. Note that they will be in scope of the **try** clause. Normally, we have to remember to close all streams that have been opened. The *try-with-resources* construct, however, takes care of closing streams no matter what happened, even if an exception has been thrown. Therefore, no **finally** clause is needed here.

Listing 2 KEP-WriteBin/WriteBin.java

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class WriteBin {
    public static void main (String[] args) {
        long before = -284803830071168L;
        System.out.println("before = " + before);
        try (
            OutputStream os =
                    new FileOutputStream("WriteBin.bin");
        ) {
            for (int i = 7; i >= 0; --i)
                os.write( (int)(before >> i*8) );
        } catch(IOException e) {
            e.printStackTrace();
            System.exit(1);
        }

        long after = 0;
        try (
            InputStream is =
                    new FileInputStream("WriteBin.bin");
        ) {
```

[1]except those on *PrintStream* objects, which ‘consumes’ possible exceptions in its methods

```java
            for (int i = 0; i < 8; ++i)
                after = (after << 8) | is.read();
        } catch(IOException e) {
            e.printStackTrace();
            System.exit(1);
        }
        System.out.println("after  = " + after);
    }
}
```

The program prints

```
before = -284803830071168
after  = -284803830071168
```

confirming that we have written and read bytes correctly. Note, that the created file has exactly 8 bytes and contains the full information about our **long**. Writing a long in the text form requires up to 20 characters (2.5 times more!). Moreover, conversion of a number to its string representation (and then back) is also a quite expensive operation.
In practice, we don't have to manipulate the bytes of variables to store them on disk in their binary form — as we will see later, there are special tools in the library that make such tasks trivial.

In the next example, we first read from **InputStream** (connected to just System.in). As this is a binary stream, by invoking **read** we get consecutive bytes (as **int**s), which we can cast on **char**. We will get -1, when the end of data is reached (note that -1 doesn't correspond to any legal character). This, however, will never happen with System.in, so we just detect the LF character ('\n') to stop reading. A text may contain characters encoded on two or three bytes — such characters will not be read correctly. However, we can pass the System.in object to the constructor of **InputStreamReader** which is kind of a 'translator' yielding an object behaving as a text stream (given a specific encoding). In principle, we could have passed it further to the constructor of **BufferedReader**: using this object we can read characters much more efficiently; moreover, it also supports the very convenient **readLine** method which allows us to read a whole line at once (as we will see in one of the following examples).

Listing 3 BHJ-BytesChars/BytesChars.java

```java
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.Reader;
import static java.nio.charset.StandardCharsets.UTF_8;

public class BytesChars {
    public static void main(String[] args) {
        System.out.print(
                "Type something and press enter ==> ");
        InputStream is = System.in;
        try {
```

```
            char c = ' ';
            while (true) {
                int i = is.read();
                c = (char)i;
                if (c == '\r' || c == '\n') break;
                System.out.printf("%3d ('", i);
                System.out.println(c + "') =>" +
                    " digit:" + Character.isDigit(c) +
                    " letter:" + Character.isLetter(c) +
                    " white:" + Character.isWhitespace(c));
            }
        } catch(IOException e) {
            e.printStackTrace();
            return;
        }
          // we don't close 'is' here, as this is the
          // standard input and we will use it later
        System.out.print("Type something again ==> ");
        try (
            Reader rd =
                new InputStreamReader(System.in, UTF_8)
        ) {
            char c = ' ';
            while (true) {
                int i = rd.read();
                c = (char)i;
                if (c == '\r' || c == '\n') break;
                System.out.printf("%#5x ('", i);
                System.out.println(c + "') =>" +
                    " digit:" + Character.isDigit(c) +
                    " letter:" + Character.isLetter(c) +
                    " white:" + Character.isWhitespace(c));
            }
        } catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

Running the program, we can get:

```
Type something and press enter ==> Żółć
197 ('Å') => digit:false letter:true white:false
187 ('»') => digit:false letter:false white:false
195 ('Ã') => digit:false letter:true white:false
179 ('³') => digit:false letter:false white:false
197 ('Å') => digit:false letter:true white:false
130 ('') => digit:false letter:false white:false
196 ('Ä') => digit:false letter:true white:false
135 ('') => digit:false letter:false white:false
```

```
Type something again ==> Żółć
0x17b ('Ż') => digit:false letter:true white:false
0xf3 ('ó') => digit:false letter:true white:false
0x142 ('ł') => digit:false letter:true white:false
0x107 ('ć') => digit:false letter:true white:false
```

As we can see, in the first case multi-byte characters (as, e.g., in Żółć) are not read correctly. This is because (at least under Linux) the text from the console is in UTF-8 encoding, in which a single character may occupy 1, 2, 3, or even four bytes. The stream System.in is, however, a byte (binary) stream, so each **read** consumes one byte, which not necessarily corresponds to any character, as it may be only a part of a multi-byte character.
The situation is different in the second case — here we 'wrap' System.in in InputStreamReader (which behaves as a text stream), passing also the correct encoding. Object of this wrapper (*decorator*) class behaves as a *text* stream, so each **read** consumes one *character*, no matter how many bytes it takes.

Let us show now how one can read and write text files, what is a very common task. This can be done in various ways, so consider the program below as an example, not necessarily the best in all situations. Note that when dealing with text files, one should always specify the encoding of all input/output files.

Listing 4 KFE-GrepNew/GrepNew.java

```java
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import static java.nio.charset.StandardCharsets.UTF_8;

public class GrepNew {
    public static void main(String[] args) {
        String iFileName = "alice.txt";
        String oFileName = "grep_alice.txt";
        String wordSearchedFor = "Cheshire";

        Path filein = Paths.get(iFileName);
        if (!Files.exists(filein)     ||
            !Files.isReadable(filein) ||
             Files.isDirectory(filein)) {
            System.out.println("Invalid input file!!!");
            return;
        }

        try (
               // UTF8 is the default in nio classes,
               // but not in older io classes
            BufferedReader br =
                Files.newBufferedReader(filein, UTF_8);
```

```java
            BufferedWriter bw =
                Files.newBufferedWriter(
                    Paths.get(oFileName), UTF_8))
        {
            String line;
            int lineNo = 0;
            while ( (line = br.readLine()) != null) {
                ++lineNo;
                if (line.indexOf(wordSearchedFor) >=0)
                    bw.write(String.format("Line %3d: %s%n",
                                           lineNo,line));
            }
            System.out.println("Results written to " +
                                oFileName);
        } catch(IOException e) {
            System.out.println("Something wrong");
            System.exit(1);
        }
    }
}
```

The program creates the file ***grep_alice.txt*** containing

```
Line 1429:   `It's a Cheshire cat,' said the Duchess, `and that's why.
Line 1437:   I didn't know that Cheshire cats always grinned; in fact, I
Line 1561: the Cheshire Cat sitting on a bough of a tree a few yards off.
Line 1567:   `Cheshire Puss,' she began, rather timidly, as she did not at
Line 2234: be a grin, and she said to herself `It's the Cheshire Cat:  now I
Line 2270:   `It's a friend of mine--a Cheshire Cat,' said Alice:  `allow me
Line 2317:   When she got back to the Cheshire Cat, she was surprised to
```

Here, we used classes from the ***java.nio*** package, which is newer than ***java.io*** and usually recommended. Objects of class **Path** represent the names of files in the current file system (not necessarily existing ones). We can create such objects by calling the static factory method **get** and passing (absolute or relative) name of a file. Then one can check if such a file exists, whether it is readable, executable, what its size or last modification time is, etc.

Note also the form of the **try** clause. Here, we used again the *try-with-resources* construct. Before the opening brace, in round parentheses, we create objects representing "resources" — in this case streams. These could be also other types or resources, like data base connections; what is important is that they have to be *closeable* (in other words, they have to implement the **Closeable** interface). If there are more than one, as here, we separate them by a semicolon. As we remember, the benefit of this form of the **try** clause is that we don't have to bother with closing the resources — they will be automatically closed whether an exception has occurred or not. Moreover, this form inserts variables declared in parentheses to the scope of the **try** clause, but *not* to the outer scope, keeping the outer scope unpolluted by variables that are not needed there.

In the loop reading the file, we call **readLine** on the **BufferedReader** object. In this way, we can read the file line by line not bothering about detecting the LF character ourselves. When the end of file is reached, **readLine** returns **null**, otherwise it returns the next line as a string (with the LF character *chopped off*).

For completeness, another version of essentially the same program is shown below. Here, we don't use classes from the ***java.nio*** package, but from an older (but still used and still necessary) ***java.io*** package.

Listing 5 KFD-Grep/Grep.java

```java
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class Grep {
    public static void main(String[] args) {
        File filein = new File("alice.txt");
        String wordSearchedFor = "Cheshire";

        if ( !filein.exists()     ||
             !filein.canRead()    ||
              filein.isDirectory()  ) {
            System.out.println("Invalid input file !!!");
            System.exit(1);
        }
        File fileou = new File("grep_" + filein.getName());
        BufferedReader br = null;
        BufferedWriter bw = null;
        String LF=System.getProperty("line.separator");

        try {
            br = new BufferedReader(
                     new FileReader(filein));
            bw = new BufferedWriter(
                     new FileWriter(fileou));
            String line;
            int lineNo = 0;
            while ( (line = br.readLine()) != null) {
                ++lineNo;
                if (line.indexOf(wordSearchedFor) >=0)
                    bw.write(String.format("Line %2d: %s%s",
                                           lineNo,line,LF));
            }

        } catch (IOException e) {
            System.out.println("Problems with reading");
        } finally {
            try { if (br != null) br.close(); }
            catch(IOException ignore) { }
```

```java
            try { if (bw != null) bw.close(); }
            catch(IOException ignore) { }
            System.out.println("Results written to " +
                                fileou.getAbsolutePath());
        }
    }
}
```

There are some differences to be noted: class **File** which, to some extend, plays the rôle of the class **Path** from the ***java.nio.file*** package. Notice also, that **BufferedReader** must be created in two steps: first we create 'raw' byte stream of type, e.g., **FileInputStream**, then we pass it to the constructor of **InputStreamReader** with, as the second argument, the required encoding, and then this to the constructor of the **BufferedReader**. In the example above there are only two steps: to the constructor of **BufferedReader** we pass directly an object of type **FileReader**, but then there is no way to specify the encoding — default system encoding will be assumed. Note how *try-with-resources* simplifies operations on IO streams; in the program above, we don't use it and therefore the somewhat complicated **finally** clause is needed (as **close** may itself also throw an exception).

It is sometimes useful to a read text file into a single string (for example, to process it with regular exceptions). This is a very common task, and it can be accomplished, for example, as shown below

Listing 6 KFI-File2Str/File2Str.java

```java
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import static java.nio.charset.StandardCharsets.UTF_8;

public class File2Str {
    public static void main(String[] args) {
        String text = null;

        try {
            byte[] bytes =
                Files.readAllBytes(Paths.get("pangram.txt"));
            text = new String(bytes, UTF_8);
        } catch(IOException e) {
            System.out.println(e.getMessage());
            System.exit(1);
        }
        System.out.println("***** File read (1):\n" + text);

          // simpler way, since java 11
        try {
            text = Files.readString(
                                Paths.get("pangram.txt"), UTF_8);
        } catch(IOException e) {
```

```
            System.out.println(e.getMessage());
            System.exit(1);
        }
        System.out.println("***** File read (2):\n" + text);
    }
}
```

2.2 *StreamTokenizer class*

The last example illustrates the **StreamTokenizer** utility class. It reads from a text file and splits the input into 'tokens' – single pieces of information (treating spaces as separators, but this can be changed). After reading a token with the **nextToken** method, the field ttype contains information about the type of this token in the form of a predefined integer constant: TT_NUMBER if the token can be interpreted as a number, TT_WORD if it's a string, TT_EOL if it's the end-of-line character, TT_EOF if the end of file has been reached. When the token is a number or a string, one can get their values from the fields nval (**double**) or sval (**String**), respectively.
For example, for a data file like this

```
Tokyo 38 Delhi
25.7     Shanghai 23.7 SaoPaulo 21 Mumbai 21
```

the following program

Listing 7 HUS-Tokens/Tokenizer.java

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.FileNotFoundException;
import java.io.StreamTokenizer;
import java.nio.file.Files;
import java.nio.file.Paths;
import static java.io.StreamTokenizer.TT_EOF;
import static java.io.StreamTokenizer.TT_NUMBER;
import static java.io.StreamTokenizer.TT_WORD;

public class Tokenizer {
    public static void main(String[] args) {
        double result = 0;
        StringBuilder sb = null;
        try ( // UTF8 assumed
            BufferedReader br =
                Files.newBufferedReader(
                    Paths.get("Tokenizer.dat")))
        {
            StreamTokenizer sTok = new StreamTokenizer(br);
            sTok.eolIsSignificant(false);
            sTok.slashSlashComments(true);
            sTok.slashStarComments(true);
```

```
            sb = new StringBuilder();
            while (sTok.nextToken() != TT_EOF) {
                switch (sTok.ttype) {
                case TT_NUMBER:
                    result += sTok.nval;
                    break;
                case TT_WORD:
                    sb.append(" " + sTok.sval);
                    break;
                }
            }
        } catch (FileNotFoundException e) {
            System.err.println("Input file not found");
            return;
        } catch(IOException e) {
            System.err.println("IO Error");
            return;
        }
        System.out.println("Total population: " + result);
        System.out.println("Cities: " +
                    sb.toString().substring(1));
    }
}
```

will print

```
Total population: 129.4
Cities: Tokyo Delhi Shanghai SaoPaulo Mumbai
```

2.3 IO cheat sheet

Let us summarize, as a reference, basic forms of reading from or writing to IO streams.

2.3.1 Reading/writing binary files byte-by-byte

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;

// ...

    try (
        InputStream  fis = new FileInputStream("fi.bin");
        OutputStream fos = new FileOutputStream("fo.bin")
    ){
        int n;
        while ( (n = fis.read()) != -1) {
            char c = (char)n;
```

```
            // ...
            System.out.print(c);
            fos.write(n);
        }
    } catch(IOException e) { /* ... */ }
```

To make reading/writing more efficient, one may also use buffered streams:

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;

// ...

    InputStream fis =
        new BufferedInputStream(
            new FileInputStream("fi.bin"));
    OutputStream fos =
        new BufferedOutputStream(
            new FileOutputStream("fo.bin"))
```

2.3.2 Reading a binary file into an array of bytes

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;

// ...

    byte[] ba = null;
    try {
        ba = Files.readAllBytes(Paths.get("fi.bin"));
    } catch(IOException e) { /* ... */ }
    for (byte b : ba) System.out.print((char)b);
```

2.3.3 Reading/writing text files line-by-line

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import static java.nio.charset.StandardCharsets.UTF_8;

// ...

    try (
        BufferedReader br =
            Files.newBufferedReader(
                Paths.get("fi.txt"), UTF_8);
        BufferedWriter bw =
            Files.newBufferedWriter(
                Paths.get("fo.txt"), UTF_8)
```

```
    ){
        String line;
        while ( (line = br.readLine()) != null) {
            System.out.println(line);
            bw.write(line);
            bw.newLine();
        }
    } catch(IOException e) { /* ... */ }
```

2.3.4 Reading/writing text files character-by-character

One can use **BufferedReader** as in the example above but, instead of **readLine**, call **read**

```
      // Reads one character and returns it as an int.
      // Returns -1 when the end of file has been reached
    int  n = br.read();
    char c = (char)n;
```

To read text from the keyboard, you can use

```
import java.io.InputStreamReader;
import java.io.Reader;
import java.io.IOException;
import static java.nio.charset.StandardCharsets.UTF_8;

    try (
        Reader rd =
            new InputStreamReader(System.in, UTF_8)
    ) {
        while (true) {
            int i = rd.read();
            // ... a condition to stop the loop
            c = (char)i;
            System.out.print(c);
        }
    } catch(IOException e) {
        e.printStackTrace();
    }
```

Section 3

Regular expressions

Regular expression (**regex**) is a sequence of characters which defines a pattern that we want to search for in a string (generally, in a text which may be arbitrary long). Regexes are 'compiled' into a form resembling functions and executed by the so called *regular expression engines* — almost all contemporary languages support regular expressions (built into the language or as part of their standard libraries). The theory behind regexes is rather involved and its full understanding requires quite advanced mathematical knowledge; it was developed by an outstanding logician Stephen Cole Kleene (pronounced KLAY-nee) in 1950s and first used in practice in early implementations of Unix text processors and utility programs (Ken Thompson). Almost all modern implementations of regular expression engines are based on Larry Wall's implementation in his Perl programming language (late 1980s).

Details on the Java implementation: see Oracle's documentation.[2]

3.1 Basic concepts

Suppose we are looking for a word, say 'elephant', in a text. Then the regular expression which defines this word will be just `"elephant"`. But what if we have, in our text, the word 'Elephant'? Of course, this won't match, because the first letter differs. Or, we look for 'cat' but only if it is a separate word, not part of another word (like in 'tomcat' or 'caterpillar'). Or, we are looking for numbers (sequences of digits) but we don't know in advance what numbers occur in our text and of what length (number of digits) they are. All these problems can be easily solved with the help of regular expressions which allow us to formulate such requirements as 'a sequence of letters', 'any uppercase letter followed by a dot', 'a sequence of at least four but at most seven digits the first of which is not 0', 'two words separated by one or more spaces or TAB characters', etc.

3.1.1 Classes

Classes define sets of characters. They are specified in square brackets — a hyphen between characters denotes a range, a 'hat' (`^`) at the beginning denotes negation, the `&&` symbol stands for ANDing. For example:

- `[abc]` — set of three letters: 'a', 'b' and 'c',
- `[a-d]` — set of four letters: 'a', 'b', 'c' and 'd',
- `[a-cu-z]` — set of lowercase letter from ranges `[a-c]` and `[u-z]`,
- `[a-zA-Z]` — set of lower- and uppercase Latin letter,
- `[a-zA-Z0-9_]` — set of all Latin letter and digits, and underscore,
- `[^0-9]` — any character, but *not* a digit,
- `[a-z&&[^i-n]]` — any character in the range `[a-z]` but simultaneously *not* in the range `[i-n]` (therefore equivalent to `[a-ho-z]`).

As we can see, there is a way to AND, but what about ORing? This can also be achieved with symbol `|`: regex `cat|dog` will look for 'cat' OR 'dog'.

[2] https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/regex/Pattern.html

3.1.2 Predefined classes

Some most useful classes are predefined and denoted by special symbols; sometimes it is just one letter after a backslash — in these cases uppercase letter means the same as the corresponding lowercase symbol, but negated. For example:

- `\d` — any digit, `\D` — any non-digit,
- `\s` — any white character (space, tab, new line), `\S` — anything except white character,
- `\w` — any Latin letter, digit or underscore, `\W` — negation of `\w` (also non-Latin letters if UNICODE_CHARACTER_CLASS option is enabled, see sec. 3.5.4),
- `.` — any character except the new line (or including it, if DOTALL option has been selected, see sec. 3.5.2),
- `\p{P}` — any punctuation character, `\P{P}` — anything but not punctuation,
- `\p{L}` — any letter, in any language, `\P{L}` — not a letter,
- `\p{Ll}` — any lowercase letter, in any language,
- `\p{Lu}` — any uppercase letter, in any language,
- etc.,

NOTE: As we have seen, and will see it again quite often, backslash symbol (\) is very common in regular expression. It must be seen by the regex engine, but there is a problem: inside string literals it has a special meaning for the compiler (or rather its part – *tokenizer*.) How to enter it into our string? Just double it: the symbol \\ inside a string literal denotes a *single* backslash!

3.1.3 Special locations

There are symbols which denote not characters but rather special locations in the text being analyzed. For example:

- `\b` — word boundary — just before or just after a word (a word is a sequence of characters matching `\w`),
- `^` — beginning of the text (or of a line, if MULTILINE option is enabled, see sec. 3.5.1),
- `$` — end of the text (or of a line, if MULTILINE option is enabled).

3.1.4 Quantifiers

You can put a so called *quantifier* just after an element of a regex. It then determines a possible number of repetitions of this element. For example:

- `+` — once or more,
- `*` — any number of occurrences (including zero),
- `?` — once or not at all,
- `{n,m}` — number of occurrences in the range $[n, m]$,
- `{n}` — exactly `n` occurrences.
- `{n,}` — at least `n` occurrences.

All these quantifiers are by default **greedy**. It means that the regex engine will try to find the *longest* possible match. For example, if our regex is `a.*z` and the text is `"abzczdz"`, then the whole text will be found as the match, even though substrings `"abz"` and `"abzcz"` would be also possible (but are shorter). If this is not what we want, we can make a quantifier **reluctant**, i.e., it will try to find the *shortest* match — in the above example `"abz"` would be found. To make a quantifier reluctant, we just add a question mark (?). Thus, continuing the above example, the regex `a.*?z` would find the shortest match (`"abz"`).

There exist the third kind of quantifiers, the so called **possessive** quantifiers, denoted by a plus symbol (+). It is, in a sense, even more greedy that greedy quantifier, because it never steps back. Normally, for a greedy quantifier, something like `.*` consumes everything and then, if there is no match, the matcher slowly backs off: it makes one step backward to see if there is a match now, if not, it steps back one more character again, and so on.

For example, let us assume that the regex is `"a.*bc"` (so `.*` is, by default, greedy) and the text is `"abcdbc"`. First, after 'a', the `.*` will consume everything. But there is no 'bc' after 'everything', so the matcher steps back one character and now there is 'c' at the end. This is still not 'bc', so the matcher makes one more step back and now it has 'bc' at the end — matching succeeds with the whole text `"abcdbc"` as the match.

Now suppose the quantifier is reluctant: `"a.*?bc"`. After 'a' the matcher consumes the shortest substring matching `.*`, that is nothing. There is no match, because there is no 'bc', so the matcher makes one step *forward* and indeed there is 'b'. After making one more step, there is 'bc', so matching succeeds and the matched substring is `"abc"`.

Now the possessive case: `"a.*+bc"`. The `.*` consumes everything, and there is no 'bc' after that. Possessive matcher never steps back, so the matching fails. Generally, you can live without possessive quantifiers, and in fact in many languages they are not supported at all. However, when they are appropriate, they make the whole process of searching for a match faster (because the matcher does not have to remember intermediate states, as it will never need to go back).

3.2 Regular expressions in methods of class String

There are some very useful methods in class **String** which use regular expressions:

1. method **String::split**;
2. method **String::replaceAll**;
3. method **String::matches**;

The first, **split**, invoked on a string, takes a regex and then splits the string into parts separated by substrings matching the regex — it returns an array of **String**s. Note that the regex specifies *separators*, not what we are looking for! For example,

```
"Łódź - 0.7; London - 8.8; Tokyo - 13.6".split("\\P{L}+")
```

will produce a three-element array containing the names of the three cities: the separator here is 'non-empty sequence of any non-letters' (note the capital 'P'). Note that there is a separator (sequence of non-letters) at the end of the input string. Therefore, there should be an empty string as the last element of the resulting array of strings. However, trailing empty strings are discarded. This does not apply to leading separator: in such a case, we *will* get an empty string as the first element of the array.

Regexes can also be ORed, as the following example illustrates: here, the separator is defined as non-empty sequence of white characters (between word boundaries) OR a punctuation mark surrounded, perhaps, by sequences of white characters:

Listing 8 GXP-Split/Splitting.java

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import static java.nio.charset.StandardCharsets.UTF_8;

public class Splitting {
    public static void main(String[] args) {
        try (
            BufferedReader br = Files.newBufferedReader(
                    Paths.get("Splitting.txt"), UTF_8)
        ) {
            String line;
            while ((line = br.readLine()) != null) {
                String[] array = line.split(
                        "(\\b\\s+\\b|\\b\\s*\\p{P}\\s*\\b)");
                System.out.print("|");
                for (String s:array) System.out.print(s+"|");
                System.out.println();
            }
        } catch(IOException e) {
            System.out.println(e.getMessage());
            System.exit(-1);
        }
    }
}
```

If the file contains

```
John,Mary    Charles    ;    Zoe
carrot   parsley:potato
```

then the program prints

```
|John|Mary|Charles|Zoe|
|carrot|parsley|potato|
```

The second method, **replaceAll**, takes a regex and a string and returns a string with all occurrences of substrings that match the regex replaced by the given string, e.g.,

```
"cat, caterpillar, tomcat, cat".replaceAll("\\bcat\\b", "dog")
```

will return `"dog, caterpillar, tomcat, dog"` (note that `cat` in `tomcat` will not be replaced because there is no word boundary before the letter 'c', and similarly for `caterpillar`). Invoking `s.replaceAll(regex, rep)` on a string `s` is equivalent to

`Pattern.compile(regex).matcher(s).replaceAll(rep)`.

In particular, we can refer to capture groups, specified in the regex string, in the replacement string (see sec. 3.4 on page 21).

The third method, **matches**, takes a regex and answers the question 'does the whole string match the regex'. For example

```
"Madagascar!".matches("\\p{L}+")
```

will return **false**, because the string contains a character which is not a letter, while

```
"Madagascar!".matches("\\p{L}+.*")
```

will return **true**.

3.3 *Patterns and Matchers*

Regexes, before being used, must be 'compiled' (in the case of the aforementioned methods of class **String**, this is done automatically). This is done by invoking the static method of class **Pattern** **compile(regex)**: it returns an object of type **Pattern** which represents the 'compiled' form of our regex (we can view it as some sort of a function). Having a compiled regex (pattern), we invoke on it **matcher(text)** where text is the text to be analyzed (as a string). It returns an object of type **Matcher** representing the result:

```
String text = "A text";
String reg  = "a regex";
Pattern pat = Pattern.compile(reg);
Matcher m = pat.matcher(text);
```

or, if we do not need to reuse the pattern:

```
String text = "A text";
String reg  = "a regex";
Matcher m = Pattern.compile(reg).matcher(text);
```

Very often, we want to use the same pattern many times, but for different texts — for example for each line of a text file. We then compile our regex once, and then we can call **matcher** on the same object of type **Pattern** and get matchers corresponding to subsequent lines. Or, we can even create a matcher once only, passing, for example, an empty string as a text, and then reset it for subsequent lines:

```
Matcher matcher = null;
try {
    matcher = Pattern.compile("a regex").matcher("");
} catch(PatternSyntaxException e) {
    System.err.println("Wrong pattern?");
    e.printStackTrace();
    System.exit(1);
}

try (
    BufferedReader br =
            Files.newBufferedReader(
                Paths.get("file.txt"), UTF_8)
) {
    String line;
    while ((line = br.readLine()) != null) {
        matcher.reset(line);
        // ... processing information
        // ... from a single line
    }
} catch (IOException e) {
    // ...
}
```

By invoking matcher's various methods, we can extract the information we need. Two basic methods which actually do the search are **matches** and **find**:

- **matches()** checks if the whole text matches the given regex (the one, on which we have called **matcher**) and returns **true** or **false**; note that it is not enough that the text contains some substrings matching the pattern — it must be the whole text;
- **find()** looks for *substrings* matching the regex (see examples below) and also returns **true** or **false**. It can be called several times; each time we invoke it, it starts searching from the location within the string after the previous match.

Matcher objects always remember the location where the last operation has finished. For successful invocation of **matches**, it will probably be the end of the input text, for an unsuccessful — the place where the matcher 'realized' that **matches** cannot succeed. Also for **find**, the location of the last successful match is remembered, so the next invocation of **find** will start looking for the next match. If we want to start from the beginning (for example, after calling **matches**), we can invoke `matcher.reset()`. The **reset** function is overloaded — a version taking a text makes the matcher 'forget' the old text and sets a new one that will be analyzed.

3.4 Capturing groups

Capturing group allows us to remember a part of text matching a pattern, so that we can use it later. A group is created when a part of a regex is enclosed in a pair of round parentheses. For example, the following regex

```
"(\\p{L}+)\\P{L}+(\\p{L}+)"
```

will match a string containing two words separated by a non-empty sequence of non-letters. However, both words will be remembered by the matcher as group number 1 and group number 2, and after a successful match we can get them by calling `matcher.get(1)` and `matcher.get(2)`, respectively. Groups may be nested and are numbered starting from 1, group number 0 being the whole substring matched, containing all the groups defined inside. Numbers are assigned according to the order in which opening parentheses of the groups are encountered — each group extends to the corresponding closing parenthesis. For example, in the program below, there will be three groups:

- one containing the first word, after, perhaps, some leading spaces, and followed by a sequence of any characters (which will not enter any group);
- then a group with two sequences of digits separated by a hyphen;
- then a group consisting of only the second of these two sequences of digits.

```
String reg = "\\s*(\\w+).*?(\\d+-(\\d+))";
  //             1         2     3
String text = "   Einstein Albert, 1879-1955";
Matcher m = Pattern.compile(reg).matcher(text);
System.out.println("Matches? " + m.matches());
System.out.println("# of groups " + m.groupCount());
for (int i = 1; i <= m.groupCount(); ++i)
    System.out.println(i + ": " + m.group(i));
```

The program prints

```
Matches? true
# of groups 3
1: Einstein
2: 1879-1955
3: 1955
```

Note the use of **groupCount** method — it reports the number of matched groups, *not* counting the special `group(0)`. Note also that in this regex we used the reluctant `.*?` — without the question mark, `.*` would consume the first three digits of the first number!

Instead of numbering groups, we can assign names to them; the syntax is `(?<name>`, where `name` is any unique name. We then refer to such groups by invoking `matcher.group("name")`. Both ways of accessing groups, by their number or by names, are illustrated in the program below:

Listing 9 GXX-RegGroups/RegGroups.java

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegGroups {
    public static void main (String[] args) {
        String str = "12-07-2014 xx 6-6-2010 yy 1-11-2011";
        String pat1 = "(\\d{1,2})-" +
                      "(\\d{1,2})-" +
                      "(\\d{4})";
        String pat2 = "(?<day>\\d{1,2})-" +
                      "(?<month>\\d{1,2})-" +
                      "(?<year>\\d{4})";

        Matcher m = Pattern.compile(pat1).matcher(str);
        System.out.println("Unnamed groups");
        while (m.find()) {
            System.out.println(m.groupCount()+" groups:");
            System.out.print("D: "+m.group(1)+", ");
            System.out.print("M: "+m.group(2)+", ");
            System.out.print("Y: "+m.group(3)+'\n');
        }

        m = Pattern.compile(pat2).matcher(str);
        System.out.println("Named groups");
        while (m.find()) {
            System.out.println(m.groupCount()+" groups:");
            System.out.print("D: "+m.group("day")+", ");
            System.out.print("M: "+m.group("month")+", ");
            System.out.print("Y: "+m.group("year")+'\n');
        }
    }
}
```

which prints

```
Unnamed groups
3 groups:
D: 12, M: 07, Y: 2014
3 groups:
D: 6, M: 6, Y: 2010
3 groups:
D: 1, M: 11, Y: 2011
Named groups
3 groups:
D: 12, M: 07, Y: 2014
3 groups:
D: 6, M: 6, Y: 2010
3 groups:
D: 1, M: 11, Y: 2011
```

Inside a regex, we can refer to the previously found groups using `\n` expressions, where `n` is the number of the group we are interested in (this is called *backreference*). In the following example we want to find fragments of a text enclosed in apostrophes or double quotes. We thus look for either an apostrophe or a double quote, but we have to ensure that the closing quote is of the same type as the opening one:

```
String reg = "(['\"])[^'\"]*\\1";
String text = "'abc' xx \"def\" yy \"ghi' zz";
Matcher m = Pattern.compile(reg).matcher(text);
while (m.find())
    System.out.println(m.group());
```

The program prints

```
'abc'
"def"
```

Note that the sequence `ghi` is enclosed in non-matching quotes, so, correctly, it has not been found.

Backreferences can also be used in replacement texts when using **String.replaceAll** method. There is also such a method in class **Matcher** and in fact

```
str.replaceAll(regex,text)
```

is equivalent to

```
Pattern.compile(regex).matcher(str).replaceAll(text)
```

For example, suppose we have a file with full names in the order first name and last name separated by at least one space, but what we want is to have them in reversed order: first the last name and then the first name separated by exactly one space. In the replacement text, we refer to groups using `$n` notation, where `n` is the group number, as in the example below:

```
String orig  = "John  Smith, Mary   Brown";
String modif = orig.replaceAll(
        "(\\p{L}+)\\s+(\\p{L}+)", "$2 $1");
System.out.println("Orig : " + orig + '\n' +
                   "Modif: " + modif);
```

The program prints

```
Orig : John  Smith, Mary   Brown
Modif: Smith John, Brown Mary
```

We could have used named groups instead; then in the replacement string we refer to groups using `${name}` notation:

```
String orig  = "John  Smith, Mary   Brown";
String modif = orig.replaceAll(
        "(?<first>\\p{L}+)\\s+(?<last>\\p{L}+)",
        "${last} ${first}");
System.out.println("Orig : " + orig + '\n' +
                   "Modif: " + modif);
```

with the result as before.

3.5 Option flags

There are several options which influence the process of compilation of a pattern. They can be specified in two ways: as an additional argument to **compile** or embedded directly into the regex. Options are defined in class **Pattern** as named static integer fields, which may be ORed, if we want several of them. If we choose to specify them embedded in our regex, we do it by including expression `(?<letters>)` where `<letters>` can be one or more letters denoting different options. For example, suppose we want to turn on the options DOTALL and MULTILINE; we can do it like this

```
import static java.util.regex.Pattern.*;
// ...
String regex = "...";
Pattern p = Pattern.compile(regex, DOTALL | MULTILINE);
```

or like this

```
String regex = "(?sm)...";
Pattern p = Pattern.compile(regex);
```

as the letter 's' denotes DOTALL ('s' because it's called 'single-line' in Perl) and 'm' stands for MULTILINE. Note that enabling options usually induces some performance penalty, so don't do it if it's not necessary.

Some of the most useful options are described below (there are others).

3.5.1 MULTILINE

This option enables the so called 'multi-line mode', which means that expressions `^` and `$` match just at the beginning of each a line and just at the and of each line, respectively. Normally, these symbols match only at the beginning and at the end of the whole input text; in multi-line mode those are still available as `\A` and `\Z`.
This option can also be enabled by embedding the flag `(?m)`.
For example, the following program

```
String s = "A 123\nD 456";
Matcher m1 = Pattern.compile("^\\w").matcher(s);
Matcher m2 = Pattern.compile("(?m)^\\w").matcher(s);
System.out.print("m1 : ");
```

```
while(m1.find())
    System.out.print(m1.group() + " ");
System.out.print("\nm2 : ");
while(m2.find())
    System.out.print(m2.group() + " ");
System.out.println();
```

will print

```
m1 : A
m2 : A D
```

because in the first case we only looked for a letter at the beginning of the entire input, while in the second case — at the beginning of each line separately.

3.5.2 *DOTALL*

This option changes the interpretation of the dot (.). By default, it denotes 'any character except new line', while with this option enabled a dot matches any character, *including* the new line.
This option can be enabled by embedding the flag `(?s)`.
For example, the following program

```
String s = "A 123\n456 B";
boolean b1 = Pattern.compile(
        "\\w.*\\w").matcher(s).matches();
boolean b2 = Pattern.compile(
        "(?s)\\w.*\\w").matcher(s).matches();
System.out.println("b1=" + b1 + "; b2=" + b2);
```

will print

```
b1=false; b2=true
```

because in the first case `.*` consumes everything up to the new line (but not any further) and there is no letter at the end, while in the second case everything, including the new line character, will be consumed by `.*` reaching the final letter 'B'.

3.5.3 *CASE_INSENSITIVE* and *UNICODE_CASE*

Enabling the CASE_INSENSITIVE option makes the lower- and uppercase letters indistinguishable. However, this works for Latin letters only. If it should apply to non-Latin letters too, we have to enable additionally the option UNICODE_CASE.
The options can be enabled by embedding the flags `(?i)` and `(?u)`, respectively, or we can enable both of them by `(?iu)`.
For example, the following program

```
String s = "PaRiS";
boolean b1 = Pattern.compile(
        "paris").matcher(s).matches();
boolean b2 = Pattern.compile(
        "paris", Pattern.CASE_INSENSITIVE)
        .matcher(s).matches();
System.out.println("b1=" + b1 + "; b2=" + b2);
```

will print

```
b1=false; b2=true
```

as in the second case we ignore the case of letters, so `PaRiS` and `paris` are considered equivalent.

3.5.4 *UNICODE_CHARACTER_CLASS*

When this option is enabled, some classes of characters, which normally apply only to ASCII characters, take into account all Unicode characters.
The options can be enabled by embedding the flags `(?U)` (note the *capital* U). Enabling UNICODE_CHARACTER_CLASS implies also UNICODE_CASE.
For example, the following program

```java
String s = "Żółć";
boolean b1 = Pattern.compile(
        "\\w+").matcher(s).matches();
boolean b2 = Pattern.compile(
        "\\w+", Pattern.UNICODE_CHARACTER_CLASS)
        .matcher(s).matches();
System.out.println("b1=" + b1 + "; b2=" + b2);
```

will print

```
b1=false; b2=true
```

as in the second case `\w` matches also non-ASCII letters.

3.6 Some examples

In the following example we look for names in file ***input.txt*** (in UTF-8 encoding): we make a rather naïve assumption that a string with the first letter in uppercase and the remaining characters in lowercase must be a name... Then we prints a list of names together with numbers of occurrences and lists of line numbers where a given name appeared. We use an auxiliary class **Name**

Listing 10 GXR-RegExNames/Name.java

```java
import java.util.ArrayList;

public class Name {
    private String name;
    private ArrayList<Integer> list;
    private int total = 0;

    public Name(String name, int lineNo) {
        this.name = name;
        list = new ArrayList<Integer>();
        list.add(lineNo);
        total = 1;
    }

```

```java
    public void addNum(int lineNo) {
        if (list.get(list.size()-1) != lineNo)
            list.add(lineNo);
        ++total;
    }

    @Override
    public String toString() {
        return name + " (" + total + ") in lines " + list;
    }
}
```

and the program might look like this:

Listing 11 GXR-RegExNames/RegEx.java

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Map;
import java.util.TreeMap;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.regex.PatternSyntaxException;
import static java.nio.charset.StandardCharsets.UTF_8;

/*  Reads file "input.txt" (in UTF-8) and selects
 *  names (strings starting with capital letter with
 *  other letters in lowercase. Then prints a list of
 *  names together with numbers of occurences and lists
 *  of line numbers where a given name appeared.
 *  Uses class Name.
*/
public class RegEx {

    private Map<String,Name> map;

    public static void main(String[] args) {
        new RegEx("input.txt");
        System.exit(0);
    }

    public RegEx(String fileName) {
        Path filein = Paths.get(fileName);
        if (!Files.exists(filein)     ||
            !Files.isReadable(filein) ||
             Files.isDirectory(filein)) {
```

```
            System.err.println("Invalid input file !!!");
            System.exit(1);
        }
//          File input = new File(filename);
//          BufferedReader br = null;
        map = new TreeMap<String,Name>();
        try (
            BufferedReader br =
                Files.newBufferedReader(filein, UTF_8)
            ) {
            String line, name,
                   patt = "\\b\\p{Lu}\\p{Ll}+\\b";
            Pattern pattern = null;
            try {
                pattern = Pattern.compile(patt);
            } catch(PatternSyntaxException e) {
                System.err.println("Wrong pattern");
                System.exit(1);
            }
            Matcher matcher = pattern.matcher("");

            int lineNo = 0;
            while ((line = br.readLine()) != null) {
                lineNo++;
                matcher.reset(line);
                if (matcher.find()) {
                    do {
                        name = matcher.group();
                        if (!map.containsKey(name))
                            map.put(name,
                                    new Name(name,lineNo));
                        else
                            map.get(name).addNum(lineNo);
                    } while (matcher.find());
                }
            }
        } catch(IOException e) {
            System.err.println("Something wrong - exiting");
            e.printStackTrace();
            System.exit(1);
        }

        for (Map.Entry<String,Name> e : map.entrySet())
            System.out.println(e.getValue());
    }
}
```

The program below asks for a regex and a text (in a loop); it then prints information about matches found:

Listing 12 GXU-Regexes/Regexes.java

```java
import java.io.Console;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Regexes {
    public static void main(String[] args){
        Console console = System.console();
        if (console == null) {
            System.err.println("Console unavailable");
            System.exit(1);
        }
        while (true) {
            String reg = console.readLine(
                    "%nRegex ('q' to quit) -> ");
            if ("q".equals(reg)) return;
            String inp = console.readLine(
                      "Input string        -> ");
            Pattern pattern = Pattern.compile(reg);
            Matcher matcher = pattern.matcher(inp);

            boolean found = false;
            while (matcher.find()) {
                found = true;
                console.format("Found '%s' at %d-%d.%n",
                    matcher.group(),
                    matcher.start(),
                    matcher.end());
            }
            if(!found){
                console.format("No match found.%n");
            }
        }
    }
}
```

and a similar program with a graphical interface:

Listing 13 GXW-REExplorer/REExplorer.java

```java
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.regex.PatternSyntaxException;
import javax.swing.AbstractAction;
import javax.swing.BorderFactory;
```

```java
import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;
import static javax.swing.JDialog.DISPOSE_ON_CLOSE;
import static javax.swing.JFrame.EXIT_ON_CLOSE;

public class REExplorer {

    public static void main(String[] args) {
        new REExplorer();
    }

    private REExplorer() {
        final JFrame f = new JFrame("RE Explorer");
        f.setDefaultCloseOperation(EXIT_ON_CLOSE);

        final JLabel labx = new JLabel("REGEX:");
        final JTextField rege = new JTextField(40);
        final JButton gobu  = new JButton("Go!");
        labx.setFont(new Font("Dialog",Font.PLAIN,18));
        gobu.setFont(new Font("Dialog",Font.PLAIN,18));
        rege.setFont(new Font("Dialog",Font.PLAIN,18));
        rege.setBorder(BorderFactory.
                        createEmptyBorder(5,5,5,5));
        JPanel pans = new JPanel();
        pans.setLayout(new FlowLayout());
        pans.add(labx);
        pans.add(rege);
        pans.add(gobu);

        final JTextArea text = new JTextArea(10,40);
        text.setFont(new Font("Dialog",Font.PLAIN,18));
        text.setBorder(BorderFactory.
                       createTitledBorder(
                       "Enter text to be searched below"));

        AbstractAction act = new AbstractAction() {
            @Override
            public void actionPerformed(ActionEvent e) {
                showText(getMatches(rege.getText(),
                                    text.getText()));
            }
        };
        rege.addActionListener(act);
```

```
        gobu.addActionListener(act);

        JScrollPane scroll = new JScrollPane(text,
            JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
            JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);

        JPanel panel = new JPanel();
        panel.setLayout(new BorderLayout());
        panel.add(scroll,BorderLayout.CENTER);
        panel.add(pans,BorderLayout.SOUTH);

        f.setContentPane(panel);

        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                f.pack();
                f.setLocationRelativeTo(null);
                f.setVisible(true);
            }
        });
    }

    private String getMatches(String regex, String text) {
        StringBuilder sb = new StringBuilder(200);
        String nL = System.getProperty("line.separator");

        sb.append("Pattern: \"" + regex + "\"" + nL +
                  "-- Text from here -----" + nL + text +
                  nL + "-- to here ------------" + nL + nL);
          // Compiling the pattern
        Pattern pattern = null;
        try {
            pattern = Pattern.compile(regex);
        } catch (PatternSyntaxException exc) {
            sb.append(
                  "Error: " + exc.getMessage() + nL);
            return sb.toString();
        }

        Matcher matcher = pattern.matcher(text);

          // does the whole text match the pattern?
        boolean match = matcher.matches();
        sb.append("matches() gives: " +
                  (match ? "YES" : " NO") + nL + nL);
          // groups (if any)
        if (match) {
            int gr = matcher.groupCount();
            sb.append(gr + " groups:" + nL);
            for (int i = 1; i <= gr; ++i)
```

```java
                sb.append("   " + i + ": " +
                          matcher.group(i) + nL);
            sb.append(nL);
        }

        matcher.reset();

          // looking for matches inside the text
        boolean found = matcher.find();
        if (!found)
            sb.append("find() didn\'t find anything" + nL);
        else
            do {
                sb.append("find() found \"" +
                    matcher.group() + "\" at " +
                    (matcher.start()+1)  + "-" +
                    matcher.end() + nL);
            } while(matcher.find());

        return sb.toString();
    }

    private void showText(String text) {
        final JDialog dg = new JDialog();
        dg.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
        JTextArea area = new JTextArea(20,45);
        area.setText(text);
        area.setBorder(BorderFactory.
                       createEmptyBorder(10,5,10,5));
        area.setFont(new Font("Monospaced",Font.PLAIN,20));
        area.setEditable(false);
        dg.add(new JScrollPane(area));
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                dg.pack();
                dg.setLocationRelativeTo(null);
                dg.setVisible(true);
            }
        });
    }
}
```

Another example:

Listing 14 GXY-RegFind/RegFind.java

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

```

```
public class RegFind {
    public static void main (String[] args) {

        spl("Joe : Mary::Jane", ":");
        spl("Joe : Mary:Jane", "\\s*:\\s*");

        mat("Joe : Mary:Jane", "[^:]*(:[^:]*){2}");
        mat("123 xxx ABCD", "\\.*\\d+[^A-Z]*[A-Z]{3,}");
        mat("Jane Crawford", "\\w+\\s+([A-Z]\\.)?\\s*\\w+");

        rep("a   b   c", "\\s+", " ");

        process("kot\\b",
                "Kot kot kotek");
        process("(kot)",
                "Kot kot kotek");
        process("\\b[A-Z][a-z]+\\b",
                "cat Dog hen Cow aHorse Z");
        process("^.*(kot).*(tek)",
                "Kot kot kotek");
        process("(a.*)(\\w).*",
                "a b c d c b x");
        process("(\\d{1,3}\\.){3}\\d{1,3}",
                " 1.1.1.2 12.12.34.231 234 xx 3.21.21.21 zz");
    }

    private static void process(String reg, String str) {
        Pattern p = Pattern.compile(reg);
        Matcher m = p.matcher(str);
        System.out.println("== match and find =======");
        System.out.println("STRING : " + str);
        System.out.println("REGEX  : " + reg);
        boolean gr = m.matches();
        System.out.println("Matches: " + gr);
        if (gr) {
            System.out.println("Groups : " + m.groupCount());
            for (int i = 1; i <= m.groupCount(); ++i) {
                System.out.println("Group " + i +
                           " = '" + m.group(i) + "'");
            }
        }
        m.reset();
        while (m.find()) {
            System.out.println(
                    "Found  " + ": " + m.group() +
                    " at " + m.start() + "-" + m.end());
        }
    }

    private static void spl(String str, String reg) {
```

```
        System.out.println("== split ================");
        System.out.println("STRING : " + str);
        System.out.println("REGEX  : " + reg);
        String[] s = str.split(reg);
        System.out.print(s.length + " terms:");
        for (int i = 0; i < s.length; ++i)
            System.out.print(" '" + s[i]+ "'");
        System.out.println();
    }

    private static void mat(String str, String reg) {
        System.out.println("== matches ==============");
        System.out.println("STRING : " + str);
        System.out.println("REGEX  : " + reg);
        System.out.println(str.matches(reg));
    }

    private static void rep(String str, String reg, String with) {
        System.out.println("== replace ==============");
        System.out.println("STRING : " + str);
        System.out.println("REGEX  : " + reg);
        System.out.println("WITH   : '" + with + "'");
        System.out.println(str.replaceAll(reg,with));
    }
}
```

Section 4

Abstract classes, interfaces, inner classes and lambdas

4.1 Abstract classes

Abstract classes are those which contain at least one abstract method, i.e., a method only declared but not defined. Both abstract classes and methods must be marked as such with the keyword **abstract**. It is not possible to create an object of an abstract class (as it is not fully implemented) — their *raison d'être* is to be extended. Extending (inheriting) class must provide definitions of methods declared but not defined in its abstract superclass, otherwise it will still be abstract itself. As a matter of fact, an abstract class *may* have all its methods implemented and still be declared as **abstract** — then it will be still impossible to create its objects and its only purpose is to serve as a base class for other classes.

Abstract classes provide a common interface (collection of methods) to a set of classes which can implement these methods in different ways. What is important is that we can declare references to abstract types while objects pointed to by these references are all of derived types (they cannot be objects of the base, abstract, class because such object cannot be even created).

Let us consider an example of an abstract class **Figure**, representing geometrical figures. Of course, if all we know is that something is a figure, we cannot say much about its area or perimeter: for this we should know its type more precisely. Therefore **getArea** and **getPerimeter** are declared abstract:

Listing 15 EPZ-AbsFig/Figure.java

```
abstract class Figure {

    abstract public double getArea();
    abstract public double getPerimeter();

    static public Figure getFigMaxArea(Figure[] figs) {
        double maxarea = 0;
        Figure maxfig  = null;
        for (Figure f : figs) {
            double area = f.getArea();
            if (area > maxarea) {
                maxarea = area;
                maxfig  = f;
            }
        }
        return maxfig;
    }

    @Override
    public String toString() {
        return " area: "
             + String.format("%6.3f",getArea())
```

```java
                 + "; perimeter: "
                 + String.format("%6.3f",getPerimeter());
    }
}
```

Class Circle inherits Figure and provides definitions of the abstract methods. It also overrides toString — note, how its implementation uses that from the base class (by using super).

Listing 16 EPZ-AbsFig/Circle.java

```java
public class Circle extends Figure {
    private double r;

    public Circle(double r) {
        this.r = r;
    }

    @Override
    public double getArea() {
        return Math.PI*r*r;
    }
    @Override
    public double getPerimeter() {
        return 2*Math.PI*r;
    }
    @Override
    public String toString() {
        return "Circle ( r=" + r + " )"
                + super.toString();
    }
}
```

as does Rectangle

Listing 17 EPZ-AbsFig/Rectangle.java

```java
public class Rectangle extends Figure {
    private double a, b;

    public Rectangle(double a, double b) {
        this.a = a;
        this.b = b;
    }

    @Override
    public double getArea() {
        return a*b;
```

```java
    }

    @Override
    public double getPerimeter() {
        return 2*(a+b);
    }

    @Override
    public String toString() {
        return "Rectangle (a=" + a + " b=" + b + ")"
               + super.toString();
    }
}
```

and we can now check our classes in **main**

Listing 18 EPZ-AbsFig/Main.java

```java
import java.util.Locale;

public class Main {
    public static void main(String[] args) {
          // to have decimal point instead of a comma...
        Locale.setDefault(Locale.US);

        Figure[] figs = {
            new Circle(2),      new Rectangle(9,1),
            new Rectangle(4,3), new Circle(4)
        };

        Figure fig = Figure.getFigMaxArea(figs);
        System.out.println("\nLargest area: \n" + fig);
    }
}
```

which prints

```
Largest area:
Circle     (    r=4.0    ) area: 50.265; perimeter: 25.133
```

4.2 Interfaces

Interfaces are, in a sense, “pure” abstract classes — they only declare one or more methods, but do not implement them (but see below). All methods are, by definition, **public**, even if not declared as such. Also, unimplemented methods are *not* declared as **abstract** (although they are abstract).
Interfaces can also *define*:

- final, static constants;
- default methods, marked with the keyword **default**;

- *private* methods, used internally by its default methods — these private methods, of course, are not visible for the user and do not belong to the 'contract'.
- *static* methods — which can be public and are accessible for the users.

Interfaces with *only one* abstract method (but, perhaps, with some default methods and static member functions already implemented) are called **functional interfaces** and play a special rôle in Java (more about it later — see 10, p. 118).

What's important is the fact that any class can *extend* (directly inherit from) only one class, but may *implement* **any number of interfaces.** Also records (see see Sec. **??**, p. **??**) can implement interfaces.

Interfaces define a "contract" — if we know that a class implements a given interface, we know that all its abstract methods must have been implemented somehow and hence it is safe to call these methods on object of such a class. We can also declare references of interface type — they can refer to objects of *any* classes implementing the interface, even if their types belong to completely separate subtrees of the class hierarchy.

Let us consider the following example. We define an interface, **IMyStack**, which defines a 'contract' of a stack. What is a stack? It is something that has **pop**, **push** and **empty** methods. We can then create concrete classes that implement this interface in different ways — for example using an array or using the singly-linked list (**MyStackArr** and **MyStackList**, respectively, in the example below). Note the function **testStack**. Its single argument is declared as 'something' of type **IMyStack** — there will be no objects of this type because it is not a concrete class but an interface. However, this declaration means 'anything what implements **IMyStack** will be accepted'. We can then safely call **push**, **pop** and **empty** on the object passed to the function because we know that they have to be implemented somehow. For example, the program below:

Listing 19 ELJ-InterStack/MyStacks.java

```java
public class MyStacks {
    public static void main(String[] args) {
        testStack(new MyStackArr(10));
        testStack(new MyStackList());
    }

    public static void testStack(IMyStack stack) {
        stack.push(5);
        stack.push(4);
        stack.push(3);
        stack.push(2);
        while (!stack.empty()) {
            System.out.print(stack.pop() + " ");
        }
        System.out.println();
    }
}


interface IMyStack {
    int pop();          // public automatically
    void push(int i);
```

```
    boolean empty();
}

class MyStackArr implements IMyStack {
    int[] arr;
    int   top;
    public MyStackArr(int size) {
        arr = new int[size];
        top = 0;
    }
    public int pop() {
        return arr[--top];
    }
    public void push(int i) {
        arr[top++] = i;
    }
    public boolean empty() {
        return top == 0;
    }
}

class MyStackList implements IMyStack {
    private static class Node {
        int data;
        Node next;
        Node(int d, Node n) {
            data = d;
            next = n;
        }
        Node(int d) {
            this(d, null);
        }
    }
    private Node head = null;
    public int pop() {
        int d = head.data;
        head = head.next;
        return d;
    }
    public void push(int d) {
        head = new Node(d, head);
    }
    public boolean empty() {
        return head == null;
    }
}
```

prints

```
2 3 4 5
```

```
 2 3 4 5
```

Let us now consider another example. We define an interface that has only one abstract method: **lt** (*less than*). All other methods (**gt** — *greater than*, **ge** — *greater or equal than*, etc.) have default implementation expressed, directly or indirectly, by this one abstract method. Therefore, to implement this interface we only have to implement **lt** — all other methods will then work automatically!

Listing 20 ELS-DefIntSimple/MyCompar.java

```java
public interface MyCompar {
      // abstract
    boolean lt(int lhs, int rhs);

      // implemented directly in terms of the abstract
    default boolean gt(int lhs, int rhs) {
        return lt(rhs,lhs);
    }
    default boolean ge(int lhs, int rhs) {
        return !lt(lhs,rhs);
    }
    default boolean le(int lhs, int rhs) {
        return !lt(rhs,lhs);
    }
    default boolean eq(int lhs, int rhs) {
        return !lt(lhs,rhs) && !lt(rhs,lhs);
    }
      // implemented indirectly in terms of the abstract
    default boolean ne(int lhs, int rhs) {
        return !eq(lhs,rhs);
    }
}
```

Then we define three classes implementing this interface: one compares integers by their values

Listing 21 ELS-DefIntSimple/CompVal.java

```java
public class CompVal implements MyCompar {
    @Override
    public boolean lt(int lhs, int rhs) {
        return lhs < rhs;
    }
}
```

the second by sum of digits

Listing 22 ELS-DefIntSimple/CompDigits.java

```java
public class CompDigits implements MyCompar {
    @Override
    public boolean lt(int lhs, int rhs) {
        return sumOfDigs(lhs) < sumOfDigs(rhs);
    }
    private static int sumOfDigs(int n) {
        int sum = 0;
        n = n < 0 ? -n : n;
        while (n != 0) {
            sum += n % 10;
            n /= 10;
        }
        return sum;
    }
}
```

and the third by value, but reversed

Listing 23 ELS-DefIntSimple/CompValRev.java

```java
public class CompValRev implements MyCompar {
    @Override
    public boolean lt(int lhs, int rhs) {
        return lhs > rhs;
    }
}
```

Now in **Main** we can use all three implementations; each of them has full set of all methods implemented although they override only one method:

Listing 24 ELS-DefIntSimple/Main.java

```java
public class Main {
    public static void main (String[] args) {

        MyCompar cmpVal = new CompVal();
        MyCompar cmpSum = new CompDigits();
        MyCompar cmpVaR = new CompValRev();

        compare("cmpVal - BY VALUE",cmpVal,
                10,2, 3,12, 5,22);
        compare("cmpSum - BY SUM OF DIGITS",cmpSum,
                10,2, 3,12, 5,22);
        compare("cmpVaR - BY VALUE REVERSED",cmpVaR,
                10,2, 3,12, 5,22);
    }

```

```
    private static void compare(String message,
                    MyCompar cmp, int... pairs) {
        System.out.println("\n========= " + message);
        for (int k = 0; k < pairs.length; k += 2) {
            int a = pairs[k], b = pairs[k+1];
            System.out.println("** (" + a + "," + b + "): "+
                    "lt->" + cmp.lt(a,b) + ", " +
                    "le->" + cmp.le(a,b) + "\n" +
                "    gt->" + cmp.gt(a,b) + ", " +
                    "ge->" + cmp.ge(a,b) + ", " +
                    "eq->" + cmp.eq(a,b) + ", " +
                    "ne->" + cmp.ne(a,b) );
        }
    }
}
```

The program prints

```
========= cmpVal - BY VALUE
** (10,2): lt->false, le->false
    gt->true, ge->true, eq->false, ne->true
** (3,12): lt->true, le->true
    gt->false, ge->false, eq->false, ne->true
** (5,22): lt->true, le->true
    gt->false, ge->false, eq->false, ne->true

========= cmpSum - BY SUM OF DIGITS
** (10,2): lt->true, le->true
    gt->false, ge->false, eq->false, ne->true
** (3,12): lt->false, le->true
    gt->false, ge->true, eq->true, ne->false
** (5,22): lt->false, le->false
    gt->true, ge->true, eq->false, ne->true

========= cmpVaR - BY VALUE REVERSED
** (10,2): lt->true, le->true
    gt->false, ge->false, eq->false, ne->true
** (3,12): lt->false, le->false
    gt->true, ge->true, eq->false, ne->true
** (5,22): lt->false, le->false
    gt->true, ge->true, eq->false, ne->true
```

Let us now consider an example of a functional interface with abstract method. In the program below, we define **Fun** interface which declares one abstract method (**apply**) but is additionally equipped with a static function (**transformArray**) taking one array of doubles and returning another whose elements are the results of applying a function to elemensts of the input array. The function takes, as its second argument, the reference to an object of any class implementing the **Fun** interface and therefore providing a definition of the **apply** method:

Listing 25 ELL-InterFun/InterFun.java

```java
public class InterFun {
    public static void main(String[] args) {
        double[] a = {-Math.PI/6, Math.PI/6, 7*Math.PI/6};
        double[] r = Fun.transformArray(a, new Sin());
        double[] s = Fun.transformArray(r, new MultBy2());
        System.out.println(java.util.Arrays.toString(s));
    }

}

@FunctionalInterface
interface Fun {
    double apply(double d);

    static double[] transformArray(double[] arr, Fun f) {
        double[] res = new double[arr.length];
        for (int i = 0; i < arr.length; ++i)
            res[i] = f.apply(arr[i]);
        return res;
    }
}

class Sin implements Fun {
    @Override
    public double apply(double d) { return Math.sin(d); }
}

class MultBy2 implements Fun {
    @Override
    public double apply(double d) { return 2*d; }
}
```

Here, we first apply the **sin** function to an array of doubles, and then we apply multiplication by 2 to the resulting array. The program prints

```
[-0.9999999999999999, 0.9999999999999999, -0.9999999999999994]
```

Many important interfaces are already defined in the standard library. For example, **Comparable** declares one abstract method

```java
int compareTo(Object)
```

Given objects `a` and `b`, `a.compareTo(b)` returns something negative if `a` is "smaller" than `b`, something positive if `b` is smaller, and 0 if they are considered equal. When declaring that a class implements this interface for objects of type **T**, we should always indicate this type (in angle brackets) — then we can declare the type of the argument as **T** and not **Object** (and no casting is required). Classes implementing **Comparable** are said to be equipped with **natural order**. .

In the (abstract) class **Figure** below , we declare and define **compareTo** (remember that all methods declared in an interface are by definition public, so overriding them we must not forget about **public**) specifically for **Figure**s and therefore we declare the class as implementing **Comparable<Figure>**:

Listing 26 EQA-AbstractFigs/Figure.java

```java
abstract class Figure implements Comparable<Figure> {

    abstract public double getArea();
    abstract public double getPerimeter();

    static public Figure getFigMaxArea(Figure[] figs) {
        double maxarea = 0;
        Figure maxfig  = null;
        for (Figure f : figs) {
            double area = f.getArea();
            if (area > maxarea) {
                maxarea = area;
                maxfig  = f;
            }
        }
        return maxfig;
    }

    @Override
    public String toString() {
        return " area: "
             + String.format("%6.3f",getArea())
             + "; perimeter: "
             + String.format("%6.3f",getPerimeter());
    }

    @Override
    public int compareTo(Figure f) {
        double diff = getPerimeter() - f.getPerimeter();
        if      (diff < 0) return -1;
        else if (diff > 0) return +1;
        else               return 0;
    }
}
```

Then we can define implementing classes: **Circle**

Listing 27 EQA-AbstractFigs/Circle.java

```java
public class Circle extends Figure {
    private double r;

    public Circle(double r) {
        this.r = r;
    }
```

```java

    @Override
    public double getArea() {
        return Math.PI*r*r;
    }
    @Override
    public double getPerimeter() {
        return 2*Math.PI*r;
    }
    @Override
    public String toString() {
        return "Circle    (    r=" + r + "    )"
                + super.toString();
    }
}
```

and **Rectangle**

Listing 28 EQA-AbstractFigs/Rectangle.java

```java
public class Rectangle extends Figure {
    private double a, b;

    public Rectangle(double a, double b) {
        this.a = a;
        this.b = b;
    }

    @Override
    public double getArea() {
        return a*b;
    }
    @Override
    public double getPerimeter() {
        return 2*(a+b);
    }

    @Override
    public String toString() {
        return "Rectangle (a=" + a + " b=" + b + ")"
              + super.toString();
    }
}
```

Then in **main** we can use **sort** and Java will know how to compare figures: it will just call **compareTo** (it knows it is there, because the compiler can see that **Figure** implements **Comparable**, otherwise the program would not even compile):

Listing 29 EQA-AbstractFigs/Main.java

```java
import java.util.Arrays;
import java.util.Locale;

public class Main {
    public static void main(String[] args) {
          // to have decimal point instead of a comma...
        Locale.setDefault(Locale.US);

        Figure[] figs = {
            new Circle(2),      new Rectangle(9,1),
            new Rectangle(4,3), new Circle(4)
        };

        Figure fig = Figure.getFigMaxArea(figs);
        System.out.println("\nLargest area: \n" + fig);

        Arrays.sort(figs);
        System.out.println("\nSorted by circumference:");
        for (Figure f : figs)
            System.out.println(f);
    }
}
```

The program prints

```
Largest area:
Circle    (   r=4.0   ) area: 50.265; perimeter: 25.133

Sorted by circumference:
Circle    (   r=2.0   ) area: 12.566; perimeter: 12.566
Rectangle (a=4.0 b=3.0) area: 12.000; perimeter: 14.000
Rectangle (a=9.0 b=1.0) area:  9.000; perimeter: 20.000
Circle    (   r=4.0   ) area: 50.265; perimeter: 25.133
```

Any class can implement only one natural order. However it may happen that we want to use (e.g., for sorting) different criteria. We then can create an object which will be used as a comparator even if a natural order exist: it will be an object of a type implementing **Comparator** with only one abstract method

```java
int compare(Object,Object)
```

Then, if ob is an object of this class and a and b are to be compared, `ob.compare(a,b)` should return something negative if `a` is "smaller" than `b`, something positive if `b` is smaller, and 0 if they are considered equal. As with **Comparable**, when declaring a class implementing **Comparator**, we should always indicate the type, call it **T**, of objects it is supposed to be able to compare (in angle brackets) — then we can declare the type of the arguments of **compare** as **T** and not **Object** (and no casting is required).

Let us consider an example: class **Person** has a natural order (as it implements **Comparable**)

Listing 30 ELP-Comps2/Person.java

```java
public class Person implements Comparable<Person> {

    final static int currentYear =
            java.util.Calendar.getInstance().
                get(java.util.Calendar.YEAR);

    String name;
    int    birthYear;
    int    height;

    Person(String n, int y, int h) {
        name      = n;
        birthYear = y;
        height    = h;
    }

    /**
     * natural order: by name, then age, then height
     */
    @Override
    public int compareTo(Person o) {
        int k = name.compareToIgnoreCase(o.name);
        if ( k != 0 ) return k;
        k = o.birthYear - birthYear;
        if ( k != 0 ) return k;
        return height - o.height;
    }

    @Override
    public String toString() {
        return name + "(" + (currentYear-birthYear) +
                                    "/" + height + ")";
    }
}
```

In order to be able to compare persons in different ways, not necessarily determined by the natural order, we define two different classes representing comparators of Persons

Listing 31 ELP-Comps2/Comparators.java

```java
import java.util.Comparator;

/**
 * Comparator 1: by height, then age , then name
 */
class Comp1 implements Comparator<Person> {
    @Override
    public int compare(Person o1, Person o2) {
```

```java
        int k = o1.height - o2.height;
        if ( k != 0 ) return k;
        k = o2.birthYear - o1.birthYear;
        if ( k != 0 ) return k;
        return o1.name.compareToIgnoreCase(o2.name);
    }
}

/**
 * Comparator 2: by age, then by name, then by height
 */
class Comp2 implements Comparator<Person> {
    @Override
    public int compare(Person o1, Person o2) {
        int k = o2.birthYear - o1.birthYear;
        if ( k != 0 ) return k;
        k = o1.name.compareToIgnoreCase(o2.name);
        if ( k != 0 ) return k;
        return o1.height-o2.height;
    }
}
```

which now can be used, for example, to sort arrays or lists of **Persons**:

Listing 32 ELP-Comps2/Main.java

```java
import java.util.Arrays;
import java.util.Comparator;

public class Main {
    public static void main(String[] args) {
        new Main();
    }

    Main() {
        Person[] list = {
          new Person("K",1980,165),
          new Person("B",1986,171),
          new Person("K",1980,168),
          new Person("H",1980,171),
          new Person("M",1980,171),
          new Person("K",1980,169),
          new Person("B",1979,171),
          new Person("G",1975,171)
        };

          // natural
        Arrays.sort(list);
        writeL(list, "Natural: name, age, height");
```

```java

        // comparator Comp1
        Arrays.sort(list, new Comp1());
        writeL(list, "Comp1:   height, age, name");

        // comparator Comp2
        Comparator<Person> comp2 = new Comp2();
        Arrays.sort(list, comp2);
        writeL(list, "Comp2:   age, name, height");

        // anonymous comparator
        Arrays.sort(list, new Comparator<Person>() {
            @Override
            public int compare(Person p, Person q) {
                int k = p.name.compareToIgnoreCase(q.name);
                if ( k != 0 ) return k;
                k = p.height - q.height;
                if ( k != 0 ) return k;
                return q.birthYear - p.birthYear;
            }
        });
        writeL(list, "Anonym:  name, height, age");

        // lambda
        Arrays.sort(list, (f,s) -> f.height-s.height);
        writeL(list, "Lambda:  name, height, age");
    }

    static void writeL(Person[] list, String header) {
        System.out.println('\n'+header);
        for (Person p : list) System.out.print(p+" ");
        System.out.println();
    }
}
```

The program prints

```
Natural: name, age, height
B(33/171) B(40/171) G(44/171) H(39/171)
K(39/165) K(39/168) K(39/169) M(39/171)

Comp1:   height, age, name
K(39/165) K(39/168) K(39/169) B(33/171)
H(39/171) M(39/171) B(40/171) G(44/171)

Comp2:   age, name, height
B(33/171) H(39/171) K(39/165) K(39/168)
K(39/169) M(39/171) B(40/171) G(44/171)

Anonym:  name, height, age
```

```
B(33/171) B(40/171) G(44/171) H(39/171)
K(39/165) K(39/168) K(39/169) M(39/171)

Lambda:  name, height, age
K(39/165) K(39/168) K(39/169) B(33/171)
B(40/171) G(44/171) H(39/171) M(39/171)
```

Let us consider another, but similar, example. We again create a class representing persons (equipped with a natural order)

Listing 33 ELM-Comps1/Person.java

```java
public class Person implements Comparable<Person> {

    private String name;
    private int    year;

    public Person(String name, int year) {
        this.name = name;
        this.year = year;
    }

    @Override
    public int compareTo(Person other) {
        int diff = year - other.year;
        if (diff != 0) return diff;
        else           return name.compareTo(other.name);
    }

    public String getName() { return name; }
    public int getYear()    { return year; }

    @Override
    public String toString() {
        return name + "(" + year + ")";
    }

    static void show(Person[] persons, String message) {
        System.out.println(message);
        for (Person person : persons)
            System.out.print(person + " ");
        System.out.println("\n");
    }
}
```

Now, we create only one class, **CompPerson**, objects of which can be used as comparators of **Person**s. The class contains one field, set by the constructor. In the example below it is an enumerator, but equally well it could have been an integer. When creating an object of this class, we will pass to the constructor information about the way we want persons to be compared — therefore, our class is in a way 'configurable'

Listing 34 ELM-Comps1/CompPerson.java

```java
import java.util.Comparator;

public class CompPerson implements Comparator<Person> {

    public static enum Comp { BY_NAME,    BY_YEAR,
                              BY_NAMERev, BY_YEARRev };
    private Comp comp;

    public CompPerson(Comp comp) {
        this.comp = comp;
    }

    @Override
    public int compare(Person p1, Person p2) {

        int rYear = p1.getYear() - p2.getYear();
        int rName = p1.getName().compareTo(p2.getName());

        int result = 0;

        switch (comp) {
            case BY_NAME:
                result = rName != 0 ?  rName : rYear; break;
            case BY_NAMERev:
                result = rName != 0 ? -rName : rYear; break;
            case BY_YEAR:
                result = rYear != 0 ?  rYear : rName; break;
            case BY_YEARRev:
                result = rYear != 0 ? -rYear : rName; break;
        }
        return result;
    }
}
```

Then we can use objects of this class as a comparator to sort collections (or arrays) of Persons in various ways:

Listing 35 ELM-Comps1/Main.java

```java
import java.util.Arrays;

public class Main {

    public static void main(String[] args) {
        Person[] persons = {
                              new Person("Mary",1990),
                              new Person("Joan",1992),
                              new Person("Suzy",1992),
```

```
                               new Person("Beth",1992),
                               new Person("Suzy",1980),
                               new Person("Katy",1982),
                           };
        Person.show(persons,"At the beginning:");

        Arrays.sort(persons);
        Person.show(persons,"Natural order: " +
                      "by year, then by name");

        Arrays.sort(persons,
            new CompPerson(CompPerson.Comp.BY_NAME));
        Person.show(persons,"Order BY_NAME: " +
                      "by name, then by year");

        Arrays.sort(persons,
            new CompPerson(CompPerson.Comp.BY_NAMERev));
        Person.show(persons,"Order BY_NAMERev: " +
                "by name reversed, then by year");

        Arrays.sort(persons,
            new CompPerson(CompPerson.Comp.BY_YEAR));
        Person.show(persons,"Order BY_YEAR: " +
                      "by year, then by name");

        Arrays.sort(persons,
            new CompPerson(CompPerson.Comp.BY_YEARRev));
        Person.show(persons,"Order BY_YEARRev: " +
                "by year reversed, then by name");

        Arrays.sort(persons,
                   (f,s) -> s.getYear() - f.getYear());
        Person.show(persons,"Order by lambda : " +
                "by year ");
    }
}
```

The program prints

```
At the beginning:
Mary(1990) Joan(1992) Suzy(1992) Beth(1992) Suzy(1980) Katy(1982)

Natural order: by year, then by name
Suzy(1980) Katy(1982) Mary(1990) Beth(1992) Joan(1992) Suzy(1992)

Order BY_NAME: by name, then by year
Beth(1992) Joan(1992) Katy(1982) Mary(1990) Suzy(1980) Suzy(1992)

Order BY_NAMERev: by name reversed, then by year
Suzy(1980) Suzy(1992) Mary(1990) Katy(1982) Joan(1992) Beth(1992)
```

```
Order BY_YEAR: by year, then by name
Suzy(1980) Katy(1982) Mary(1990) Beth(1992) Joan(1992) Suzy(1992)

Order BY_YEARRev: by year reversed, then by name
Beth(1992) Joan(1992) Suzy(1992) Mary(1990) Katy(1982) Suzy(1980)

Order by lambda : by year
Beth(1992) Joan(1992) Suzy(1992) Mary(1990) Katy(1982) Suzy(1980)
```

4.3 Inner and anonymous classes

It is possible to define a class inside another class — we then say that it is an **inner class**; the class in which an inner class is defined is its **outer** (or **surrounding**) class. An inner class may be defined as static or non-static.

4.3.1 Non-static inner classes

Let us consider *non-static* inner classes first. Objects of an inner class cannot be created independently of objects of its surrounding class: they always contain a reference to a "parent" object of the outer class — this reference is accessible under the name Outer.this, where **Outer** is the name of the surrounding class. Therefore, such objects may only be created inside methods of the outer class (and **this** will be equivalent to Outer.this inside the object created) or by invoking **new Inner(...)** on an object of the outer class.

What is also important is the fact that both classes, an inner class and its surrounding class, are "friends", i.e., all members, even private, of one of them are directly accessible by methods of the other, what is illustrated in the example below:

Listing 36 ELH-OutInn/OutInn.java

```java
class Outer {
    private String sOut;
    Outer(String s) { sOut = s; }

    class Inner {
        private String sInn;
        Inner(String s) { sInn = s; }
        @Override
        public String toString() {
            return "Inner-" + sInn + " parent " +
                   "Outer-" + Outer.this.sOut; // <- syntax!
                                               // note that
                                               // sOut is
                                               // private!
        }
    }

    public Inner getInner(String i) {
        Inner inn = new Inner(i);
```

```java
        System.out.println("Creating inner " + inn.sInn);
        return inn;
    }

    @Override
    public String toString() { return "Outer-" + sOut; }
}

public class OutInn {
    public static void main (String[] args) {
        Outer out1 = new Outer("out1");
        Outer.Inner inn1 = out1.getInner("inn1");
        Outer.Inner inn2 = out1.new Inner("inn2");
        System.out.println(out1);
        System.out.println(inn1);
        System.out.println(inn2);
        System.out.println(out1.getClass().getName());
        System.out.println(inn1.getClass().getName());
        System.out.println(inn2.getClass().getName());
    }
}
```

which prints

```
Creating inner inn1
Outer-out1
Inner-inn1 parent Outer-out1
Inner-inn2 parent Outer-out1
Outer
Outer$Inner
Outer$Inner
```

4.3.2 Static inner classes

An inner class may also be declared as **static**. It is still a "friend" of the outer class, but there is no Outer.this inside the object of the inner class; therefore, objects of the inner class may exist independently of any objects of the outer class.

In the example below class **MyStack** represents a stack (of integers) implemented as a singly linked list. We need a class representing individual nodes of the list, but the user of the stack doesn't need to know about its existence; therefore we define **Node** as a private static inner class inside **MyStack**:

Listing 37 GMC-StackSimple/MyStack.java

```java
public class MyStack {
      // static inner class
    private static class Node {
        int  data;
        Node next;
        Node(int d, Node n) {
```

```
                data = d;
                next = n;
            }
        }

        private Node top;

        public MyStack() {
            top = null;
        }
        public void push(int d) {
            top = new Node(d, top);
        }
        public int pop() {
            int d = top.data;
            top = top.next;
            return d;
        }
        public boolean empty() {
            return top == null;
        }
    }
```

and the user uses only objects of type **MyStack**; class **Node** is just an “implementation detail”:

Listing 38 GMC-StackSimple/StackSimple.java

```
public class StackSimple {
    public static void main (String[] args) {
        MyStack stInt = new MyStack();
        for (int i = 5; i > 0; --i)
            stInt.push(i);
        while (!stInt.empty())
            System.out.print(stInt.pop() + " ");
        System.out.println();
    }
}
```

4.3.3 Anonymous classes

Sometimes we have to create just one object of a type which implements an interface, or extends an abstract class, or behaves as an object of an existing concrete class but with one or a few methods overridden: in such situation one can use object of an **anonymous class**. The syntax is illustrated in the example below: after **new** we specify a class (concrete or abstract) that we want our anonymous class to extend, or an interface that we want it to implement. In the first case we can also pass arguments to a constructor; in any case round parentheses are obligatory. Then, in curly braces, we write an implementation (normally, we just override one or more methods). The compiler will then create an anonymous class and return an object of this type:

Listing 39 ELI-Anon/Anon.java

```java
interface BiIntOperator {
    int apply(int i, int j);
}

class AddAndMult implements BiIntOperator {
    int seed;
    AddAndMult(int seed) { this.seed = seed; }
    AddAndMult()         { this(1);          }
    @Override
    public int apply(int i, int j) { return seed*(i + j); }
}

class MultAndAdd implements BiIntOperator {
    int seed;
    MultAndAdd(int seed) { this.seed = seed; }
    MultAndAdd()         { this(1);          }
    @Override
    public int apply(int i, int j) { return seed + i*j; }
}

public class Anon {
    public static void main(String[] args) {
        BiIntOperator[] opers = {
              // objects of concrete classes
              // implementing an interface
            new AddAndMult(2),
            new MultAndAdd(5),
              // object of anonymous class
              // implementing an interface
            new BiIntOperator() {
                @Override
                public int apply(int i, int j) {
                    return i*i + j*j;
                }
            },
              // object of anonymous class
              // extending a 'normal' class
            new MultAndAdd(3) {
                @Override
                public int apply(int i, int j) {
                    return seed*(i*i + j*j);
                }
            }
        };
        int a = 1, b = 2;
```

```
        for (BiIntOperator op : opers)
            System.out.print(op.apply(a,b) + " ");
        System.out.println();
    }
}
```

(the program prints 6 7 5 15).
Of course, we cannot create another object of such an anonymous class, because it doesn't even have a name. For the same reason, anonymous classes cannot define any constructors.

Another example illustrating abstract and anonymous classes: we define an abstract class **Animal**

Listing 40 EQE-AbstractAnimals/Animal.java

```
public abstract class Animal {
    String name;
    double weight;

    public Animal(String name, double weight) {
        this.name   = name;
        this.weight = weight;
    }

    abstract public String speak();

    @Override
    public String toString() {
        return name + "(" + weight + ") - " + speak();
    }
}
```

and then we use it in a program creating various animals

Listing 41 EQE-AbstractAnimals/Main.java

```
public class Main {
    public static void main(String[] args) {

        Animal max =
            new Animal("Max", 15) {
                @Override
                public String speak() {
                    return "bow-wow";
                }
            };

        Animal[] animals =
```

```
            {
                max,
                new Animal("Batty", 3.5) {
                    @Override
                    public String speak() {
                        return "miaou-miaou";
                    }
                }
            };

        for (Animal a : animals)
            System.out.println(a);
    }
}
```

Program prints

```
Max(15.0) - bow-wow
Batty(3.5) - miaou-miaou
```

Important: When we define an anonymous class, local variables visible in the current scope will be accessible (and can be used) inside the body of methods of the anonymous class being created. The only condition is that these local variables are

- declared as **final** (and, of course, initialized), or
- **effectively final**, i.e., they are defined and initialized and then not modified.

4.4 Lambdas

Instead of passing an object of a concrete or anonymous class implementing an interface, one can often provide just a simple expression describing a functionality of a method which is supposed to be implemented (we call such an expression a **lambda**). For this to be possible, the compiler has to know which method it is and from which interface. Therefore, we can use a lambda only when it is clear from the context implementation of which interface is expected. In order for the compiler to know what method is to be overridden, there must be only one abstract method in the interface involved — such interfaces, with only one abstract method, are called **functional interfaces**.

A lambda expression itself is composed of three parts:

- List of parameters, as in declaration of a 'normal' function. However, usually the types of parameters need not be specified, because the compiler is able to deduce them from the context. If there is only one parameter and type is omitted, parentheses are optional. If the list of parameters is empty, we just write empty parentheses.
- The "arrow" token: $->$.
- A function body enclosed in curly braces. If the body contains just one expression (i.e., something that has a value), there are no braces, no semicolon at the and no **return** statement: the value of the expression will be evaluated and returned — return type will be deduced as the type of this value. Braces may be also omitted if the body consists of one statement which does not have any value — return type **void** will then be assumed.

Examples:

```
(int x, int y) -> x + y
(x, y) -> x*y < 0
() -> Math.random()
e -> System.out.println(e)
```

In these examples we assume that the compiler will be able to infer all necessary types and deduce what functional interface is to be implemented.

The example below illustrates both cases: when the body of a lambda is a single expression (no semicolon, no **return**, no braces) and when it is implemented as a compound statement (with braces, **return** and semicolons after statements, as usually). Note that the compiler expects, as the second argument of the **sort** function 'something that implements the **Comparator** interface'. As what is to be sorted is an array of **Persons**, the compiler knows also that it should be in fact **Comparator<Person>**, so the type of p1 and p2 is deduced to be **Person**:

Listing 42 EMD-InterF/InterF.java

```java
import java.util.Arrays;

class Person {

    private String name;
    private int    year;

    public Person(String name, int year) {
        this.name = name;
        this.year = year;
    }

    public String getName() { return name; }
    public int getYear()    { return year; }

    @Override
    public String toString() {
        return name + "(" + year + ")";
    }

    static void show(Person[] persons, String message) {
        System.out.println(message);
        for (Person person : persons)
            System.out.print(person + " ");
        System.out.println();
    }
}

public class InterF {

    public static void main(String[] args) {
        Person[] persons =
            { new Person("Mary",1990),
```

```
                new Person("Joan",1992),
                new Person("Suzy",1992),
                new Person("Beth",1992),
                new Person("Suzy",1980),
                new Person("Katy",1982), };
        Person.show(persons,"At the beginning:");

          // lambda as a single expression -
          // no return, no semicolon
        Arrays.sort(persons,
            (p1, p2) -> p1.getYear()-p2.getYear());
        Person.show(persons, "Ordered by age");

          // lambda as a compound statement -
          // return and semicolons, as usually
        Arrays.sort(persons, (p1, p2) ->
            {
                int d = p1.getName().compareTo(p2.getName());
                if (d != 0) return d;
                return p1.getYear() - p2.getYear();
            });
        Person.show(persons,"Ordered by name then age");
    }
}
```

The program prints

```
At the beginning:
Mary(1990) Joan(1992) Suzy(1992) Beth(1992) Suzy(1980) Katy(1982)
Ordered by age
Suzy(1980) Katy(1982) Mary(1990) Joan(1992) Suzy(1992) Beth(1992)
Ordered by name then age
Beth(1992) Joan(1992) Katy(1982) Mary(1990) Suzy(1980) Suzy(1992)
```

The scoping rules for lambdas are somewhat special. We can imagine that the compiler creates an object of an anonymous class implementing an interface and then implements its abstract method based on our lambda expression. However, it is not so — the body of a lambda behaves as a block inside the function it is defined in. This fact has several consequences, among others these

- inside the body of a lambda, one cannot define variables with same name as local variables from the surrounding scope;
- the reference **this** used inside the body of a lambda refers to an object of the surrounding class, *not* to an object of an anonymous class.

4.5 More examples

Let's look at some examples.

Listing 43 ELU-FInter/FInter.java

```java
@FunctionalInterface
interface Calc {
    boolean test(Double d);
}

@FunctionalInterface
interface Cons {
    void consume(Object ob);
}

public class FInter {
    public static void main (String[] args) {
        double mn = 0, mx = 10; // effectively final
        Calc[] arr = {
            d -> d <  0,        // type of d inferred
            d -> d >= 0,
            d -> mn <= d && d <= mx
        };

        Cons cons = ob -> System.out.print(ob + " ");

        for (double d = -2; d < 15; d += 5) {
            for (Calc calc : arr)
                cons.consume(calc.test(d));
            System.out.println();
        }
    }
}
```

The program prints

```
true false false
false true true
false true true
false true false
```

And another simple example. Here, we define an interface which declares a method **transform** which 'transforms' a strings — it takes a string and returns another string. Note that the function **transArray** in class **SimpleInter** takes, as its second argument 'something that implements **Transformation** interface'. In **main** we call this function passing this 'something' in three different ways:

- as an object of a separate class **Reverse** implementing **Transformation**;
- as an object of an anonymous class implementing the same interface;
- as a lambda.

Listing 44 ELK-SimpleInter/SimpleInter.java

```java
import java.util.Arrays;

interface Transformation {
    String transform(String arg);
}

class Reverse implements Transformation {
    @Override
    public String transform(String s) {
        char[] a = s.toCharArray();
        for (int i = 0, j = s.length()-1; i < j; ++i, --j) {
            char c = a[i];
            a[i] = a[j];
            a[j] = c;
        }
        return new String(a);
    }
}

public class SimpleInter {
    private static void transArray(String[] array,
                                   Transformation t) {
        for (int i = 0; i < array.length; ++i)
            array[i] = t.transform(array[i]);
    }

    public static void main (String[] args) {
        String[] arr = {"Mary", "Alice", "Janet", "Rachel"};
        System.out.println(Arrays.toString(arr));

          //object of a class
        transArray(arr, new Reverse());
        System.out.println(Arrays.toString(arr));

          // object of an anonymous class
        transArray(arr,
                   new Transformation() {
                       @Override
                       public String transform(String s) {
                           return s.toUpperCase();
                       }
                   });
        System.out.println(Arrays.toString(arr));

          // lambda
        transArray(arr, s -> "" + s.charAt(0));
        System.out.println(Arrays.toString(arr));
    }
```

```
}
```

The program prints

```
[Mary, Alice, Janet, Rachel]
[yraM, ecilA, tenaJ, lehcaR]
[YRAM, ECILA, TENAJ, LEHCAR]
[Y, E, T, L]
```

Another important example of using a lambda has been already shown in one of the previous examples — see Listing 35. The **sort** function called with two arguments expects as the second one 'something implementing the **Comparator** interface'. This interface *is* a functional interface, because it declares only one abstract method: **compare**. Therefore, as the implementation of this method is usually rather short, it is very convenient to use lambdas instead of creating separate classes whose only purpose is to 'wrap' the **compare** method.

Section 5

Introduction to generic classes

This section covers a few basic concepts related to generics, or parametrized classes, in Java.[3]

The subject is not easy; generics were added to Java rather late and their implementation is quite complicated, as it had to be consistent with earlier versions of the language. Let us also mention here that parametrized classes are somewhat related to class templates in C++ and Ada, but this similarity is rather misleading and superficial; it definitely should not be taken too literally.

5.1 *Type parameters*

Parametrized class uses as names of types arbitrary identifiers — called **type parameters** — which are then concretized, when we use objects of the generic type. Then we have to tell the compiler, explicitly or implicitly, what concrete class this identifier should denote. We introduce type parameters of a class like this:

```java
class Pair<F,S> {
    // here we use F and S as names of types
}
```

Here F and S stand for names of some, as yet unknown, types. In the definition of the class we can use names F and S as names of classes (but *not* primitive types!). Of course, there are no classes F or S, so when creating an object of our class, we have to specify (in angle brackets) what these names should stand for:

```java
Pair<Integer,String> p =
           new Pair<Integer,String>(2,"Sue");
  // or just (using diamond operator)
Pair<Integer,String> q = new Pair<>(1,"Lea");
```

In the second case, we used "type inferring" (diamond operator); we don't need to repeat types on the right-hand side, because the compiler already knows them looking at the left-hand side (but angle brackets, although empty, are still required).

However, in the code produced by the compiler, type parameter will *not* be present: at run time types denoted by, say, T will be just Object. Information about the type of T is only used at compile time.

Let us consider, as an example, the following program:

Listing 45 GMA-GenerPair/Pair.java

```java
import java.lang.reflect.Method;

public class Pair<F, S> {
    private static int count = 0;
    private F first;
    private S second;
    public Pair(F f, S s) {
```

[3] By *class* we understand a "regular" class, an interface, an abstract class or a record (see Sec. ??, p. ??).

```java
        count++;
        first  = f;
        second = s;
    }
    public F getFirst()  {return first; }
    public S getSecond() {return second;}
    public void setFirst(F f)  {first  = f;}
    public void setSecond(S s) {second = s;}
    public String toString() {return first + " " + second;}

    public static void main(String[] args) throws Exception {
        Pair<String, Integer> p1 = new Pair<>("A",1);
        Pair<String, String>  p2 = new Pair<>("C", "D");

          // what are the dynamic types of p1 and p2 ?
        System.out.println("Class of p1: " + p1.getClass() +
                "; Class od p2: " + p2.getClass());

          // method signatures
        for (Method m : p1.getClass().getDeclaredMethods())
            if (!m.getName().equals("main"))
                System.out.println(m); // type erasure

          // only one static member 'count'
        System.out.println(p1.count + " " + p2.count);

          // casting not needed, autoboxing int -> Integer
        p1.setSecond(2);

          // automatic unboxing Integer -> int
        int i = p1.getSecond();

          // no casting, must be a String
        String s = p2.getFirst();

        System.out.println("p1.second = " + i);
        System.out.println("p2.first  = " + s);
    }
}
```

As we can see, the compiler remembers types used when creating objects, so casting is not needed; calling **getFirst** on `p2` must give us a **String** and invoking **setSecond** on `p1` we pass an **int** and the compiler knows that it has to be converted to **Integer**. However, the dynamic type, or, strictly speaking, the so called “raw” type of both `p1` and `p2` is just **Pair**. We can see it from the output:

```
Class of p1: class Pair; Class od p2: class Pair
public java.lang.Object Pair.getSecond()
public void Pair.setFirst(java.lang.Object)
public void Pair.setSecond(java.lang.Object)
```

```
public java.lang.String Pair.toString()
public java.lang.Object Pair.getFirst()
2 2
p1.second = 2
p2.first  = C
```

Therefore, at run time, there exist only one type, namely just Pair. Notice that at run time arguments of setters (setFirst and setSecond) and return type of getters (getFirst and getSecond) are of type Object: this is known as **type erasure**. Only at compile time the compiler knows the required (declared) types of the two elements of a pair and will not allow us to insert (for example using setFirst) an object of a wrong type. Also notice that there exist only one static member count because dynamic types of both p1 and p2 are the same — just Pair.

Parametrized type can be, and are, useful, because they

- allow us to avoid explicit conversions, because the compiler knows expected types of arguments and return values of methods;
- make the code shorter and simpler to read and understand;
- allow the compiler to detect many errors related to type mismatch, which would otherwise manifest themselves at run time, triggering, for example, ClassType-Exceptions.

On the other hand, mainly because of the type erasure that we have just mentioned, there are some quite severe restrictions when defining and using parametrized types. If T is a type parameter

- We cannot create objects of type T;
- We cannot create arrays of (references to) objects of type T (but it *is* possible to create collections parametrized by T);
- We cannot use instanceof operator with T;
- Type T cannot be used for static fields (as parametrized class corresponds to one raw type);

Let us look at another example of a parametrized class: a class representing a queue. Making our class generic (parametrized) allows us to create queues of elements of various types preserving strict type checking:

Listing 46 GME-QueueGener/MyQueue.java

```java
public class MyQueue<E> {

    private class Node {
        E    data;
        Node next = null;
        Node(E d) { data = d; }
    }
    private Node head, tail;

    public MyQueue() {
        head = tail = null;
    }
    public void enqueue(E d) {
```

```java
        if (head == null)
            head = tail = new Node(d);
        else
            tail = tail.next = new Node(d);
    }
    public E dequeue() {
        E d = head.data;
        if ((head = head.next) == null) tail = null;
        return d;
    }
    public boolean empty() {
        return head == null;
    }
}
```

In **main** we create queues of **String**s and **Double**s and use them in a uniform way:

Listing 47 GME-QueueGener/QueueGener.java

```java
public class QueueGener {
    public static void main (String[] args) {
        MyQueue<String> queueS = new MyQueue<>();
        MyQueue<Double> queueD = new MyQueue<>();
        for (double d = 0.5; d < 5; d += 1) {
            queueS.enqueue(String.valueOf(d));
            queueD.enqueue(d); // boxing
        }
        while (!queueS.empty() && !queueD.empty()) {
              // no casting required
            String s = queueS.dequeue();
            double d = queueD.dequeue();
            System.out.println(
                    "String: " + s +  "   " +
                    "Double: " + d);
        }
    }
}
```

In the program, class **MyQueue** implements a queue of objects of any type. This does not mean that we can enqueue any object to any queue: when creating a queue, we decide what the type of enqueued objects should be. For example, the queue queueS is declared as queue of **String**s. Therefore, the compiler will not allow us to enqueue anything other than **String** (and in the case of queueD — **Double**s). Also notice that when enqueueing and dequeueing elements, we don't have to use casting: the compiler knows what type is expected and will even automatically perform boxing/unboxing of primitive types for us.

Let us consider another example. We define a generic interface **Operat** which declares one function representing an operator, i.e. a function taking two arguments of the same type and returning a result also of this type. Object implementing this interface will then be used by static functions defined in class **Tools**. Note, how we define generic *static functions* — here we inform the compiler that **T** is not a class but a type parameter for every static function defined in the class separately; one of these functions performs the so called *left-folding* and the other combines two sequences into one. We don't create any concrete classes implementing **Operat**; instead, in **main**, we just pass objects of anonymous classes implementing it:

Listing 48 ELZ-IFac/IFaces.java

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

interface Operat<T> {
    T oper(T lhs, T rhs);
}

class Tools {
    public static <T> T foldl(
            Operat<T> op, List<T> list, T id) {
        T acc = id;
        for (T e : list)
            acc = op.oper(acc,e);
        return acc;
    }

    public static <T> List<T> combine(
            Operat<T> op, List<T> l1, List<T> l2) {
        List<T> res = new ArrayList<T>();
        int size = Math.min(l1.size(), l2.size());
        for (int i = 0; i < size; ++i)
            res.add(op.oper(l1.get(i), l2.get(i)));
        return res;
    }
}

public class IFaces {
    public static void main (String[] args) {
        Operat<Integer> operInt = new Operat<Integer>() {
            @Override
            public Integer oper(Integer lhs, Integer rhs) {
                return lhs + rhs;
            }
        };
        Operat<String> operStr = new Operat<String>() {
            @Override
            public String oper(String lhs, String rhs) {
                return lhs + rhs;
            }
        };
```

```java

        List<Integer> listInt1  = Arrays.asList(1,2,3,4),
                      listInt2  = Arrays.asList(5,6,7,8);
        List<Integer> intRes =
            Tools.combine(operInt, listInt1, listInt2);
        System.out.println("intRes = " + intRes);

        List<String> listStr1  = Arrays.asList("a","b","c"),
                     listStr2  = Arrays.asList("1","2","3");
        List<String> strRes =
            Tools.combine(operStr, listStr1, listStr2);
        System.out.println("strRes = " + strRes);

        int intFold = Tools.foldl(operInt, intRes, 0);
        System.out.println("intFold = " + intFold);

        String strFold = Tools.foldl(operStr, strRes, "");
        System.out.println("strFold = " + strFold);
    }
}
```

The program prints

```
intRes = [6, 8, 10, 12]
strRes = [a1, b2, c3]
intFold = 36
strFold = a1b2c3
```

Other examples: here we use generic functional interfaces.

The first example: here we define a generic functional interface which represents an *operator*, i.e., function accepting two objects of the same type and returning a result also of this type:

Listing 49 ELW-LambdaInter/MyBiOp.java

```java
import java.util.ArrayList;
import java.util.List;

@FunctionalInterface
interface MyBiOpInterface<T> {
    T apply(T a, T b);
}

class Mult implements MyBiOpInterface<Double> {
    @Override
    public Double apply(Double a, Double b) { return a*b; }
}

public class MyBiOp {
```

```
    public static void main (String[] args) {
        List<MyBiOpInterface<Double>>
                    opers = new ArrayList<>();

          // addition - reference to (static) method
        opers.add( Double::sum );

          // subtraction - anonymous class
        opers.add( new MyBiOpInterface<Double>() {
            @Override
            public Double apply(Double a, Double b) {
                return a - b;
            }
        });

          // multiplication - object of a named class
          // implementing the MyBiOpInterface interface
        opers.add( new Mult() );

          // division - lambda
        opers.add( (a,b) -> a/b );

          // with closure ('shift' is effectively final)
        int shift = 10;
        opers.add( (a,b) -> a/b + shift);

        for (MyBiOpInterface<Double> op : opers)
            System.out.println(op.apply(10.5,3.5));
    }
}
```

The program prints

```
14.0
7.0
36.75
3.0
13.0
```

and another one

Listing 50 ELX-FuncInter/FuncInter.java

```
@FunctionalInterface
interface MyInterface<T> {
    int len(T t);
}

public class FuncInter {
    static <T> int calc(T arg, MyInterface<T> f) {
```

```java
        return f.len(arg);
    }

    public static void main (String[] args) {
        String s = "Alice";
        int result1 = calc(
            s,
            new MyInterface<String>() {
                @Override
                public int len(String s) {
                    return s.length();
                }
            });
        System.out.println("result1 = " + result1);

        Double d = 123.456; // boxing
        int result2 =
            calc(d, v -> v.toString().length());
        System.out.println("result2 = " + result2);

        int result3 =
            calc(d, v -> (int)(1000*v+4));
        System.out.println("result3 = " + result3);
    }
}
```

The program prints

```
result1 = 5
result2 = 7
result3 = 123460
```

The following example is very similar to that in Listing 48, but now we use lambdas: directly as an argument, or to create a variable which could be used more than once:

Listing 51 EMA-IFacLam/IFacesLam.java

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

interface Operat<T> {
    T oper(T lhs, T rhs);
}

class Tools {
    public static <T> T foldl(
            Operat<T> op, List<T> list, T id) {
        T acc = id;
        for (T e : list)
```

```java
            acc = op.oper(acc,e);
        return acc;
    }

    public static <T> List<T> combine(
            Operat<T> op, List<T> l1, List<T> l2) {
        List<T> res = new ArrayList<T>();
        int size = Math.min(l1.size(), l2.size());
        for (int i = 0; i < size; ++i)
            res.add(op.oper(l1.get(i), l2.get(i)));
        return res;
    }
}

public class IFacesLam {
    public static void main (String[] args) {
        Operat<String> operStr = (lhs, rhs) -> lhs + rhs;

        List<Integer> listInt1  = Arrays.asList(1,2,3,4),
                      listInt2  = Arrays.asList(5,6,7,8);
        List<Integer> intRes =
            Tools.combine((lhs, rhs) -> lhs + rhs,
                              listInt1, listInt2);
        System.out.println("intRes = " + intRes);

        List<String> listStr1  = Arrays.asList("a","b","c"),
                     listStr2  = Arrays.asList("1","2","3");
        List<String> strRes =
            Tools.combine(operStr, listStr1, listStr2);
        System.out.println("strRes = " + strRes);

        int intFold = Tools.foldl(
                (lhs, rhs) -> lhs + rhs, intRes, 0);
        System.out.println("intFold = " + intFold);

        String strFold = Tools.foldl(operStr, strRes, "");
        System.out.println("strFold = " + strFold);
    }
}
```

The program prints

```
intRes = [6, 8, 10, 12]
strRes = [a1, b2, c3]
intFold = 36
strFold = a1b2c3
```

The last example illustrates an interface with one *defined* default method; note, that we have to inform the compiler that **S** is an additional type parameter (**T** and **R** have already been declared as such.) The interface represents an operation (mapping), called here **apply**, from values of one type to values of another type (T→R). However, the interface *defines* also a method (**compos**). This is a method, so it is called on an object representing one operation (say, f : T→R), takes a value arg of type T as an argument and an object representing another operation (say, g : R→S). It then returns the result of the composition $(g \circ f)(\text{arg})$. The following program

Listing 52 EMB-Compos/Composit.java

```java
@FunctionalInterface
interface Func<T,R> {
    R apply(T e);
    default <S> S compos(T arg, Func<R,S> g) {
        return g.apply(apply(arg));
    }
}

public class Composit {
    public static void main(String[] args) {
        Func<String,Integer> f = s -> s.length();
        System.out.println("g(f(\"abc\")) = " +
                f.compos("abc", v -> v*Math.PI));
    }
}
```

prints

```
g(f("abc")) = 9.42477796076938
```

as the length of "abc" is 3 and $3\pi \approx 9.42477796076938$. Note that the g operation was given as a lambda but still the compiler was able to deduce that the type S of the result should be Double.

5.2 Bounded types

Specifying a type parameter, say T, we can limit possible types that it can represent to types meeting some requirements. This allows us, for example, to use, on objects of T, methods not necessarily only from class Object but more specific methods from other classes that has been specified as type bounds. Such restrictions define sets of classes that can be substituted for T: then we can use methods which the classes from these sets contain. The syntax looks like this:

```
T extends Type1 & Type2 & Type3  & ... & TypeN
```

where:

- T is the type parameter;
- Type1 is the name of a class or an interface;
- Type2 ... TypeN are names of interfaces.

As we can see, if among restrictions there is a (concrete or abstract) class, it must be specified as the first in the list, all others bounds must be interfaces. Types Type1 ... TypeN may be themselves parametrized (generic) classes or interfaces.

Objects of types restricted in this way can be used as objects of type Type1: compilation will fail if we try to substitute for T something that does not extend or implement

Type1 or does not implement **Type2** ... **TypeN**. Therefore, it will be safe to call methods of all these restricting classes or interfaces.

In the example below, class **GenArr** is parametrized by type **T**, but not arbitrary but only by classes implementing **Comparable** — therefore, we can use the method **compareTo** on objects from the array (in order to find the minimum and maximum elements of the array):

Listing 53 GMH-MinMaxGener/MinMaxGener.java

```java
class GenArr<T extends Comparable<T>> {
    private T[] arr;
    private T   min, max;

    public GenArr(T[] arr) {
        if (arr == null || arr.length == 0)
            throw new IllegalArgumentException();
        min = arr[0];
        max = arr[0];
        for (int i = 1; i < arr.length; ++i) {
              // we can use compareTo, as we know
              // that T extends Comparable
              // (and the compiler has checked it)
            if (arr[i].compareTo(min) < 0) min = arr[i];
            if (arr[i].compareTo(max) > 0) max = arr[i];
        }
    }
    public T getMin() { return min; }
    public T getMax() { return max; }
}

public class MinMaxGener {
    public static void main(String[] args) {
        GenArr<Integer> mnmxI =
            new GenArr<>(new Integer[]{3, -2 , -7, 2});
        GenArr<String> mnmxS =
            new GenArr<>(new  String[]{"A", "Z", "C"});

        System.out.println("I - min = " + mnmxI.getMin() +
                           "\nI - max = " + mnmxI.getMax());
        System.out.println("S - min = " + mnmxS.getMin() +
                           "\nS - max = " + mnmxS.getMax());
    }
}
```

It is also possible to parametrize just a method (also static), not the whole class. We then specify the name of the type parameter (in angle brackets) just before the return type of the method in question:

```
access_specifier [static] <T> return_type name(parameter_list)
```

For example, in the following program, the class **MinMaxMeth** is *not* generic, however the method **getMax** is: it may take an array of elements of any type, as long as this type implements **Comparable**:

Listing 54 GMI-MinMaxMeth/MinMaxMeth.java

```
class MinMaxMeth {
    static <T extends Comparable<T>> T getMax(T[] arr) {
        if (arr == null || arr.length == 0)
            throw new IllegalArgumentException();
        T max = arr[0];
        for (int i = 1; i < arr.length; ++i)
            if (arr[i].compareTo(max) > 0) max = arr[i];
        return max;
    }

    public static void main(String[] args) {
        int mxi = getMax(new Integer[]{3, -2 , -7, 2});
          // one may enforce type to be substituted for T;
          // usually, as here, not needed, as the correct
          // type will be inferred by the compiler anyway
        String mxs = MinMaxMeth.<String>getMax(
                new  String[]{"A", "Z", "C"});

        System.out.println("I - max = " + mxi);
        System.out.println("S - max = " + mxs);
    }
}
```

Section 6

Enum types

Enum types roughly correspond to enumerations in C++, but are implemented in a different way: they are all *objects*, in contrast to C++ where they are always backed by an integer type. They are very useful in situations when we need a type with only very limited number of possible values.

6.1 Basic definitions

Defining an enum we define a new type (just like defining a class). However, this type is somewhat special: limited number of *unmodifiable* enum constants (which are implements as immutable objects) of this type is created when an enum class is loaded by the JVM and then it is impossible to create any more such objects. Therefore, we can say, that this type defines just a — usually small — set of constants. In fact, we know another example of type with only a few possible values: type **boolean** has only two — **true** and **false**. However, **boolean** is a primitive type, while enum constants are full-fledged objects.

Each of these constants has a fixed name, as is the case for **boolean**s, and can be in fact a singleton of a *different* class.

Enums can be useful for defining types that by their very nature have only a small number of possible values: there are only two sexes, four seasons, seven Wonders of the World, four card suits and four Horsemen of the Apocalypse. Without enums, we could just assign numbers to them; for example, in a class or an interface we could write

```
final static int CLUBS    = 0,
                 DIAMONDS = 1,
                 HEARTS   = 2,
                 SPADES   = 3;
```

but this poses a lot of problems

- if a function takes a card suit as its parameter, it has to declare it as an **int**. But then nothing can prevent us from passing, for example, 129 as the argument, which wouldn't make any sense;
- of course, all functions operating on card suits could check if the value passed to them is in the range $[0, 3]$, but then it would be hard to add a new suit (joker...) — we would have to correct the checking and interpretation in many places;
- in many situations we would have to remember the interpretation of the numbers; for example, printing just numbers as they are wouldn't be very informative — we would have to "convert" them manually into **String**s or do something equivalent.

Let us look at an example:

Listing 55 CYG-Enum1/EnumEx1.java

```java
public class EnumEx1 {

    public enum Season {SPRING, SUMMER, FALL, WINTER};
```

```java

    public static void main (String[] args) {
        Season[] seasons = Season.values();             //+1
        for (Season s : seasons)
            System.out.print(s + " ");                  //+2
        System.out.println();

        System.out.println(Season.WINTER + " is the " +
            (Season.WINTER.ordinal()+1) + "th season");//+3
        Season f = Season.valueOf("FALL");              //+4
        System.out.println("FALL is " + f + "...");
        System.out.println("Is f equal to FALL? " +
                                (f == Season.FALL));    //+5
        for (Season s : seasons)
                System.out.print(german(s) + " ");
        System.out.println();
    }

    private static String german(Season s) {
        return switch (s) {                             //+6
            case SPRING -> "Fr\u00fchling";
            case SUMMER -> "Sommer";
            case FALL   -> "Herbst";
            case WINTER -> "Winter";
        };
    }
}
```

The program prints

```
SPRING SUMMER FALL WINTER
WINTER is the 4th season
FALL is FALL...
Is f equal to FALL? true
Frühling Sommer Herbst Winter
```

We define an enum type **Season**. The definition is inside a class here, but this is not important; equally well we could have defined the enum (with **public** or default accessibility) in a separate file. This enum has exactly four values corresponding to four, and only four, objects: SPRING, SUMMER, FALL, WINTER. What can we do with type **Season** and its constants?

- static function **values()** returns an array of all (four in our case) constants of the enum (line //+1);
- method **toString()** is automatically overridden and returns the name of the enum constant (it is used in line //+2);
- we can call **ordinal()** on enum constants (remember that these are objects) and get their 'index' — starting from 0 and in the order as they were defined (//+3);
- static function **valueOf(String name)**) returns enum constant named name; if there were no constant with this name, exception would have been thrown (//+4).

If we have two references to the same enum constant, they are exactly equal, i.e., they point to the same object, because each enum constant is represented by exactly one object. Therefore, to compare enum constant we don't need (although we can) to use method **equals**: just '==' or '!=' are safe and sufficient (//+5).

- as integer types and **String**s, enum constants may be used in **switch** statements (//+6). In **case** clauses we don't have to use full names (like **Season.SPRING**), because the type of the selector (s in our case) is known to the compiler.

After compiling the above program

```
java> ls -1 EnumExample1*
    EnumExample1.java
java> javac EnumExample1.java
java> ls -1 EnumExample1*.class
    EnumExample1$1.class
    EnumExample1.class
    EnumExample1$Season.class
```

we can see that the compiler created *one* class for the **Season** type (which in our case was embedded in the main class, but this is not important here — anyway, a separate class has been created). The program prints

```
SPRING SUMMER FALL WINTER
WINTER is the 4th season
FALL is FALL...
Is f equal to FALL? true
Frühling Sommer Herbst Winter
```

The third line convinces us that indeed two references to FALL are exactly equal, that is they refer to the same object (and therefore can be compared by '==').

Enumerations are also comparable. If e1 and e2 are two enumerators of the same enumeration, then they can be compared in the usual way

```
e1.compareTo(e2)
```

The order is determined by the order in which enumerators are declared in the definition of enumeration.

In the following example there are two enumerations describing suits

Listing 56 CYE-SimpEnum/Suit.java

```java
public enum Suit {CLUBS,DIAMONDS,HEARTS,SPADES};
```

and ranks

Listing 57 CYE-SimpEnum/Rank.java

```java
public enum Rank {TWO,THREE,FOUR,FIVE,SIX,SEVEN,
                  EIGHT,NINE,TEN,JACK,QUEEN,KING,ACE};
```

of playing cards. The class **Card** has two fields, both are enumerators (rank and suit); it also defines one static function:

Listing 58 CYE-SimpEnum/Card.java

```java
public class Card {
    private Rank rank;
    private Suit suit;

    public Card(Rank rank, Suit suit) {
        this.rank = rank;
        this.suit = suit;
    }

    public Rank getRank() { return rank; }
    public Suit getSuit() { return suit; }

    public static Card getHigher(Card c1, Card c2) {
        if      (c1.rank.ordinal() > c2.rank.ordinal())
            return c1;
        else if (c1.rank.ordinal() < c2.rank.ordinal())
            return c2;
        else if (c1.suit.ordinal() > c2.suit.ordinal())
            return c1;
        else
            return c2;
    }

    @Override
    public String toString() {
        return rank + " of " + suit;
    }
}
```

In Main we read data and use our classes

Listing 59 CYE-SimpEnum/Main.java

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        System.out.print("Suits:");
        for (Suit s : Suit.values())
            System.out.print(" " + s );
        System.out.print("\nRanks:");
        for (Rank r : Rank.values())
            System.out.print(" " + r);
        System.out.println();

        Scanner scan = new Scanner(System.in);
        System.out.print("1. rank -> ");
        String r1 = scan.next().toUpperCase();
```

```java
        System.out.print("1. suit -> ");
        String s1 = scan.next().toUpperCase();
        System.out.print("2. rank -> ");
        String r2 = scan.next().toUpperCase();
        System.out.print("2. suit -> ");
        String s2 = scan.next().toUpperCase();

        Rank rank1 = Rank.valueOf(r1);
        Suit suit1 = Suit.valueOf(s1);
        Rank rank2 = Rank.valueOf(r2);
        Suit suit2 = Suit.valueOf(s2);

        Card card1 = new Card(rank1,suit1);
        Card card2 = new Card(rank2,suit2);

        Card higher = Card.getHigher(card1,card2);
        System.out.println("Card1 = " + card1 +
                          ", Card2 = " + card2 +
                       "\nHigher card: " + higher);
    }
}
```

6.2 *Fields, constructors and methods in enumerations*

In the previous example, enum constants differ only in their names, so objects representing them were of the same type. However, one can define methods for them, and these methods can have different implementation for each constant separately. First, let us look at the following example:

Listing 60 CYH-EnumMet/EnumEx2.java

```java
public class EnumEx2 {

    enum Season {
        SPRING {                                    //+1
            @Override
            public String toString() {return "Printemps";}
        },
        SUMMER {
            @Override
            public String toString() {
                return "\u00c9t\u00e9";
            }
        },
        FALL {
            @Override
            public String toString() {return "Automne";}
        },
```

```
        WINTER {
            @Override
            public String toString() {return "Hiver";}
        }
    };

    public static void main (String[] args) {
        Season[] seasons = Season.values();
        for (Season s : seasons)
                System.out.print(s + " ");         //+2
        System.out.println();
        for (Season s : seasons)
                System.out.print(s.name() + " ");   //+3
        System.out.println();
    }
}
```

Here, *for each constant separately*, in braces just after their names (//+1), we override **toString** method from class **Object**. When we compile this program:

```
java> ls -1 EnumExample2*
    EnumExample2.java
java> javac EnumExample2.java
java> ls -1 EnumExample2*.class
    EnumExample2$1.class
    EnumExample2.class
    EnumExample2$Season$1.class
    EnumExample2$Season$2.class
    EnumExample2$Season$3.class
    EnumExample2$Season$4.class
    EnumExample2$Season.class
```

we can see that the compiler created distinct classes for each constant. This is quite obvious: they cannot be objects of the same class, as it would be impossible to have in one class several implementations of the same method **toString**! Now, when we print s (in line //+2), the overridden version of **toString** will be used. Still, however, the original 'true' name of enum constant may be retrieved by method **name()** (//+3), as we can see from the output

```
Printemps Été Automne Hiver
SPRING SUMMER FALL WINTER
```

Methods that we can define for enum constants are not limited to those inherited. We can define our own methods; moreover, we can add data fields and a constructor (but only one) as well!

In the program below we add two data fields: desc and numOfMonths (line //+3). We also define a standard constructor. Fields and constructors in enumerations are by definition **private**, so we don't even need to declare them as such. Our constructor takes two arguments, so we supply them when defining the enum constants (line //+1):

Listing 61 CYI-EnumAbs/EnumEx3.java

```java
public class EnumEx3 {
    enum Season {
        SPRING("nice",2) {                                   //+1
            @Override                                        //+2
            public String getDesc() {
                return name() + ": " + desc;
            }
        },
        SUMMER("hot",3) {
            @Override
            public String getDesc() {
                return desc + " in " + name();
            }
        },
        FALL("so, so",4) {
            @Override
            public String getDesc() {
                return name() + "? Well, " + desc;
            }
        },
        WINTER("cold",3) {
            @Override
            public String getDesc() {
                return name() + "! very " + desc;
            }
        }; // <-- semicolon after the last value!

          // fields and constructors are private anyway
        String     desc;                                     //+3
        int numOfMonths;
        Season(String d, int i) {                            //+4
            desc        = d;
            numOfMonths = i;
        }

        public int getNumb() { return numOfMonths; } //+5
        public abstract String getDesc();                    //+6
    };

    public static void main (String[] args) {
        for (Season s : Season.values())
            System.out.println(s.getNumb() +
                    " months - " + s.getDesc());
    }
}
```

There are also two methods. One of them, **getNumb** has the same common implementation for all objects (line //+5). However, the second one, **getDesc**, is in line //+5 only *declared* (as abstract) and implemented differently for each constant (line //+2). The program prints

```
2 months - SPRING: nice
3 months - hot in SUMMER
4 months - FALL? Well, so, so
3 months - WINTER! very cold
```

Objects representing enum constants are created once only, when the enum is loaded by the JVM, even before initialization of static members of the class, if there are any.

6.3 Enumarations implementing an interface

Enumarator can also implement interfaces. Let us look at an example:

Listing 62 CYK-EnumInterf/CompEnum.java

```java
import java.util.Arrays;
import java.util.Comparator;

public class CompEnum {
    public static void main(String[] args) {
        String[]  arr = {"Alice", "Sue", "Janet", "Bea"};

        Arrays.sort(arr, StrCmp.ByLenAsc);
        System.out.println(Arrays.toString(arr));

        Arrays.sort(arr, StrCmp.ByLenDesc);
        System.out.println(Arrays.toString(arr));

        Arrays.sort(arr, StrCmp.ByLexAsc);
        System.out.println(Arrays.toString(arr));

        Arrays.sort(arr, StrCmp.ByLexDesc);
        System.out.println(Arrays.toString(arr));
    }
}

enum StrCmp implements Comparator<String> {
    ByLenAsc( (s1, s2) -> s1.length() - s2.length()),
    ByLenDesc((s1, s2) -> s2.length() - s1.length()),
    ByLexAsc( (s1, s2) -> s1.compareTo(s2)),
    ByLexDesc((s1, s2) -> s2.compareTo(s1)); // <- semicolon!
    Comparator<String> cmp;                  // field
    StrCmp(Comparator<String> c) {           // constructor
        cmp = c;
    }
    @Override
    public int compare(String s1, String s2) {
        return cmp.compare(s1, s2);
    }
```

```
}
```

The enumeration **StrCmp** implements the **Comparator** interface. It has a field cmp of type **Comparator** which will be different for all enumeration constants: it is set in the constructor. When calling the constructor, we have to pass something which *is* a comparator — here, we use different lambdas. Objects of this enumeration can then be used wherever a comparator (of **String**s) is expected, as we can see form the output of the program:

```
[Sue, Bea, Alice, Janet]
[Alice, Janet, Sue, Bea]
[Alice, Bea, Janet, Sue]
[Sue, Janet, Bea, Alice]
```

Section 7

Introduction to collections

Collections represent... well, collections, i.e., aggregates of pieces of data of a given type organized in some form. Objects representing collections in Java have to implement the interface Collection (there is a different kind of collections, map, which implement different interface: Map). All collections are **iterable** (they implement also the Iterable interface) — the importance of this will soon become clear.

We already know interfaces. Generally speaking its something like a class, but with declarations of (abstract) methods — without implementation (although, as we know, it can contain some *default* methods *with* implementation). Other classes may *implement* this interface by providing *definitions* of all the abstract methods. Any class may inherit from (extend) only one class, but can implement many interfaces. Important:

1. Collections (and maps) are always collections of *references* (pointers) — *never* objects themselves!
2. Collections fall into two general categories: **Collection**s and **Map**s.
3. Classes representing collections are generic — they are parametrized by the type of the elements they hold.
4. Their properties and functionality (API) is specified by a hierarchy of interfaces that they implement (i.e., they implement methods declared in these interfaces).

Very often we want a collection of data of a primitive type, like int or double. We cannot insert such data into any collection, because these are not references. However, for each such type, there is a class, objects of which serve as wrappers of such data: Integer for ints, Double for doubles, etc. Normally, we don't even have to create object of these wrapper classes ourselves — this will be done automatically: if a collection is declared as a collection of Integers, we can insert just ints and they will be automatically wrapped into objects of type Integer and put into this collection. Such automatic conversion of values of primitive types to objects is called **boxing**; the reverse operation is called **unboxing**.

When creating objects representing a collection, we should always specify the type of its elements; for maps there are two types to be specified: type of keys and of values; we do it using angle brackets, as we will see in the examples below.

7.1 Collections

The Collection interface contains the following methods, which therefore must be implemented in all concrete classes representing collections:

- add — adds an element;
- addAll — adds all elements of another collection;
- clear — removes all elements;
- contains — checks if an object belongs to this collection;
- containsAll — checks if all elements of another collection belong to this collection;
- equals — compares this collection with another one;
- hashCode — return hash code of the collection (overriding the method inherited from Object);

- **empty** — returns **true** if this collection is empty;
- **iterator** — returns an iterator associated with this collection (this is an implementation of the interface **Iterable**);
- **parallelStream** — returns parallel stream with this collection as its source (defaulted);
- **remove** — removes an element from this collection;
- **removeAll** — removes all elements from another collection from this collection;
- **removeIf** — removes all elements satisfying a given predicate (defaulted);
- **retainAll** — retains in this collection only elements from another collection;
- **size** — returns the number of elements in this collection;
- **splitIterator** — returns a split iterator associated with this collection (defaulted);
- **stream** — returns the stream with this collection as its source;
- **toArray** — returns an array containing all elements of this collection;

Some collections do not permit certain operations — then their implementations just throw an exception; we say that these operations (represented by methods) are then *optional*.

There are some subinterfaces of **Collection** that are more specific for various kinds of collections. The most important are

- **List**: lists represent collections of elements that are ordered (like elements of an array) — it makes sense to talk about element number 0, number 1, etc. Therefore, there are some methods which are not applicable to all collections but specific to lists: e.g., **get(i)** which returns element with the index specified, **indexOf(val)** which returns the index of an element with a given value, **add(i,val)** which adds new element at the position specified, etc. For all lists, adding a new element at the end is very efficient; at other locations — not necessarily so. A list may contain equal values at different positions. There are two main implementations of lists (concrete classes, *not* interfaces):
  - **ArrayList** — implementation is based on arrays. Adding elements *not* at the end may be very inefficient but access to all elements (by index) is almost immediate.
  - **LinkedList** — implementation is based on doubly-linked lists. Adding and removing may be fast, but access is slower.
- **Set**: sets represent collections of *unique* elements (no two elements are equal). There are two main implementations of sets:
  - **TreeSet** — implementation is based on red-black tree; the elements have a well defined order (therefore they have to be *comparable*), access to elements is fast (logarithmic).
  - **HashSet** — implementation is based on hashing technique — access to elements is even faster (constant time) but the order is unspecified.

7.2 Maps

Maps represent sets of *pairs* of objects: the first element of a pair is a **key** and the second is a **value** associated with this key (as elements of an array are associated with their indices, which therefore may be viewed as playing the rôle of keys). Types of keys and values may be, and usually are, different. As keys are used as “indices”, there may be no two equal keys in any map — they have to be unique (but the same value may be associated with two different keys). All maps have to implement methods from **Map** interface; among others, these are:

- **put(key, val)** — adds a (key,value) pair to this map;
- **putIfAbsent(key, val)** — adds a pair to this map if the specified key is *not* already present (returning **null**); otherwise it does nothing and returns the value already associated with the key;
- **get(key)** — returns the value associated with a given key, or **null** if the map doesn't contain this key;
- **getOrDefault(key, defaultVal)** — returns the value associated with a given key, or defaultVal if the map doesn't contain this key;
- **clear** — removes all elements;
- **containsKey** — checks if there is a pair (the so called **entry**) in this map with a given key;
- **containsValue** — checks if there is a pair in this map with a given value (this is usually rather inefficient);
- **size** — returns the number of elements (entries) in this map;
- **isEmpty** — checks if the map is empty;
- **remove(key)** — removes the entry with a given key;
- **keySet** — returns a set of all keys;
- **values** — returns a collection of all values;
- **entrySet** — returns a set of entries, each of which has a key and a value accessible by methods **getKey** and **geValue**;
- etc.

There are two main implementations of maps:

- **TreeMap** — implementation is based on red-black tree of *keys*; the elements must have a well defined order. Therefore, *keys* have to be **Comparable**, or you have to pass a **Comparator** to the constructor. Access to elements is fast (logarithmic).
- **HashMap** — implementation is based on hashing technique — access to elements is even faster (constant time) but the order is unspecified.

7.3 Iterators

Iterators are objects representing a "view" of the elements of a collection and can yield on demand these elements, remembering which have already been yielded and knowing if there is anything that has not been returned yet. Iterators are also "generic", so we should always specify the type of elements which they will return. The **Iterator** interface declares methods (here **T** is the type of elements)

```
boolean hasNext()
T next()
```

The second one returns the next element from those that have not been returned yet. The first one tells us if there is still an element that has not been returned by **next** (calling **next** when **hasNext** returns **false** should always trigger the exception **NoSuchElementException**).

There is the third operation which can be invoked on an iterators, **remove**: it removes the last element of the underlying collection that has just been returned by **next**. However, we don't have to implement it, because this operation is optional and it already *has* a default implementation (which just throws an exception.) There is also a default (already implemented) method **forEachRemaining**.

The collection of elements which backs an iterator can be anything that implements **Iterable** interface. All "real" collections — classes from the standard library that implement **Collection** — *do* implement **Iterable**. **Iterable** declares just one abstract method:

```
Iterator<T> iterator()
```

(where **T** is the type of elements) which returns an iterator associated with a given collection implementing **Iterable**. In fact, this need not to be a "real" collection, it should only behave as one from the point of view of the iterator returned.

Let us consider an example — here object of type **IterableRange** plays the rôle of a collection, although there is no array or any other "true" collection involved: however, it implements **Iterable** and the iterator returned by its **iterator** method meets all the requirements of an iterator:

Listing 63 JIX-RangIter/RangIter.java

```java
import java.util.Iterator;
import java.util.NoSuchElementException;

  // main class for testing
public class RangIter {
    public static void main (String[] args) {
        Iterable<Integer> iterab1 = new IterableRange(3,7);
        Iterator<Integer> iter = iterab1.iterator();
        while (iter.hasNext())
            System.out.print(iter.next() + " ");

        System.out.println("\nand now foreach:");
        for (Integer i : new IterableRange(3,7))
            System.out.print(i + " ");
        System.out.println();
    }
}

  // objects of this class are 'iterable', i.e., they
  // behave (at least to some extend) as collections
class IterableRange implements Iterable<Integer> {
    private int a, b;
    IterableRange(int a, int b) {
        this.a = a;
        this.b = b;
    }
    public Iterator<Integer> iterator() {
        return new RangeIterator(a,b);
    }
}

  // object of this class bahave like interators
  // traversing the IterableRange 'collections'
```

```java
class RangeIterator implements Iterator<Integer> {
    private int a, b;
    private int curr;
    RangeIterator(int a, int b) {
        this.a = a;
        this.b = b;
        curr = a;
    }
    @Override
    public boolean hasNext() {
        return curr <= b;
    }
    @Override
    public Integer next() {
        if (!hasNext()) throw new NoSuchElementException();
        return curr++;
    }
    // since Java 1.8 remove has a default implementation
}
```

All classes are here defined in one file for simplicity, it is generally not a recommended practice. Note that the class **RangeIterator** could have been defined inside **IterableRange** (see sec. 4.3). Note also the *for-each* (called also *ranged for*) loop used here in the **main** function:

```java
for (Integer i : new IterableRange(3, 7))
    System.out.print(i + " ");
```

This form of loops works not only with standard collections, but in fact with anything that implements **Iterable**. Basically, the form

```java
for (Type elem : iterable_object)
    do_something_with_elem
```

is by the compiler automatically translated into something like this

```java
{
    Iterator<Type> it = iterable_object.iterator();
    while (it.hasNext()) {
        Type elem = it.next();
        do_something_with_elem
    }
}
```

It is quite common to create classes that are iterable and at the same time iterators. Implementation of the **iterator** method is then very simple — it just returns **this**. Let us rewrite the previous example in this way:

Listing 64 JIZ-IterIterable/IterIterable.java

```java
import java.util.Iterator;
import java.util.NoSuchElementException;
```

```java

  // main class for testing
public class IterIterable {
    public static void main (String[] args) {
        for (Integer e : new IterableRange(3,11))
            System.out.print(e + " ");
        System.out.println();
    }
}

  // objects of this class are iterable and iterators!
class IterableRange implements Iterable<Integer>,
                               Iterator<Integer> {
    private int a, b, curr;
    IterableRange(int a, int b) {
        this.a = a;
        this.b = b;
        curr   = a;
    }
    @Override
    public Iterator<Integer> iterator() {
        return this;
    }
    @Override
    public boolean hasNext() {
        return curr <= b;
    }
    @Override
    public Integer next() {
        if (!hasNext()) throw new NoSuchElementException();
        return curr++;
    }
}
```

As we can see, implementation is now much simpler and shorter.

7.4 Examples

Let us consider a few examples. First, lists and sets (i.e., collections implementing the Collection interface)

Listing 65 HUG-Lists/AList.java

```java
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.List;
import java.util.Set;
import java.util.HashSet;
```

```java
import java.util.TreeSet;
import java.util.Iterator;

public class AList {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Sue");
        list.add("Lea");
        list.add("Ann");
        list.add("Kim");
        list.add("Lea");
        list.add(1,"Amy");  // inefficient!

        System.out.println("First 'Lea' under index " +
                                     list.indexOf("Lea"));
        System.out.println("Last  'Lea' under index " +
                                 list.lastIndexOf("Lea"));
        System.out.println("Does the list contain 'Sue': " +
                                       list.contains("Sue"));
        System.out.println("Does the list contain 'Bea': " +
                                       list.contains("Bea"));
        System.out.println("Size of list: " + list.size());

          // traditional looping
        System.out.print("With get():  ");
        for (int i = 0; i < list.size(); ++i)
            System.out.print(" " + list.get(i));
          // `foreach' loop
        System.out.print("\nAnother way: ");
        for (String s : list)
            System.out.print(" " + s);
          // iterators
        System.out.print("\nnow iterator:");
        Iterator<String> iter = list.iterator();
        while (iter.hasNext())
            System.out.print(" " + iter.next());

          // sorting and sublists
        Collections.sort(list);
        System.out.println("\nSorted:      " + list);
        List<String> subl = list.subList(1,4);
        System.out.println("Sublist:      " + subl);

          // HashSet
        Set<String> hSet = new HashSet<>(list);
        System.out.println("Hash set:     " + hSet);
          // TreeSet
        Set<String> tSet = new TreeSet<>(list);
        System.out.println("Tree set:     " + tSet);
        System.out.println("Does the tSet contain 'Sue': " +
```

```
                                  tSet   .contains("Sue"));
        tSet.add("Zoe");
        tSet.remove("Lea");
        System.out.println("tSet now:     " + tSet);
    }
}
```

The program prints

```
First 'Lea' under index 2
Last  'Lea' under index 5
Does the list contain 'Sue': true
Does the list contain 'Bea': false
Size of list: 6
With get():   Sue Amy Lea Ann Kim Lea
Another way:  Sue Amy Lea Ann Kim Lea
now iterator: Sue Amy Lea Ann Kim Lea
Sorted:       [Amy, Ann, Kim, Lea, Lea, Sue]
Sublist:      [Ann, Kim, Lea]
Hash set:     [Ann, Sue, Lea, Amy, Kim]
Tree set:     [Amy, Ann, Kim, Lea, Sue]
Does the tSet contain 'Sue': true
tSet now:     [Amy, Ann, Kim, Sue, Zoe]
```

Note, that iteration over a **TreeSet** collection gives its elements in alphabetical order, what corresponds to the natural order of **String**s. This is not true for a **HashSet**. Note also various forms of iterations over **List**s, in particular the *for-each* loop (that we already used for arrays).

Now maps (i.e., collections implementing the **Map** interface, but not **Collection**):

Listing 66 HUK-SimpleMap/ASimpleMap.java

```java
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import java.util.TreeMap;

public class ASimpleMap {
    public static void main(String[] args) {
        Map<String,Integer> map = new TreeMap<>();
        map.put("Sue",167);
        map.put("Ann",173);
        map.put("Lea",170);
        map.put("Kim",173);

        Iterator<String> iter = map.keySet().iterator();
        while (iter.hasNext()) {
            String key = iter.next();
            int    val = map.get(key); // auto(un)boxing
```

```
            System.out.print(key + ": " + val + "  ");
            if (val == 170) iter.remove();
        }

        System.out.print("\nWe can print a map: " + map);

          // returns null, or the value if present
        map.putIfAbsent("Lea",169);
          // returns old value or null if not present
        map.replace("Lea",170);

        System.out.print("\nIs there a key 'Lea'? " +
                              map.containsKey("Lea"));
        System.out.print("\nIs there a key 'Zoe'? " +
                              map.containsKey("Zoe"));
          // inefficient!
        System.out.println("\nIs any girl 170 cm tall? " +
                                   map.containsValue(170));

        Integer was = map.remove("Ann");
        if (was != null)
            System.out.println("Ann removed, she was " +
                was + " cm tall");
        was = map.remove("Zoe");
        if (was == null)
            System.out.println("There was no 'Zoe'!");

        System.out.println("getOrDefault: 'Zoe'->" +
                         map.getOrDefault("Zoe",-1));
        System.out.println("getOrDefault: 'Lea'->" +
                         map.getOrDefault("Lea",-1));

        System.out.print("Iterating over 'keySet': ");
        for (String s : map.keySet())
            System.out.print(s + "->" + map.get(s) +"  ");

        System.out.print("\nMore efficient way: ");
        for (Map.Entry<String,Integer> e : map.entrySet())
            System.out.print(e.getKey() + "->" +
                             e.getValue() + "  ");
        System.out.println();
    }
}
```

The program prints

```
Ann: 173  Kim: 173  Lea: 170  Sue: 167
We can print a map: {Ann=173, Kim=173, Sue=167}
Is there a key 'Lea'? true
Is there a key 'Zoe'? false
```

```
Is any girl 170 cm tall? true
Ann removed, she was 173 cm tall
There was no 'Zoe'!
getOrDefault: 'Zoe'->-1
getOrDefault: 'Lea'->170
Iterating over 'keySet': Kim->173  Lea->170  Sue->167
More efficient way: Kim->173  Lea->170  Sue->167
```

Note, that **Map**s themselves are *not* iterable; however, a map's key set (returned by the **keySet** method), being a **Set**, i.e., a **Collection**, *is* iterable. The same applies to the **Set** of type **Map.Entry** objects returned by the **entrySet** method — each of such objects represents one (key, value) pair, of which both components may be accessed separately by the methods **getKey** and **getValue**.

7.5 Importance of equal and hashCode methods

Class **Object** defines — among others (in particular

`public String toString()`

that we already know) two other important methods:

- `public boolean equals(Object ob)` — which is used to check if two objects are, according to some criterion, equal: for two references `ob1` and `ob2`, the expression `ob1.equals(ob2)` compares the objects and yields **true** or **false**. But according to what criterion? In class **Object** there is no data to be compared, so the default implementation just compares addresses of the two objects. Very often, this is *not* what we want. When we have two objects of class **Person** and these persons have identical names, dates of birth, passport numbers etc., we rather want to consider the two objects as representing exactly the same person, in other words we want to consider these two objects equal. To get this behavior, we thus have to redefine (override) **equals** in our class.
  The **equals** method should implement an equivalence relation on non-null object references. This means that for any non-null references `a`, `b` and `c` `a.equals(a)` should always be **true** (reflexivity), `a.equals(b)` should have the same value (**true** or **false**) as `b.equals(a)` (symmetry), and if `a.equals(b)` and `b.equals(c)` are **true** then `a.equals(c)` must also be **true** (transitivity).
  Reflexivity and symmetry are usually obvious, but transitivity — not always. Suppose, we consider two two-element sets equal if they have at least one common element. Then $A = \{a, b\}$ is equal to $B = \{b, c\}$ and B is equal to $C = \{c, d\}$, but A and C have no common element and are not equal.
- `public int hashCode()` — which is used to calculate the so called **hash code** of an object. This is necessary if we want to put objects of our class in collections implemented as hash tables (e.g., **HashSet** or keys in a **HashMap**). The implementation of **hashCode** method from the **Object** class uses just the address of the object to calculate its hash code. However, very often this is not desirable as it would lead to situations when two objects that are considered equal (according to **equals**) have different hash codes: as a consequence the collections of such objects would be invalidated. Therefore, we have to override also this method remembering to do it consistently: whenever two objects are equal according to **equal**, their hash codes should be exactly the same.

The following example illustrates **toString** and **equals** methods:

Listing 67 AAR-EquToString/EquToString.java

```java
public class EquToString {
    public static void main(String[] args) {
        PersonGood johny = new PersonGood("John",1980);
        PersonGood john  = new PersonGood("John",1980);
        PersonBad  billy = new PersonBad("Bill",1980);
        PersonBad  bill  = new PersonBad("Bill",1980);

        if (johny.equals(john))
            System.out.println("johny == john");
        else
            System.out.println("johny != john");

        if (billy.equals(bill))
            System.out.println("billy == bill");
        else
            System.out.println("billy != bill");

        System.out.println("johny: " + johny);
        System.out.println("billy: " + billy);
    }
}

class PersonBad {
    private String name;
    private    int byear;
    PersonBad(String n, int y) {
        name  = n;
        byear = y;
    }
    public String getName() { return name;  }
    public    int getYear() { return byear; }
}

class PersonGood {
    private String name;
    private    int byear;
    PersonGood(String n, int y) {
        name  = n;
        byear = y;
    }

    public String getName() { return name;  }
    public    int getYer()  { return byear; }

    @Override
    public String toString() {
        return name + "(" + byear + ")";
    }
```

```java

    @Override
    public boolean equals(Object ob) {
        if (ob == null || getClass() != ob.getClass()) return false;
        PersonGood p = (PersonGood)ob;
        return name.equals(p.name) && byear == p.byear;
    }
}

```

As we can see, with **toString** and **equals** methods *not* overriden (in class **PersonBad**), the program works but uses the versions of these methods from class **Object**. To get the expected results, we have to override these methods, as we did in class **PersonGood**.

The next example illustrates the effect of overriding the **hashCode** method:

Listing 68 HUM-HashEquals/Person.java

```java
public class Person {

    private String name;
    private String idNumber;

    public Person(String name, String idNumber) {
        this.name     = name;
        this.idNumber = idNumber;
    }



    @Override
    public boolean equals(Object other) {
        if (other == null ||
            getClass() != other.getClass()) return false;
        Person p = (Person)other;
        return idNumber.equals(p.idNumber) &&
                  name.equals(p.name);
    }



    @Override
    public int hashCode() {
        return 17*name.hashCode() + idNumber.hashCode();
    }


    @Override
    public String toString() {
        return name + "(" + idNumber + ")";
```

```
    }
}
```

with **main** as below

Listing 69 HUM-HashEquals/AHash.java

```
import java.util.HashMap;
import java.util.Map;

public class AHash {
    public static void main(String[] args) {
        Map<Person,String> map = new HashMap<>();

        map.put(new Person("Sue","123456"),"Sue");

          // new object, but should be equivalent to
          // the one which has been put into the map
        Person sue = new Person("Sue","123456");

        if (map.containsKey(sue))
            System.out.println(sue + " has been found");
        else
            System.out.println(sue + " has NOT been found");
    }
}
```

As can be easily checked, the program will print

```
Sue(123456) has been found
```

only when both **hashCode** and **equals** are consistently overridden in the class of keys of the map (i.e., **Person**).

Section 8

Streams

8.1 Introduction

Streams provide an abstraction for dealing with collections of values (which do not have to correspond to 'real' collections) and specifying what you want to be done, leaving the details of how to do it to the library. For example, you can create a stream of object of type **Person** and say something like *select only women of age in the range* $[35, 60]$ *and form a map with country of origin as keys and lists of women form these countries as values.* Or, having an infinite (*sic!*) stream of **double**s, you can say *select only positive numbers, round them to integers, calculate their squares, take 1000 first elements and give me their arithmetic average.* In addition, you can often say something like *take my stream, do this and this and try to do it using as many separate threads as necessary to ensure maximum efficiency.*

Streams can be created from collections, arrays, or using generators or iterators. However, themselves, they are *not* collections. Rather, they represent streams of individual pieces of data that can be processed one by one, not necessarily storing anywhere and anytime all of them together.

- Usually, streams do not store their elements anywhere, they take it one by one from a source. However, for some operations they have to store the elements internally, for example in order to sort them.
- Streams do not modify the source of their elements, they can transform them and yield transformed values in a new stream.
- Operations on streams are 'lazy', i.e. only those operation that are needed to get the desired result are actually executed.

In order to use streams, you have to

- create a stream — you can require it to be parallelized if it makes sense for a problem at hand;
- apply zero, one or more *intermediate operations*; each of them yields another stream to which one can apply a next intermediate operation, thus forming the so called *pipeline*;
- apply a *terminal operation* that produces a result. Only a terminal operation triggers all the other (intermediate) operations; before that no processing takes place. This is why streams can be lazy — when it is known what result is expected, it is known which operations are necessary and which are not. After applying a terminal operation, the stream is deemed 'consumed'; its is closed and cannot be reused.

Some operations, both intermediate and terminal, can be *short-circuiting*: an intermediate operation is short-circuiting if for an infinite stream it may produce a finite stream, a terminal operation is short-circuiting if for an infinite stream it may terminate in finite time. Of course, for infinite streams, there must be at least one short-circuiting operation in the whole pipeline of operations.

Intermediate operations can be stateless or stateful. Most of them are stateless — elements are processed one by one and to process a given element no knowledge of elements seen previously, or those to be seen later, is needed. Some operations, however, are stateful — at least some information on the previous elements has to be remembered. There are four such operations in the library: **sorted**, **distinct**, **limit** and **skip**. For obvious reasons, the stateful operations cannot be applied to infinite streams and are less efficient.

In the examples below, we will use references to methods, which can be used instead of lambdas — they are covered in more detail in sec. 9 on p. 115.
We will also refer to functional interfaces that are already defined in the library: more about them in sec. 10 on p. 118.

8.2 Creating streams

Streams can be created from any collection by invoking its **stream** method

```
collection.stream()
```

which produces a stream of type **Stream<T>**, where **T** is the type of elements of the collection. There is also a static function **of** in the **Stream** class. It takes any number of arguments, or an array, and creates a stream:

```
Stream<String> s1 = Stream.of("Alice", "Cindy", "Kate");
Stream<String> s2 = Stream.of(new String[]{"A","B"});
```

Another static function of **Stream** is **generate**. It takes a **Supplier** and produces a stream by using the supplier to create (potentially infinite) stream of values

```
Stream<Double> d = Stream.generate(Math::random);
```

One can also use **iterate** which takes a 'seed' and a **UnaryOperator** and repeatedly applies the operator to the previous element with seed as the first one, so the resulting stream will be
(seed, op.apply(seed), op.apply(op.apply(seed)), . . .)
For example:

```
Stream.iterate(1, n -> 2*n).limit(7).forEach(System.out::println);
```

will print the sequence $(1, 2, 4, 8, 16, 32, 64)$; note that we had to limit the number of elements in the stream, as **iterate** generates potentially infinite sequence of numbers.

Functions **generate** and **iterate** may also be used with primitive types; the type of the stream will then be **DoubleStream**, **LongStream** or **IntStream**.

Streams of primitive type, **IntStream** and **LongStream** can also be created by calling **range** and **rangeClosed**:

```
IntStream s = IntStream.range(int startIncl, int endExcl)
IntStream s = IntStream.rangeClosed(int startIncl, int endIncl)
```

which return an ordered stream of **int**s from startIncl (inclusive) to endExcl (exclusive) for the first form or to endIncl (inclusive) in the second form, by an incremental step of 1 (and analogously for **long**s).

Methods creating streams have also been added to some other classes, for example **Files** from ***java.nio.file***. Its static method **lines** yields a stream of lines of a file as **String**s:

```java
try (Stream<String> str = Files.lines(path,charset)) {
    // str is a stream of lines as strings
}
```

We use try-with-resources here to be sure that when the stream is closed, the underlying file is also closed (if your charset is UFT-8, it can be omitted, as this is the default).

There is also, in class **Pattern** from ***java.util.regex***, the method **splitAsStream** which yields a stream of **String**s resulting from splitting a text (actually, a **CharSequence**) with a given regex specifying the separator:

```java
Stream<String> words =
    Pattern.compile("\\P{L}+").splitAsStream(text);
```

will yield a stream of words from text (separated by non-empty sequences of non-letters).

You can find examples of stream creation in listings that follow (in particular, Listing 72 on page 105).

8.3 Intermediate operations

Let us enumerate the most useful intermediate operations (implemented as methods of class **Stream** from the package ***java.util.stream***). In what follows, the symbol **T** denotes the type of elements in a stream on which the methods are invoked (its type is therefore **Stream<T>**, while **R** stands for another type, where applicable.

- `Stream<T> filter(Predicate<T> pred)` — produces a new stream where only elements that satisfy the predicate from the original stream are retained. Examples: Listing 71, Listing 76, Listing 77, Listing 79.
- `Stream<R> map(Function<T,R> fun)` — applies the function to each element of the input stream and produces a new stream of values obtained. There are similar operations with a primitive type instead of **R**: **mapToInt**, **mapToLong** and **mapToDouble**. Examples: Listing 70, Listing 71, Listing 72, Listing 73, Listing 77, Listing 78.
- `Stream<T> distinct()` — produces the same stream as the input one but with all duplicates removed (they need not to be on subsequent positions). To compare elements, method **equal** is used. Stateful (and rather expensive) operation. Examples: Listing 72, Listing 79.
- `Stream<R> flatMap(Function<T,Stream<R>> fun)` — applies **fun** to all elements to get stream of streams; then 'flattens' these streams into one stream of all elements from the individual streams. There are specialized versions for primitive types instead of **R**: **flatMapToInt**, **flatMapToLong** and **flatMapToDouble** which return **IntStream**, **LongStream** and **DoubleStream**, respectively. Example: Listing 79.
- `Stream<T> sorted(Comparator<T> cmp)` — yields the same stream but sorted by using the given comparator cmp. There is also version without any arguments — then the natural order of type **T** is used. Stateful operation. See also the documentation of **Comparator** from ***java.util*** for various ways of creating comparators; an example is shown in Listing 79. Other examples include Listing 72 and Listing 77.
- `Stream<T> peek(Consumer<T> cons)` — returns the same stream but executes **cons** on each element 'on the fly'. Used often for debugging. Examples: Listing 77, Listing 70, Listing 71.

- `Stream<T> limit(long lim)` — returns the same stream but limited to at most lim first elements. Stateful, short-circuiting operation. Example: Listing 70.
- `Stream<T> skip(long skp)` — returns the same stream but with first skp elements suppressed. Stateful operation.

8.4 Terminal operations

Zero, one or more transformations by intermediate operations must be followed by exactly one final (terminal) operation; otherwise no operation at all would be actually executed as this is a terminal operation that forces the execution of all (lazy) operations that precede it. Terminal operation yields a result — this can be one value (in particular, a number) or a collection (or array) of elements.

Many terminal operations can be created by stream method **collect(Collector)**, where a collector can be chosen from the rich set of ready-to-use collectors returned by static factory methods defined in class **Collectors**.

There are several terminal operations yielding a number as their result. For example (**T** denotes the type of elements in the stream, **NumType** stands for **Int**, **Long** or **Double**):

- `long count()` — returns the number of elements in the input stream;
- `Optional<T> min(Comparator cmp)`, `Optional<T> max(Comparator cmp)` — return minimum and maximum elements of the stream using a provided comparator. For primitive-type streams no comparator is needed and the return type is **OptionalInt**, **OptionalLong** or **OptionalDouble**. To get minimum or maximum element, one can also use **collect** with a collector returned by invoking **minBy(Comparator)** or **maxBy(Comparator)** from class **Collectors**. The optional returned is empty, if stream was empty.
- `NumType sum()` — returns the sum of elements for primitive-type streams. Similarly, a collector returned by **summingType(ToTypeFunction)** can be used for non-primitive type (**Type** is **Int**, **Long** or **Double**). See examples in Listing 72 and Listing 77.
- `Optional<Double> average()` — returns the mean (arithmetic average) for streams of primitive types. A collector returned by **averagingType(ToTypeFunction)** can be used for non-primitive types (**Type** is **Int**, **Long** or **Double**).

For primitive streams, there are also terminal operations **summaryStatistics**, which return an object of type **TypeSummaryStatictics**, where **Type** is **Int**, **Long** or **Double**. This object may then be queried for the number of elements, their sum, average, minimum and maximum. For non-primitive types one can use a collector returned by **summarizingType(ToTypeFunction)**. Very often, the **ToTypeFunction** function will be just the reference to a method returning a number. See an example in Listing 73.

Another group of terminal operations are those returning collections (or arrays). Let us mention some of them

- `Object[] toArray()` — returns all elements of the stream as an array (of type **Object[]**). See example in Listing 78.
- `R[] toArray(IntFunction<R[]> gen)` — returns an array of type **R[]**; gen is a function taking an **int** and creating an array of this size. Most often this will be just the reference to an array constructor **R[]::new**. See examples in Listing 71 and Listing 76.

- collector retuned by `groupingBy(Function<T,K> keyExtr)` — produces a map of type **Map<K,List<T>>**. It applies the function **keyExtr** to each element of the stream and treats the obtained value as a key in the resulting map; lists of elements yielding the same key will be the values of the map (there are others, very powerful versions of this collector). See simple examples in Listing 74 and Listing 75.

A very useful terminal operation just 'consumes' the stream

```
void forEach(Consumer<T> cons)
```

by invoking cons on each element; a very common example is just printing:
`stream.forEach(System.out::println)`.
One can also create a **String** by concatenating strings obtained from subsequent elements of a stream; such a collector can be created by factory method

```
String stream.collect(Collectors.joining())
```

(a desired delimiter may also be passed as the argument). See examples in Listing 72, Listing 73, Listing 77 and Listing 79.

There are also short-circuiting operations taking a **Predicate** and returning **boolean** — **allMatch** (whether all elements satisfy the given predicate), **anyMatch** (whether at least one satisfies it) and **noneMatch** (whether all elements do not satisfy the predicate). They are short-circuiting because the stream is deemed consumed (and processing stops) when the result is already known. For example, in the case of **allMatch**, when an element *not* satisfying the predicate is encountered, the answer is **false** and cannot change, so the processing stops. See example in Listing 72.

A very important and versatile reduction of a stream to a single value can be obtained by invoking **reduce** (the operation performed by **reduce** is called *left fold*). The operation has a few variants. The basic one looks like this

```
T reduce(T iden, BinaryOperator<T> acc)
```

Here, acc is a **BinaryOperator<T>** acting on two values of type **T** (type of elements of the stream). First the operator is applied to iden and the first element of the stream, then on the result and the second element, than again on the result and the third, and so on. Basically, this is equivalent to

```
T result = iden;
for (T e : elements of the stream)
    result = acc.apply(result, e)
return result;
```

There are two important conditions, though:

- iden must be an identity of the operation, i.e.,
  `apply(iden,e)`
  must return e (like number 0 for addition or 1 for multiplication);
- operator **apply** must be associative, i.e.,
  `apply(apply(e,f),g) = apply(e,apply(f,g))`
  must always hold.

Let us suppose that we have a stream str of **Integers**. Then we could use **reduce** to find the sum or product of all elements, or their count, like this

```
  // sum
Integer a = str.reduce(0, Integer::sum);
  // product
Integer a = str.reduce(1, (a,e) -> a*e);
  // count
Integer a = str.reduce(0, (a,e) -> a+1);
```

There are also other versions of **reduce** (without iden, but returning **Optional** — see documentation).

Streams can be parallelized. You either get a parallel stream directly from a collection

```
Stream<T> parellelStream = collection.parallelStream();
```

or parallelize an existing stream

```
Stream<T> parallelStream = sequentialStream.parallel();
```

You can also make a parallel stream sequential

```
Stream<T> sequentialStream = parallelStream.sequential();
```

Parallel streams allow the compiler to divide operations on the stream into parts that can be executed on different threads. Of course, the final result will have to be somehow combined from the partial results — this is not always trivial! It is your responsibility to ensure that operations performed on different threads do not lead to data races or deadlocks.

8.5 Examples

The example below uses **limit**, **peek**, **map** and **forEach** terminal operation. It demonstrates the 'laziness' of streams: only those elements which are necessary to get the result are actually processed. Also, a reference to a constructor is used:

Listing 70 LDC-Lazy/Lazy.java

```java
import java.util.stream.Stream;
import java.io.File;

public class Lazy {
    public static void main (String[] args) {
        Stream.of("Alice", "Bella", "Cecilia", "Dorothy")
         .peek(e -> System.out.print("peek: " + e + "; "))
         .map(s -> s + ".txt")
         .map(File::new)
         .limit(2)
         .forEach(f -> System.out.println(f + " exists? " +
                                (f.exists() ? "Yes" : "No") ));
    }
}
```

The output is interesting:

```
peek: Alice; Alice.txt exists? No
peek: Bella; Bella.txt exists? Yes
```

As we can see, the first element (`"Alice"`) started its “journey” and went all the way down to the **forEach** before the next element (`"Bella"`) even started! After two first elements (`"Alice"` and `"Bella"`) finished, the **limit(2)** “says” *that's enough* and no other element even starts its journey along the chain of operations — `"Cecilia"` and `"Dorothy"` don't even reach **peek**! This nicely illustrates the laziness of streams: only what is necessary to get the final result will be really executed.

The example below demonstrates the function **Files.lines** which creates a stream of lines of a given file which then can be transformed and reduced to a desired result (in this case, a list). It also demonstrates method references, as well as methods **filter**, **peek**, **map** and **toList** and **forEach** terminal operations.

Listing 71 LDF-StreamGrep/StreamGrep.java

```java
import java.util.List;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.stream.Collectors;
import java.util.stream.Stream;
import static java.nio.charset.StandardCharsets.UTF_8;

public class StreamGrep {
    public static void main(String[] args) {
        List<String> list = null;
        try (Stream<String> lines =
                Files.lines(Paths.get("StreamGrep.java"),
                            UTF_8)) {
            String substr = "String";
            list = lines
                      // Predicate expected
                    .filter(s -> s.indexOf(substr) >= 0)
                      // Consumer expected
                    .peek(System.out::println)
                    .collect(Collectors.toList());
        } catch(IOException e) { return; }
        System.out.println("and now the list...");
        list.stream()
                // reference to method
            .map(String::toUpperCase)
            .forEach(System.out::println);
    }
}
```

The program prints

```
    public static void main(String[] args) {
        List<String> list = null;
```

```
            try (Stream<String> lines =
                String substr = "String";
                .map(String::toUpperCase)
    and now the list...
        PUBLIC STATIC VOID MAIN(STRING[] ARGS) {
            LIST<STRING> LIST = NULL;
            TRY (STREAM<STRING> LINES =
                STRING SUBSTR = "STRING";
                .MAP(STRING::TOUPPERCASE)
```

A stream of lines of a file can also be obtained from **BufferedReader**, as the example below demonstrates. It also shows streams of primitive type (e.g., **int**), various ways of creating streams, functions **sorted**, **distinct**, **filter**, **map**, **mapToInt**, **peek** and **sum**, **joining**, **allMatch** and **forEach** terminal operations. The last part of the program shows how to obtain a stream from a regex.

Listing 72 LDD-StreamMisc/StreamMisc.java

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
import java.util.stream.Stream;
import java.util.regex.Pattern;

public class StreamMisc {
    public static void main(String[] args) {
        System.out.println("*From an arrray...");
        String[] ws = {"To", "be", "or", "not", "to", "be"};
        Stream.of(ws)
          .map(String::toLowerCase)
          .distinct()
          .sorted()
          .forEach(e -> System.out.print(e + " "));
        System.out.println();

        System.out.println("*From varargs...");
        System.out.println(
            Stream.of("To", "be", "or", "not", "to", "be")
              .collect(Collectors.joining(" - "))
        );

        System.out.println("*From a collection...");
        List<String> list = Arrays.asList(
```

```
                        "1","10","100","1000","10000","100000");
        System.out.println("Sum = " +
            list.stream().mapToInt(Integer::parseInt).sum()
        );

          // generating a stream by iterating a unary
          // function starting from a given seed
        System.out.println("*From a generator...");
        ArrayList<Integer> arri = new ArrayList<>();
        IntStream.iterate(17, n -> n%2 == 0 ? n/2 : 3*n+1)
         .peek(arri::add)          // arri in closure,
         .allMatch(n -> n != 1);   // allMatch is short-
                                   // circuited, so will
        System.out.println(arri); // stop iteration!

        System.out.println("*Lines of a file as stream...");
        try (BufferedReader br = Files.newBufferedReader(
                        Paths.get("StreamMisc.java"),
                        StandardCharsets.UTF_8)) // default
        {
            br.lines()
             .filter(e -> e.contains("collect"))
             .forEach(System.out::println);
        } catch (IOException never_ignore_exceptions) { }

        System.out.println("*From a regex...");
        String s = "a is 1, b=3 and c:7 X";
        System.out.println("Sum of extracted numbers is " +
                Pattern.compile("\\D+")
                 .splitAsStream(s)
                 .filter(e -> e.length() > 0)
                 .mapToInt(Integer::parseInt).sum());
    }
}
```

The program prints

```
*From an arrray...
be not or to
*From varargs...
To - be - or - not - to - be
*From a collection...
Sum = 111111
*From a generator...
[17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
*Lines of a file as stream...
             .collect(Collectors.joining(" - "))
        System.out.println("*From a collection...");
             .filter(e -> e.contains("collect"))
*From a regex...
```

```
Sum of extracted numbers is 11
```

Primitive-type streams are also demonstrated in the example below; note the **summaryStatistics** terminal operation and the **generate** supplier. Also, functions **sorted**, **limit** and **map** are used here (as well as method references).

Listing 73 LDG-MethRefStr/MethRefStr.java

```java
import java.util.Random;
import java.util.stream.DoubleStream;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class MethRefStr {
    public static void main(String[] args) {
        Random r = new Random(); // effectively final
        System.out.println(
                  // Supplier of double's expected
                DoubleStream.generate(r::nextGaussian)
                  // we want just ten million numbers
               .limit(10_000_000)
                  // reduction to DoubleSummaryStatistics
               .summaryStatistics()
                  // arithmetic average of all numbers
               .getAverage());

        System.out.println(
            Stream.of(new Person("C"), new Person("A"),
                      new Person("D"), new Person("B"))
                    // Function<Person,otherType> expected
                  .map(Person::getName)
                  .sorted()
                    // Function<String,otherType> expected
                  .map(String::toLowerCase)
                    // reduction to a single String
                  .collect(Collectors.joining("-")));

        Thread t = new Thread(MethRefStr::fibos);
        t.start();
        try {
            t.join();
        } catch(InterruptedException ignore) { }
    }

    public static void fibos() {
        StringBuilder sb = new StringBuilder("0, 1");
        int a = 0, b = 1;
        for (int i = 0; i < 8; ++i) {
            b += a;
            a  = b - a;
```

```
            sb.append(", " + b);
        }
        System.out.println(sb);
    }
}

class Person {
    private String name;
    public Person(String n) { name = n; }
    public String getName() { return name; }
}
```

The program prints (the first number may be different)

```
-1.9002508699231762E-4
a-b-c-d
0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

The next example demonstrates the use of, extremely useful, **groupingBy** terminal operation. It comes in many variants; below, the simplest of them is used. It takes a function, which applied to elements of the stream will yield a value which will be then used as the key of the map: values of this map will be lists of elements that yield this key:

Listing 74 LDE-Grouping/Grouping.java

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class Grouping {
    public static void main (String[] args) {
        List<Person> list = Arrays.asList(
                new Person("John",  "UK"),
                new Person("Mary",  "US"),
                new Person("Xue",   "CH"),
                new Person("Kate",  "UK"),
                new Person("Janek", "PL"),
                new Person("Cindy", "US"),
                new Person("Bao",   "CH"),
                new Person("Kasia", "PL")
        );

          // collect gives Map<String,List<Person>>
          // groupingBy expects Function...
        list
         .stream()
         .collect(Collectors.groupingBy(Person::getCountry))
         .entrySet()
         .stream()
```

```java
            .forEach(e -> System.out.println(e.getKey() +
                                        " -> " + e.getValue()));
    }
}

class Person {
    private final String name;
    private final String country;
    public Person(String n, String c) {
        name = n; country = c;
    }
    public String getName()    { return name;    }
    public String getCountry() { return country; }
    @Override
    public String toString() {
        return name + " (" + country + ")";
    }
}
```

The program prints

```
CH -> [Xue (CH), Bao (CH)]
UK -> [John (UK), Kate (UK)]
PL -> [Janek (PL), Kasia (PL)]
US -> [Mary (US), Cindy (US)]
```

A very similar example is given below: here, we add the second argument (a collector) to the **groupingBy** function. This second collector determines what to do with elements corresponding to the same key — here, we just count them

Listing 75 LDJ-GroupCount/GroupCount.java

```java
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class GroupCount {
    public static void main (String[] args) {
        List<Person> list = Arrays.asList(
            new Person("John",  "UK"), new Person("Ewa",   "PL"),
            new Person("Mary",  "US"), new Person("Xue",   "CH"),
            new Person("Kate",  "UK"), new Person("Jane",  "UK"),
            new Person("Janek", "PL"), new Person("Cindy", "US"),
            new Person("Bao",   "CH"), new Person("Kasia", "PL")
        );

          // collect gives Map<String, Long>
        list
         .stream()
         .collect(Collectors.groupingBy(
```

```java
                        Person::getCountry,
                        Collectors.counting()))
             .entrySet()
             .stream()
             .forEach(e -> System.out.println(e.getKey() +
                                 " -> " + e.getValue()));
    }
}

class Person {
    private final String name;
    private final String country;
    public Person(String n, String c) {
        name = n; country = c;
    }
    public String getName()    { return name;    }
    public String getCountry() { return country; }
    @Override
    public String toString() {
        return name + " (" + country + ")";
    }
}
```

The program prints

```
CH -> 2
UK -> 3
PL -> 3
US -> 2
```

The example below demonstrates combining predicates, and also **filter** function:

Listing 76 LDB-Predicates/Predicates.java

```java
import java.util.List;
import java.util.function.Predicate;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Predicates {
    public static void main (String[] args) {
       Predicate<Integer> p1 = e -> e%2 ==  0;
       Predicate<Integer> p = p1
                              .and(e -> e <= 10)
                              .or(e -> e == 19);
       List<Integer> filteredList =
           Stream.of(1,2,3,4,19,22,12)
          .filter(p)
          .collect(Collectors.toList());
       filteredList.stream().forEach(System.out::println);
```

```
        // prints 2, 4, 19
    }
}
```

Several intermediate and terminal operations from previous examples are used below:

Listing 77 LDA-Streams/Streams.java

```
import java.util.Collections;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
import java.util.stream.Stream;

enum HairColor { BLACK, BROWN, BLOND, RED, WHITE };
enum EyeColor  { AMBER, BLUE, BROWN, GRAY, GREEN, HAZEL };
enum Sex       { WOMAN, MAN };

public class Streams {
    public static void main(String... aargs) {
        System.out.println("*Sorting by length of name");
        Stream.of("Alice", "Margot", "Mary", "Sue")
           .sorted((a,b) ->
                    Integer.compare(a.length(),b.length()))
           .forEach(System.out::println);

        System.out.println("*Summing ints...");
        int sum = IntStream.of(3,2,9,12,8,4)
           .filter(n -> n%2 == 0)
           .map(n -> 3*n+1)
           .sorted()
           .peek(n -> System.out.print(n+" "))
           .sum();
        System.out.println("\nSum = " + sum);

        List<Person> listp = Arrays.asList(
            new Person("Ann",Sex.WOMAN,
                       HairColor.BLOND,EyeColor.BLUE),
            new Person("Joe",Sex.MAN,
                       HairColor.BLACK,EyeColor.BROWN),
            new Person("Sue",Sex.WOMAN,
                       HairColor.RED,EyeColor.HAZEL),
            new Person("Ben",Sex.MAN,
                       HairColor.BROWN,EyeColor.GREEN),
            new Person("Bea",Sex.WOMAN,
                       HairColor.WHITE,EyeColor.GRAY)
        );

```

```
        System.out.println("*Women's names...");
        String womenNames =
            listp.stream()
                 .filter(p -> p.getSex() == Sex.WOMAN)
                 .map(Person::getName)
                 .collect(Collectors.joining(", "));
        System.out.println("Women: " + womenNames);

        System.out.println("*Counting men...");
        long menCount =
            listp.stream()
                 .filter(p -> p.getSex() == Sex.MAN)
                 .count();
        System.out.println("No. of men: " + menCount);

        System.out.println("*Names staring with 'B'");
        String nameB =
            listp.stream()
                 .filter(p -> p.getName().charAt(0) == 'B')
                 .map(Object::toString)
                 .collect(Collectors.joining("\n"));
        System.out.println(nameB);
    }
}

class Person {
    private final String    name;
    private final Sex       sex;
    private final HairColor hairColor;
    private final EyeColor  eyeColor;
    public Person(String n, Sex g,
                  HairColor hc, EyeColor ec) {
        name      =  n;
        sex       =  g;
        hairColor = hc;
        eyeColor  = ec;
    }
    public String getName() { return name; }
    public Sex    getSex() { return sex; }
    public HairColor getHairColor() { return hairColor; }
    public EyeColor getEyeColor() { return eyeColor; }

    @Override
    public String toString() {
        return (sex == Sex.WOMAN ? "Mrs " : "Mr ") +
            name + " (hair:" + hairColor + ", eyes:" +
            eyeColor + ")";
    }
}
```

The program prints

```
*Sorting by length of name
Sue
Mary
Alice
Margot
*Summing ints...
7 13 25 37
Sum = 82
*Women's names...
Women: Ann, Sue, Bea
*Counting men...
No. of men: 2
*Names staring with 'B'
Mr Ben (hair:BROWN, eyes:GREEN)
Mrs Bea (hair:WHITE, eyes:GRAY)
```

Next example demonstrates **map**, references to a constructor (also, to a 'constructor' of an array) and the **toArray** terminal operation.

Listing 78 LDH-RefConstr/RefConstr.java

```java
import java.util.Arrays;
import java.util.stream.Stream;

public class RefConstr {
    public static void main (String[] args) {
        String[] names = {"Ada", "Bea", "Sue", "Lea" };
        Person[] persons =
            Stream.of(names)
             .map(Person::new)
             .toArray(Person[]::new); // otherwise Object[]
        System.out.println(Arrays.toString(persons));
    }
}

class Person {
    private String name;
    Person(String n)        { name = n; }
    public String getName() { return name; }
    @Override
    public String toString(){ return "Miss " + name; }
}
```

The program prints

```
[Miss Ada, Miss Bea, Miss Sue, Miss Lea]
```

The final example demonstrates **flatMap**, which 'flattens' a stream of streams into one stream, consisting of all elements from these individual streams:

Listing 79 LDI-FlatMap/Flat.java

```java
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.Comparator;
import java.util.stream.Collectors;
import java.util.stream.Stream;
import static java.nio.charset.StandardCharsets.UTF_8;

public class Flat {
    public static void main(String[] args) {
        String result = null;
        try (Stream<String> stream =
                Files.lines(Paths.get("Flat.java"),UTF_8))
        {
            result =
                stream
               .flatMap(l -> Stream.of(l.split("\\P{L}+")))
               .filter(w -> w.length() > 9)
               .distinct()
               .sorted(Comparator.comparing(
                                String::length).reversed())
               .collect(Collectors.joining(", "));
        } catch(IOException ignore) { }
        System.out.println(result);
    }
}
```

The program prints

```
StandardCharsets, IOException, Comparator, Collectors
```

Section 9

Method references

Quite often, when we override a method of a functional interface, its implementation reduces to invoking another method that already exists in a class. In such situations we can pass the reference to this method and the compiler will do the rest all by itself. For example, suppose we have a stream stream of objects of type **AClass**; we can terminate the pipeline of operations on the stream with **forEach** which then expects an object implementing **Consumer<AClass>**. Suppose, we just want to print elements of the stream — we can achieve this by writing

```
stream.forEach(e -> System.out.println(e))
```

or, in an abbreviated form, we just pass a method reference

```
stream.forEach(System.out::println)
```

In the latter case, we are telling the compiler *take elements of the stream and pass them, one by one, to the method indicated as an argument.* Notice, that in a method reference the name of the method must be preceded by a double colon and there are no parentheses after the name, because we don't invoke this function here; it is just a reference to the method itself. As **println** is a non-static method, it must be called on an object, so in front of the method reference we specify the object it is to be invoked on (in this case it is System.out).

There are three main forms of method references:

- anObject::nonstaticMethod
- AClass::staticMethod
- AClass::nonstaticMethod

In each case arguments for methods are needed, and also, in the third case, an object which will be the receiver of the invocation.

In the first case, anObject::nonstaticMethod, arguments will be passed as the argument to the method, so it is essentially equivalent to a lambda

```
e -> anObject.nonstaticMethod(e)
(e,f) -> anObject.nonstaticMethod(e,f)
```

depending of the number of arguments expected. Note that there can be several overloaded versions of the method: compiler will select the one matching the number and types of arguments passed. We have already seen an example: System.out::println corresponds to

```
e -> System.out.println(e)
```

In the second case, we can use a static method if the number and type of arguments matches the number and type of arguments of the static method. For example, if **Function<Double,Double,Double>** (or **BinaryOperator<Double>**) is expected in a given context, we can pass just Math::pow; this will be equivalent to passing the the lambda

```
(x, y) -> Math.pow(x,y)
```

In the third case, the first argument becomes the target (receiver) of the method and the remaining arguments are passed to the method as arguments. For example, **String::compareToIgnoreCase** corresponds to

```
(x, y) -> x.compareToIgnoreCase(y).
```

Suppose you want to sort an array of **String**s (say, strings) ignoring the case. You can call **sort** which expects a **Comparator** that will be called with two arguments; you can pass the method reference

```
Arrays.sort(strings, String::compareToIgnoreCase)
```

Let us consider an example:

Listing 80 LCP-MethRefs/MethRefs.java

```java
import java.util.Arrays;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;

public class MethRefs {
    List<String> list = Arrays.asList(
                "Zoe", "kate", "Cindy", "barbra");
    public static void main(String[] args) {
        new MethRefs();
    }

    public MethRefs() {
          // case anObject::nonstaticMethod
          // Iterable<String> expected
        Iterable<String> iterObj = this::getIter;
        for (String s : iterObj) System.out.print(s + " ");
        System.out.println();

          // case AClass::staticMethod
          // Runnable expected
        Thread t = new Thread(MethRefs::tenFibos);
        t.start();
        try {
            t.join();
        } catch(InterruptedException ignore) { }

          // case AClass::nonstaticMethod
          // Comparator<String> expected
        Collections.sort(list, String::compareTo);
        System.out.println(list);
        Collections.sort(list, String::compareToIgnoreCase);
        System.out.println(list);
    }

    public static void tenFibos() {
```

```
            // prints first 10 Fibonacci numbers
        StringBuilder sb = new StringBuilder("0, 1");
        int a = 0, b = 1;
        for (int i = 0; i < 8; ++i) {
            b += a;
            a  = b - a;
            sb.append(", " + b);
        }
        System.out.println(sb);
    }

    public Iterator<String> getIter() {
        return list.iterator();
    }
}
```

which prints

```
Zoe kate Cindy barbra
0, 1, 1, 2, 3, 5, 8, 13, 21, 34
[Cindy, Zoe, barbra, kate]
[barbra, Cindy, kate, Zoe]
```

More examples can be found in Listing 71, Listing 73, Listing 74 and other listings from section 8 on p. 98.

Method references may also refer to constructors, even 'constructors' of arrays. The syntax is as follows:

- AClass::new
- AType[]::new

where in the second case **AType** may also be a primitive type (like **int**). In the first case, arguments are passed to a constructor, and its type decides which constructor will be used (if they are overloaded).
In the second case, there can be only one argument, a non-negative integer, which will be used as the size of the array created. There are also functions that accept a reference to an array constructor specifying type of the array to be created, which would otherwise be **Object[]**. For examples — seeListing 70 and Listing 78.

Section 10

Functional interfaces

Functional interfaces are those which declare *only one abstract* method, although they may contain definitions of default methods (marked with the keyword **default**) and static functions. When defining a functional interface, we should, although it is not strictly required, mark them with the annotation **FunctionalInterface** — the compiler will then check if the class definition actually defines a functional interface; e.g.,

```
@FunctionalInterface
interface Calc {
    double calculate(double d);
}
```

The standard library (in package ***java.util.function***) defines several functional interfaces. The definitions are generic, i.e., they are expressed in terms of type parameters which may correspond to different (object) types. There are also versions with primitive types — **int**, **long**, **double** and **boolean**.

Let us briefly mention functional interfaces from the standard library.

10.1 Consumers

Consumers represent operations which accept ('consume') one or two arguments but do not return anything; therefore they are used for their side effects.

Interface **Consumer<T>** declares one abstract method of type **void** taking one argument:

```
void accept(T t)
```

where **T** denotes any *object* type. There are also versions for arguments of primitive types: **int**, **long** or **double**

- **IntConsumer** $\Longrightarrow$ `void accept(int t)`
- **LongConsumer** $\Longrightarrow$ `void accept(long t)`
- **DoubleConsumer** $\Longrightarrow$ `void accept(double t)`

Interface **BiConsumer<T,U>** declares one abstract method of type **void** taking two arguments:

```
void accept(T t, U u)
```

where **T** and **U** denote any *object* types. There are also versions with one of the arguments, the *second*, of a primitive type: **int**, **long** or **double**

- **ObjIntConsumer<T>** $\Longrightarrow$ `void accept(T t, int u)`
- **ObjLongConsumer<T>** $\Longrightarrow$ `void accept(T t, long u)`
- **ObjDoubleConsumer<T>** $\Longrightarrow$ `void accept(T t, double u)`

10.2 Functions

Functions represent operations which accept one or two arguments of some (possibly different) types and return a value of a certain type, which may be different from types of arguments (return types are conventionally denoted by the letter R).

Interface **Function<T,R>** declares one abstract method taking one argument and returning a value

```
R apply(T t)
```

where **T** and **R** denote any *object* types. There are also versions for arguments of primitive types: **int**, **long** or **double**

- **IntFunction<R>** $\Longrightarrow$ `R apply(int t)`
- **LongFunction<R>** $\Longrightarrow$ `R apply(long t)`
- **DoubleFunction<R>** $\Longrightarrow$ `R apply(double t)`

Other versions take argument of an object type but return values of primitive types: **int**, **long** or **double** (note different names of their abstract methods!)

- **ToIntFunction<T>** $\Longrightarrow$ `int applyAsInt(T t)`
- **ToLongFunction<T>** $\Longrightarrow$ `long applyAsLong(T t)`
- **ToDoubleFunction<T>** $\Longrightarrow$ `double applyAsDouble(T t)`

Finally, there are versions with argument and return value of (different) primitive types (note different names of their abstract methods!)

- **IntToLongFunction** $\Longrightarrow$ `long applyAsLong(int t)`
- **IntToDoubleFunction** $\Longrightarrow$ `double applyAsDouble(int t)`
- **LongToIntFunction** $\Longrightarrow$ `int applyAsInt(long t)`
- **LongToDoubleFunction** $\Longrightarrow$ `double applyAsDouble(long t)`
- **DoubleToIntFunction** $\Longrightarrow$ `int applyAsInt(double t)`
- **DoubleToLongFunction** $\Longrightarrow$ `long applyAsLong(double t)`

The cases when both types are the same will be handled by the interface **Operator**.

Interface **BiFunction<T,U,R>** declares one abstract method taking two arguments and returning a value

```
R apply(T t, U u)
```

where **T**, **U** and **R** denote any *object* types. There are also versions for return value of primitive type: **int**, **long** or **double** (note different names of their abstract methods!)

- **ToIntBiFunction<T,U>** $\Longrightarrow$ `int applyAsInt(T t, U u)`
- **ToLongBiFunction<T,U>** $\Longrightarrow$ `long applyAsLong(T t, U u)`
- **ToDoubleBiFunction<T,U>** $\Longrightarrow$ `double applyAsDouble(T t, U u)`

10.3 Operators

Operators represent functions, for which the return type and the types of argument(s) are all the same (like for 'normal' operators: addition, multiplication, etc.). They fall into two categories: unary operators (with one argument) and binary operators (with two arguments). This interface extends **Function**, so the abstract methods have the same names as the corresponding functions.

Interface **UnaryOperator<T>** represents an operation on a single argument that produces a result of the same type as that of the argument; the abstract method is

```
T apply(T t)
```

where T is any *object* type. There are also versions for primitive types (note different names of their abstract methods!)

- IntUnaryOperator $\Longrightarrow$ `int applyAsInt(int t)`
- LongUnaryOperator $\Longrightarrow$ `long applyAsLong(long t)`
- DoubleUnaryOperator $\Longrightarrow$ `double applyAsDouble(double t)`

Interface BinaryOperator<T> represents an operation with two arguments and returning a result: all of the same type (this interface extends BiFunction). The abstract method is therefore

```
T apply(T t1, T t2)
```

where T is any *object* type. There are versions for primitive types (note different names of their abstract methods!)

- IntBinaryOperator $\Longrightarrow$ `int applyAsInt(int t1, int t2)`
- LongBinaryOperator $\Longrightarrow$ `long applyAsLong(long t1, long t2)`
- DoubleBinaryOperator $\Longrightarrow$ `double applyAsDouble(double t1, double t2)`

10.4 Predicates

Predicates are functions returning a logical value: either **true** or **false**.
Iterface Predicate<T> represents a predicate with one argument

```
boolean test(T t)
```

where T is any *object* type. There are versions for primitive types:

- IntPredicate $\Longrightarrow$ `boolean test(int t)`
- LongPredicate $\Longrightarrow$ `boolean test(long t)`
- DoublePredicate $\Longrightarrow$ `boolean test(double t)`

Interface BiPredicate<T,U> represents a predicate with two arguments and its abstract method is

```
boolean test(T t, U u)
```

10.5 Suppliers

Suppliers represent functions which do not take any argument but return a value.
Interface Supplier<T> declares an abstract method

```
T get()
```

where T is any *object* type. There are versions for primitive types (note different names of their abstract methods!)

- BooleanSupplier $\Longrightarrow$ `boolean getAsBoolean()`
- IntSupplier $\Longrightarrow$ `int getAsInt()`
- LongSupplier $\Longrightarrow$ `long getAsLong()`
- DoubleSupplier $\Longrightarrow$ `double getAsDouble()`

10.6 Example

The static function **mapFilter** in the program below expects a list, a **Predicate** and a **Function**. The predicate selects from a list elements satisfying the predicate, while the function transforms them to another type:

Listing 81 EMC-FInterfs/FInterfs.java

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.function.Function;
import java.util.function.Predicate;
public class FInterfs {
    public static void main(String[] args) {
        List<String> ls = Arrays.asList(
            "Jane", "Sue", "Alice", "Kim", "Cecilia");
        List<Character> lc = mapFilter(
            ls, s -> s.length() > 3, s -> s.charAt(0));
        System.out.println(lc);
    }
    static <T,R> List<R> mapFilter(
            List<T> list, Predicate<T> p, Function<T,R> f) {
        List<R> n = new ArrayList<>();
        for (T e : list) if(p.test(e)) n.add(f.apply(e));
        return n;
    }
}
```

In the example above the predicate selects only strings longer than three characters, while the function transforms **String** into **Character**. The program prints `[J, A, C]`.

Section 11

Introduction to multithreading

11.1 Processes and threads

Modern computers can run many applications at the same time, simultaneously. For each such application the operating system creates a separate **process** which gets its address space, standard input and output streams (and also the so called standard error stream), and other resources that the process needs. Usually, however, there are much more processes than available physical processors or cores. Therefore, the operating system has to stop (preempt) some processes, storing their current state (contents of registers, stack, etc.) and load another process in their place for some **time slice**. This operation is called **context switching** and is quite expensive. Basically, we cannot know when and how often these switchings will take place.

A **threads** are kind of a subprocesses executed "inside" a process. Each of them has its own stack, but they all share one address space, in particular the heap, and other resources allocated by the operating system to the parent process. Like processes, they can run concurrently, be preempted at unpredictable moments, etc. Each runs the same or another sequence of actions as the others.

Threads running inside the same process share data on the heap: this ensures easy communication between them. On the other hand, it can happen that two or more threads access the same piece of data and one is modifying it. In such situation it is possible that the data will be read by one thread when only 'half-modified' by another thread. Moreover, if one thread assigns a new value to a variable, this modification may be not visible by other threads, because it was only effectuated in a cache or register. Even worse: compiler can reorder instructions executed by one thread as long as this doesn't change the semantics of a sequence of instructions *from the point of view* of this thread. However, if data is shared, it may happen that the order of these instructions *does* matter from the point of view of other threads! We will therefore need certain means to handle all these situations.

Threads are represented by objects of class **Thread**. The class defines method `public void run()` — its implementation determines what the thread will do. The default implementation of **run** does nothing. Starting the thread will invoke this method, exiting it will the stop the execution of the thread — it cannot be restarted (although the object itself still exists).

Threads have a priority, which can be modified by **setPriority** method — threads with higher priority are supposed to be executed in preference to those with lower priority, but implementation of this mechanism depends on the operation system, so it cannot be relied on.

Threads cannot be "killed" — exiting the **run** method is the only way for the thread to stop execution. Each thread has a boolean flag which indicates if it is interrupted — this flag may be set by another thread, but by itself it does *not* interrupt anything: the thread may detect that its interruption flag is set and "commit suicide" by exiting **run** (or it may just ignore the interruption).

Let us now explain

- how to create and launch a new thread;
- how to ensure integrity of data shared by many threads and synchronize actions on this data.

11.2 Creating threads

There are two ways of creating the object of class **Thread** representing a thread:

- Create a class extending **Thread** and override its **run** method, so it does something useful. Then create an object of this class and invoke method **start()** on it.
- Create a class implementing the functional interface **Runnable**. This interface has one method which has to be implemented: `public void run()`. Then create an object of class **Thread** passing to its constructor an object of your class implementing **Runnable**. Call **start** on this object.

At any moment, a created thread can be in one, and only one, of six different states, represented by constants of enumeration **Thread.State**

1. NEW — a thread exists but has not yet started;
2. RUNNABLE — a thread is being executed by the JVM, although it may be waiting for some resources from the operating system (e.g., preempted thread waiting for a processor);
3. BLOCKED — a thread is blocked waiting for a monitor lock (see below) — as soon as it acquires the lock, it will be in state RUNNABLE again;
4. WAITING — a thread is waiting to be "awaken" (see below); this happens after calling, without any timeout specified, **wait** on a monitor lock or static **Thread.join**;
5. TIMED_WAITING — a thread is waiting to be "awaken", but with a specified waiting time; this happens after calling **Thread.sleep** or, with timeout specified, **wait** on a monitor lock or static **Thread.join**;
6. TERMINATED — a thread is "dead"; it has completed its execution (and cannot be restarted).

Now let us consider an example of a "data race" — several threads access the same variable at the same time:

Listing 82 QKC-BadThreads/BadThreads.java

```
public class BadThreads extends Thread {

    private long number = 0L;

    public static void main(String[] args) {
        new BadThreads().start();
    }

    public BadThreads() {
        final int MAXNUM = 40;
        for (int i = 0; i < MAXNUM; ++i)
            new Thread(new MyRunner(this),""+i).start();
        System.err.println(MAXNUM + " THREADS STARTED");
    }

    public long getNumber() {
        if (number < 1) number = number + 1;
```

```java
            number = number - 1;
            return number;
        }

        @Override
        public void run() {
            try {
                Thread.sleep(4000);
            } catch (InterruptedException ignored) { }
            System.err.println("Killing program");
            System.exit(0);
        }
}

class MyRunner implements Runnable {
    private final BadThreads bad;

    public MyRunner(BadThreads bad) {
        this.bad = bad;
    }

    @Override
    public void run() {
        String name = Thread.currentThread().getName();
        while (true) {
            long n = bad.getNumber();
            if (n != 0) {
                System.err.println(
                    "n = " + n + " in thread " + name);
                break;
            }
        }
    }
}
```

The output can be something like:

```
n = -1 in thread 1
n = 1 in thread 2
n = 2 in thread 0
n = 1 in thread 3
n = -1 in thread 5
n = -1 in thread 6
n = -1 in thread 4
n = -1 in thread 7
n = -1 in thread 8
n = -1 in thread 9
n = -1 in thread 10
n = -1 in thread 13
n = -1 in thread 11
```

```
n = -1 in thread 14
n = -1 in thread 12
n = -1 in thread 18
n = -1 in thread 15
n = -1 in thread 16
n = -1 in thread 17
n = -1 in thread 20
n = -1 in thread 19
n = -1 in thread 22
n = -1 in thread 23
n = -1 in thread 21
n = -1 in thread 26
n = -1 in thread 25
n = -1 in thread 27
n = -1 in thread 24
n = -1 in thread 28
n = -1 in thread 30
n = -1 in thread 31
n = -1 in thread 29
n = -1 in thread 33
n = -1 in thread 32
n = -1 in thread 34
n = -1 in thread 35
n = -1 in thread 38
40 THREADS STARTED
n = -1 in thread 37
n = -1 in thread 36
n = -1 in thread 39
Killing program
```

All threads stop prematurely! How to avoid simultaneous access to data by many threads?

11.3 Synchronization

All objects in Java have a hidden field (sometimes called a lock) which can be in two states: closed or open. This allows us to use any object as a monitor lock. Let `obj` be any object. Then we can synchronize a fragment of code on this object like this

```java
synchronized (obj) {
    // code
}
```

If the execution of a thread encounters a block of code synchronized on an object (`obj` in this case),

- it will be blocked, if `obj` is "locked" (closed);
- if it is open, the thread locks it and enters the block. When leaving the block, it releases the lock again.

The block of code synchronized on a lock is called **critical section**. There can be many fragments of code (critical sections), perhaps scattered in different places of the whole program, synchronized on the same object; only one of them can be executed at any instance of time — the one executed by thread which acquired the lock when it was open and closed it. All other threads which encountered a block synchronized on *exactly the same* object will have to wait until the lock is released (their state is BLOCKED). When this happens, only one of them (there is practically no way to predict which one) will acquire the lock and enter the critical section — all other will still be blocked.

Let us illustrate this:

Listing 83 QKE-BetterThreads/BetterThreads.java

```java
public class BetterThreads extends Thread {

    private long number = 0L;

    public static void main(String[] args) {
        new BetterThreads().start();
    }

    public BetterThreads() {
    final int MAXNUM = 40;
        for (int i = 0; i < MAXNUM; ++i)
            new Thread(new MyRunner(this),""+i).start();
        System.out.println(MAXNUM + " THREADS STARTED");
    }

    public long getNumber() {
        if (number < 1) number = number + 1;
        number = number - 1;
        return number;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(4000);
        } catch (InterruptedException ignored) { }
        System.out.println("Killing program");
        System.exit(0);
    }
}

class MyRunner implements Runnable {
    private final BetterThreads better;

    public MyRunner(BetterThreads better) {
        this.better = better;
    }

    @Override
```

```java
    public void run() {
        String name = Thread.currentThread().getName();
        long n;
        while (true) {
            synchronized(better) {
                n = better.getNumber();
            }
            if (n != 0) {
                System.out.println(
                    "n = " + n + " in thread " + name);
                break;
            }
        }
    }
}
```

It often happens that there are methods of a class that modify fields of an object of this class and the object is accessible by many threads. We can then synchronize whole methods of this class. This is equivalent to synchronizing the whole body of the method on **this**, i.e.,

```java
class AClass {
    synchronized void fun(/* ... */) {
        // ...
    }
}
```

is equivalent to

```java
class AClass {
    void fun(/* ... */) {
        synchronized (this) }
            // ...
        }
    }
}
```

Only one synchronized method will be executed at any given instance of time, provided they are all called on *exactly the same object*.
Example:

Listing 84 QKA-SimpleThreads/FibThreads.java

```java
public class FibThreads {

    private int counter;

    public static void main(String[] args) {
        new FibThreads();
    }

```

```
    FibThreads() {
        Runnable[] runs =
            {
                new Fibo(46L,this), new Fibo(44L,this),
                new Fibo(46L,this), new Fibo(45L,this),
                new Fibo(45L,this), new Fibo(45L,this),
            };
        counter = runs.length;

        for (Runnable r : runs)
            new Thread(r).start();

        System.out.println("Exiting from \"main\"");
    }

    synchronized void finished(long arg, long res) {
        counter = counter - 1;
        System.out.println("Fib(" + arg + ") = " + res +
            ". Still running: " + counter);
    }
}

class Fibo implements Runnable {

    private final long  arg;
    private final FibThreads parent;

    static long fibon(long n) {
        return (n < 2) ? n : fibon(n-2) + fibon(n-1);
    }

    Fibo(long n, FibThreads w) {
        arg    = n;
        parent = w;
    }

    @Override
    public void run() {
        System.out.println("Fibo(" + arg + ") starting");
        long res = fibon(arg);
        parent.finished(arg,res);
    }
}
```

This is also possible to synchronize static methods: this is equivalent to synchronizing the whole body of the function on the object of class **Class** representing the class — there is only one such object and it can be referenced to as AClass.class, where **AClass** is the name of a class (or by calling **getClass** on any object of this class):

```
class AClass {
```

```
    synchronized static void fun(/* ... */) {
        // ...
    }
}
```

is equivalent to

```
class AClass {
    static void fun(/* ... */) {
        synchronized (AClass.class) }
            // ...
        }
    }
}
```

It is crucial to always remember which object plays the rôle of a lock in a given context. For example in the code below

```
class AClass {
    static double d;
    synchronized static void set(double x) { d = x; }
    synchronized double get() { return d; }
}
```

the (non-static) method **get** is synchronized on **this**, while **set** on AClass.class, because it is static. These two functions, both having access to the static field `d`, *can* be executed simultaneously, probably contrary to our intentions.

One may encounter a similar situation when dealing with outer and inner classes:

```
class Outer {
    double n;
    synchronized set(int nn) { n = nn; }
    // ...
    class Inner {
        synchronized int get() { return n; }
        // ...
     }
}
```

Here **set** in synchronized on **this** object of the outer class, while **get** on **this** which points to an object of the inner class. To avoid this inconsistency, we could have synchronized **get** on the object pointed to by **this** from the outer class:

```
class Outer {
    double n;
    synchronized set(int nn) { n = nn; }
    // ...
    class Inner {
        int get() {
            synchronized (Outer.this) {
                return n;
            }
        }
        // ...
     }
}
```

11.4 Inter-thread coordination

A thread which encountered a critical section with a closed lock is in the BLOCKED state. It is ready to continue as soon as the lock becomes open. However, sometimes we want a thread to wait for another thread to finish, before it can continue (e.g., because the other thread prepares some data which is needed by the current thread to be able to proceed). This may be achieved by calling

```
otherThread.join();
```

where otherThread is the reference to a thread; the current thread (executing this statement) will wait until otherThread has finished.

In another scenario, a thread has to wait for some event to take place before continuing, and this event is somehow controlled by another thread. We then have to change its state to WAITING. There are *two* queues associated with an object playing the rôle of a lock:

- those which are BLOCKED on it, i.e., they are ready to continue as soon as the lock is released, and
- those which are WAITING on this lock, i.e., they do nothing until they are 'woken up' by another thread — at this moment they are transferred to the BLOCKED queue and will be able to continue as soon as the lock has been open.

This scenario can be realized by using methods (from class **Object**)

- **wait**;
- **notify**;
- **notifyAll**.

All these methods must be invoked

- on an object which plays the rôle of the lock of some critical sections;
- inside a critical section guarded by this object.

The sequence of actions is as follows (by lock we mean the object on which critical sections involved are synchronized):

- A thread calls **wait** on lock. After that the thread is in state WAITING, the lock is *released* (open) and the thread becomes idle (doesn't do anything). It is crucial that the lock is released, because another thread will have to "wake up" this thread also being in a critical section guarded by the same lock — this other thread would never be able to enter this critical section to do it, if the lock is closed! The waiting thread will not proceed until it is awoken by another thread.
- Another thread can now enter a critical section guarded by lock, do something (like modifying the value of a field of an object) and then notify ("wake up") the waiting thread by calling **notify** on lock. At this moment, the waiting thread is moved from the queue of waiting threads to the queue of blocked threads — it cannot resume its execution immediately because the other thread, the one which called **notify**, was in a critical section, so lock is at this moment closed.

There is also a version of **wait** which takes an additional argument — time to wait. After this time has elapsed, the waiting thread will be woken up even without notification.

The method **notify** wakes up only one thread waiting on lock (if there are many, it is not known which one). To notify *all* threads waiting on a lock, call **notifyAll** (which should be avoided if not necessary, because it is rather expensive).

11.5 Terminating threads

A thread cannot be "killed" by another thread. The only way for any thread to terminate is to exit from the **run** function. How can we notify a running thread that we would like it to terminate?

In order to do it, one can set its flag interrupted by calling **interrupt** on an object representing the thread to be stopped. But the thread that receives such signal must detect it and somehow react to this event; as a matter of fact it may completely ignore it and happily proceed!

However, remember that if a thread for which this flag has been set

- is waiting (has called **wait**) or calls **wait**;
- is sleeping (executed **Thead.sleep**) or calls **Thread.sleep**
- awaits for another thread to terminate (executed **anotherThread.join** or calls **anotherTread.join**

then the **InterruptedException** *checked* exception will be thrown (all aforementioned methods must be called in a **try-catch** clause).

> When **InterruptedException** is thrown, the interrupted status is cleared, so inside the **catch** clause the flag is already *not* set!

Therefore, to stop the thread, we can just put **return** into this clause. Threads may also check their interrupted status by calling **isInterrupted** on the thread executing a given piece of code. For example, one can get the reference to the thread executing the current code and check if it has been interrupted by invoking

`Thread.currentThread().isInterrupted()`

which returns **true** or **false**. Similar method, **interrupted**, also checks the interrupted status, but also clears it if it was set.

11.6 Examples

Let us consider a couple of examples.

The first will illustrate the way to stop a thread by modifying a boolean variable canRun. Modifying this variable doesn't need to be synchronized, because operations on four-byte variables of primitive type are atomic (but not on **double**s or **long**s). However, as it is accessed by two different threads, it should be declared as **volatile**. This means that it should always be read directly from memory and stored in memory: compiler is not allowed to cache it anywhere (in registers or cache memory). Otherwise, modifications made by one thread wouldn't be necessarily seen by other threads!

Listing 85 QKF-StopThread/StopThread.java

```java
public class StopThread {
      // booleans are written/read atomically,
      // 'volatile' here to avoid caching value
    static volatile boolean canRun = true;

    public static void main (String[] args) {

```

```java
        Thread runner = new Thread(() -> {
            while(canRun) {
                System.out.println("still running...");

                try {Thread.sleep(750);}
                catch(InterruptedException ignored) { }
            }
            System.out.println("INTERRUPTED!");
        });
        runner.start();

        try {Thread.sleep(5000);}
        catch(InterruptedException ignored) { }

        canRun = false;
    }
}
```

The next example will illustrate interrupting threads by setting their interrupted flag. Note that **interrupt** does *not* interrupt anything; the 'interrupted' thread must detect the interruption itself and decide what to do — here it just returns from **run** and hence becomes TERMINATED.

Listing 86 QJS-RunThreads/RunThreads.java

```java
public class RunThreads {
    public static void main (String[] args) {
          // ShowTime extends Thread
        Thread tTime = new ShowTime();

          // ShowLett implements Runnable
        Thread tLett = new Thread(new ShowLett());

          // object of anonymous class extending Thread
        Thread tNumb = new Thread() {
            @Override
            public void run() {
                int num = 0;
                while (true) {
                    try {
                        Thread.sleep(2000);
                    } catch(InterruptedException exc) {
                        System.out.println(
                                " |\nNumb interrupted.");
                        return;
                    }
                    System.out.printf(" | N %d", ++num);
                }
            }
```

```
        };

          // object of anonymous class extending Thread;
          // using a lambda (as Runnable is functional)
        Thread tHebr = new Thread( () -> {
            int lett = 0x5D0-1;
            while (true) {
                try {
                    Thread.sleep(1750);
                } catch(InterruptedException exc) {
                    System.out.println(
                            " |\nHebr interrupted.");
                    return;
                }
                int c =  0x5D0 + (++lett-0x5D0)%27;
                System.out.printf(" | H %c", (char)c);
            }
        });

        tTime.start();
        tLett.start();
        tNumb.start();
        tHebr.start();
        try {
            Thread.sleep(10*1000);

            tTime.interrupt(); Thread.sleep(3*1000);
            tLett.interrupt(); Thread.sleep(4*1000);
            tNumb.interrupt(); Thread.sleep(3* 200);
            tHebr.interrupt(); Thread.sleep(   200);
        } catch (InterruptedException e) {
            System.out.println("Should never happen!!!");
            System.exit(1);
        }
        System.out.println("ALL DONE");
    }
}

class ShowTime extends Thread {
    @Override
    public void run() {
        int time = 0;
        while (true) {
            try {
                Thread.sleep(1500);
            } catch(InterruptedException exc) {
                System.out.println(" |\nTime interrupted.");
                return;
            }
            int min = ++time/60;
```

```java
            int sec = time%60;
            System.out.printf(" | T %02d:%02d",min,sec);
        }
    }
}

class ShowLett implements Runnable {
    @Override
    public void run() {
        int lett = 'A'-1;
        while (true) {
            try {
                Thread.sleep(1250);
            } catch(InterruptedException exc) {
                System.out.println(" |\nLett interrupted.");
                return;
            }
            int c =  'A' + (++lett-'A')%26;
            System.out.printf(" | L %c", (char)c);
        }
    }
}
```

Let us now consider another example illustrating inter-thread coordination. In class **Texts** there is room for only one text (variable txt); there is also a boolean value newTxT which is by assumption **true** only if a text has been set by author (class **Author**) but not yet read (taken) by the publisher (class **Publisher**). Therefore, if newTxT is **true**, the author cannot set a new value of txt — he waits until the publisher takes the text and notifies him about this. On the other hand, the publisher cannot proceed if newTxt is **false** — then *he* waits until the author sets a new text and wakes him up (by means of **notify**). Note that the condition newTxT is checked in **while** loop, and *not* by an **if**. In this simple program **if** would be sufficient. However, when more threads are active, **while** has to be used. Imagine that a thread has already executed the **if** statement went on hold. Then another thread modifies the condition variable (here, newTxT) and notifies this thread, which is then transferred to the blocked queue. After the lock has been open, the thread resumes execution, but by this time yet another thread could have changed the variable again — this will not be detected, as the **if** statement has already been executed!

Note also that here the variable newTxT doesn't need to be **volatile**: modifications of variables made inside or before entering a critical section *are* visible by the code synchronized on the same lock which enters its critical section later.

Listing 87 QKG-Coord/Coord.java

```java
class Texts {
    private String txt = null;
    private boolean newTxt = false;

      // invoked by Author to set a new text
```

```java
    synchronized public void setText(String s) {
        while (newTxt) { // not if!!!
            try {
                wait();
            } catch(InterruptedException exc) {}
        }
        txt = s;
        newTxt = true;
        notify(); // invoked on 'this'
    }

      // invoked by Publisher to get a text
    synchronized public String getText() {
        while (!newTxt) { // not if!!!
            try {
                wait(); // invoked on 'this'
            } catch(InterruptedException exc) {}
        }
        newTxt = false;
        notify(); // invoked on 'this'
        return txt;
    }
}

class Publisher extends Thread {
    private Texts txtArea;
    public Publisher(Texts t) {
        txtArea=t;
    }

    public void run() {
        String txt = null;
        while ((txt = txtArea.getText()) != null) {
            System.out.println("-> " + txt);
        }
    }
}

class Author extends Thread {
    private Texts txtArea;
    public Author(Texts t)  {
        txtArea=t;
    }

    public void run() {
        String[] texts = {"Hamlet", "War and Peace",
            "Macbeth", "The Trial", "Crime and Punishment",
            "Madame Bovary", null };
        for (int i=0; i<texts.length; i++) {
            try {
```

```
                    // writing a book takes some time...
                    sleep((int)(1500 + Math.random()*300));
                } catch(InterruptedException ignored) { }

                txtArea.setText(texts[i]);
            }
        }
}

public class Coord {
    public static void main(String[] args) {
        Texts t    = new Texts();
        Thread t1 = new Author(t);
        Thread t2 = new Publisher(t);
        t1.start();
        t2.start();
    }
}
```

The next example illustrates stopping and resuming threads. Variables stopped and suspended should be **volatile**, because their values are modified when executing a code which is *not* in a critical section guarded by the lock, so without **volatile** there would be no guarantee that inside a critical section a new, modified value, is visible. Again, it is very important to check the values of these 'condition' variables in a **while** loop, and *not* just by an **if**. Suppose a thread is waiting on suspended (suspended is **true**)

while (suspended) wait();

and another thread sets suspended to **false** and notifies this thread. This thread cannot resume execution immediately — it is now blocked on the lock. When the lock is released, in unforeseeable future, and the thread can finally resume its execution, it may happen that suspended is again **true**, because it has been modified again by another thread! If we check the condition in a loop, it *will* be checked again, while with **if** there would no additional checking and the thread would proceed, although suspended would now be **true**!

Listing 88 QKH-SuspResum/SuspResum.java

```
import javax.swing.JOptionPane;

class MyThread extends Thread {
    volatile boolean stopped   = false;
    volatile boolean suspended = false;

    public void run() {
        int num = 0;
        while(!stopped) {
            try {
                synchronized(this) {
                    while (suspended) wait();
                }
```

```java
            } catch (InterruptedException exc) {
                System.out.println(
                        "Interrupted on wait");
            }
            if (suspended) System.out.println(
                              "Still suspended");
            else           System.out.println(++num);
        }
    }

    public void stopThread()    { stopped = true;   }
    public void suspendThread() { suspended = true; }
    public boolean isSusp()     { return suspended; }
    public boolean isStop()     { return stopped;   }

    public void resumeThread() {
        suspended = false;
        synchronized(this) {
            notify();
        }
    }
}

public class SuspResum {
    public static void main(String args[]) {
        String msg = "I = interrupt\n" +
                     "E = end\n" +
                     "S = suspend\n" +
                     "R = resume\n" +
                     "N = new start";
        MyThread t = new MyThread();
        t.start();
        while (true) {
            String cmd = JOptionPane.showInputDialog(msg);
            if (cmd == null) break;
            if (cmd.trim().length() == 0) continue;
            char c = Character.toUpperCase(cmd.charAt(0));
            switch (c) {
                case 'I' : t.interrupt();     break;
                case 'E' : t.stopThread();    break;
                case 'S' : t.suspendThread(); break;
                case 'R' : t.resumeThread();  break;
                case 'N' :
                    if (t.isAlive())
                        JOptionPane.showMessageDialog(
                                null, "Thread alive!!!");
                    else {
                        t = new MyThread();
                        t.start();
                    }
```

```java
                    break;
                default   : break;
            }
            JOptionPane.showMessageDialog(null,
                    "Command " + cmd + " executed.\n" +
                    "Thread alive? " +
                    (t.isAlive() ? "Y\n" : "N\n") +
                    "Thread interrupted? " +
                    (t.isInterrupted() ? "Y\n" : "N\n") +
                    "Thread suspended? " +
                    (t.isSusp() ? "Y\n" : "N\n") +
                    "Thread stopped? " +
                    (t.isStop() ? "Y\n" : "N")
            );
        }
        System.exit(0);
    }
}
```

The Java standard library provides a multitude of classes which make multi-threaded programming much easier and less error prone. As an example, in the program below, we use a **BlockingQueue** which represents a queue automatically guarded against simultaneous accesses. To its constructor, we pass the required capacity of the queue. The mechanism is a possible solution of the classic *producer-consumer* problem. Threads representing producers add new elements by invoking **put**: if there is no room for a new element because the queue is full (number of its elements reached the capacity), they will be blocked and will have to wait until a consumer has popped at least one element. On the other hand, a consumer thread invoking **take** is blocked when there are no elements in the queue available, and will wait until a producer has supplied a new element. What is important is the fact that synchronization of **put** and **take** operations will be taken care of by the library — we don't have to worry about it.

Listing 89 QKI-BlockQ/BlockQ.java

```java
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class BlockQ {
    public static void main(String[] args) {
        BlockingQueue<Integer> queue =
                new ArrayBlockingQueue<>(10);
        Cons c = new Cons(queue,0);
        c.start();
        Prod p1= new Prod(queue,10);
        Prod p2= new Prod(queue,20);
        p1.start();
        p2.start();
    }
```

```

    static class Prod extends Thread {
        private final BlockingQueue<Integer> queue;
        private final int low;
        Prod(BlockingQueue<Integer> queue, int low) {
            this.queue = queue;
            this.low   = low;
        }

        @Override
        public void run() {
            for (int i = 0; i < 10; ++i) {
                try {
                    int d = low + (int)(10*Math.random());
                    System.err.println("Try to put " + d);
                    queue.put(d);
                    System.err.println("       put " + d);
                    sleep(500 + (int)(300*Math.random()));
                } catch(InterruptedException ignore) { }
            }
            try {
                  // poison pill
                queue.put(-1);
            } catch(InterruptedException ignore) { }
        }
    }

    static class Cons extends Thread {
        private final BlockingQueue<Integer> queue;
        private final int poison;
        Cons(BlockingQueue<Integer> queue, int poison) {
            this.queue  = queue;
            this.poison = poison;
        }

        @Override
        public void run() {
            int pills = 0;
            while (pills < 2) { // as there are 2 producers
                try {
                    System.err.println("taking");
                    int d = queue.take();
                    System.err.println(" taken " + d);
                    if (d < poison) {
                        System.out.println(
                                    "Poison pill received");
                        ++pills;
                    }
                    sleep(700 + (int)(300*Math.random()));
                } catch(InterruptedException ignore) { }
```

```
            }
        }
    }
}
```

Another useful tool provided by the library is the **Timer** class. It allows to run a task (represented by an object of a class extending **TimerTask** with its method **run** overridden) repeatedly: we can set a frequency of running the task (or rather a period) and a delay before running it for the first time. This is illustrated in the program below. First, we create a timer which after 20 seconds will run once the **run** method on an object of an anonymous class extending **TimerTask** (what will stop the program). In a **JOptionPane** window, we display a mathematical puzzle and, using a **Timer**, run repeatedly a task displaying a message prompting the user for an answer and printing the time he/she has already spent on solving this puzzle.

Listing 90 QKN-TimerExample/TimerExample.java

```java
import javax.swing.JOptionPane;
import java.util.Random;
import java.util.Timer;
import java.util.TimerTask;

public class TimerExample {
    static int total   = 0,
               correct = 0;

    public static void main(String[] args) {
          // will be run once only, 20 seconds from now
        new Timer().schedule(new TimerTask() {
                public void run() {
                    System.out.println(correct + "/" +
                            total + " correct answers.");
                    System.exit(0);
                }
            }, 20*1000);
        Random rand = new Random();
        while (true) {
            int a = rand.nextInt(10) + 1;
            int b = rand.nextInt(10) + 1;
            String oper = a + " x " + b + " is?";
            int expected = a * b;
            Timer timer = new Timer();
              // 1 second delay and then every 2 seconds
            timer.schedule(new Prompt(a,b),1000,2000);
            String s = JOptionPane.showInputDialog(
                    null,oper,"Higher math drill",
                    JOptionPane.QUESTION_MESSAGE);
            if (s == null) System.exit(1);
```

```
            int ans = 0;
            try {
                ans = Integer.parseInt(s);
            } catch(NumberFormatException e) {
                timer.cancel();
                continue;
            }
            ++total;
            if (ans == expected) {
                ++correct;
                System.out.println("OK");
            }
            else
                System.out.println("Wrong!!!");
            timer.cancel();
        }
    }
}

class Prompt extends TimerTask {
    private String oper;
    long start = System.currentTimeMillis();

    public Prompt(int a, int b) {
        oper = a + " x " + b + " is so easy... ";
    }

    @Override
    public void run() {
        long time = System.currentTimeMillis() - start;
        System.out.println(oper +
            "you've been thinking for " +
            (System.currentTimeMillis()-start) + " ms");
    }
}
```

The last examples illustrate **Executors**: these are objects that accept tasks (in the form of object implementing **Runnable** or **Callable**) and then create threads and start them according to a specified policy. In the example below, the executor can accept any number of tasks, but will never execute more than two of them simultaneously:

Listing 91 QKJ-ThreadPool/ThreadPool.java

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class ThreadPool extends Thread {

    ExecutorService pool = null;
```

```java

    public static void main(String[] args) {
        new ThreadPool().start();
    }

    ThreadPool() {
        Runnable[] runs =
            {
                new Fibo(40L), new Fibo(46L),
                new Fibo(41L), new Fibo(45L),
                new Fibo(42L), new Fibo(44L),
                new Fibo(43L), new Fibo(43L),
                new Fibo(47L), new Fibo(48L),
                new Fibo(44L), new Fibo(42L),
                new Fibo(45L), new Fibo(41L),
                new Fibo(36L), new Fibo(40L),
            };
          // pool for 2 concurrent threads
        pool = Executors.newFixedThreadPool(2);

          // submitting 16 threads...
        for (Runnable r : runs)
            pool.execute(r);

          // no other threads will be added
        pool.shutdown();
        System.err.println("Shutdown executed");
    }

    public void run() {
        while (!pool.isTerminated()) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ignored) { }
            System.err.println("Still running...");
        }
        System.err.println("All done");
    }
}

class Fibo implements Runnable {

    private final long arg;

    Fibo(long n) {
        arg = n;
    }

    static long fibon(long n) {
        return (n < 2) ? n : fibon(n-2) + fibon(n-1);
```

```java
    }

    @Override
    public void run() {
        System.err.println("Fibo(" + arg + ") starts");
        long res = fibon(arg);
        System.err.println("Fibo(" + arg + ") completed " +
                                "with res = " + res);
    }
}
```

Executors accept also tasks defined by object of classes implementing the functional **Callable** interface

```java
interface Callable<V> {
    V call() throws Exception;
}
```

Unlike the **run** in a **Runnable**, the **call** method in **Callable** is allowed to throw an exception and, if there was no exception, returns a value. We can pass a callable to an executor by calling **submit** method which returns an object of type **Future<V>**. It represents the future result of the task, which we can get by calling the (blocking) method **get**. If the task has completed successfully, we will get the result. If an exception was thrown inside the **call** method, it will be caught, stored in the future object and rethrown when we call **get** in the form of an **EcecutionException** object (from which we can extract information about the original exception). There are also other useful methods of **Future** class, which allow to check without blocking if the task has been completed or to make an attempt to cancel the task.

In the example below, we use the **invokeAll** method of executors. It takes a collection of tasks, returns a list of **Future**s and blocks until all tasks are completed. We can then get the results from these **Future**s:

Listing 92 QLG-Futures/FuturesExample.java

```java
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Future;

class SingleTask implements Callable<Integer> {
    Integer num;
    public SingleTask(int n) {
        num = n;
    }
      // do NOT handle exceptions here!
    public Integer call() throws Exception {
        Thread.sleep(3000);
```

```java
            if (num%3 == 0) throw new NumberFormatException();
            return num;
        }
    }

    public class FuturesExample {

        public static int sum(ExecutorService exec,
                              List<Callable<Integer>> tasks) {
            List<Future<Integer>> results = null;
            try {
                System.out.println("...invoking all");
                results = exec.invokeAll(tasks);
            } catch(InterruptedException e) {
                System.out.println("invokeAll failed");
                System.exit(1);
            }
            exec.shutdown(); // will not wait for other tasks
            System.out.println("Seem like done");

            int sum = 0;
            for (Future<Integer> r : results) {
                try {
                    // get gets the result or rethrows
                    // exception thrown in the call
                    int nextint = r.get();
                    System.out.println(nextint + " added");
                    sum += nextint;
                }
                catch(InterruptedException e) {
                    System.err.println("Should not happen");
                    System.exit(1);
                }
                catch(ExecutionException e) {
                    System.err.println("** ExecutionException " +
                            "caused by " + e.getCause());
                    System.err.println("** No value to add, " +
                            "but continuing... ");
                }
            }
            return sum;
        }

        public static void main(String[] args) {
            List<Callable<Integer>> taskList =
                new ArrayList<Callable<Integer>>();
            ExecutorService exec =
                            Executors.newFixedThreadPool(10);
            for (int i=1; i <=5; i++) {
                Callable<Integer> task = new SingleTask(i);
```

```java
            taskList.add(task);
        }
        int result = sum(exec, taskList);
        System.out.println("Result: " + result);
    }
}
```

Finally, there is the **FutureTask** class, also representing a task — we pass a task to the constructor in the form of a **Callable** or **Runnable** object. The class has a useful protected method **done** which we can override and which will be called automatically when the task completes — either normally or by throwing an exception. Let us see an example:

Listing 93 QLF-Tasks/Tasks.java

```java
import java.util.concurrent.Callable;
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.FutureTask;
import java.util.concurrent.TimeUnit;

public class Tasks {
    public static void main(String[] args) {
        FutureTask<Long>[] fs = new MyFutureTask[]{
                new MyFutureTask(new CallableFibo(43)),
                new MyFutureTask(new CallableFibo(45)),
                new MyFutureTask(new CallableFibo(-2)),
                new MyFutureTask(new CallableFibo(47))
            };
        ExecutorService ex =
                Executors.newFixedThreadPool(2);
        for (FutureTask<Long> t : fs) ex.submit(t);
        ex.shutdown();
        try {
            boolean term =
                ex.awaitTermination(10,TimeUnit.SECONDS);
            if (term)
                System.err.println( "All tasks completed");
            else
                System.err.println(
                    "Timeout: some tasks still running");
        } catch(InterruptedException e) {
            System.err.println("Main thread interrupted");
        }
    }
}

class MyFutureTask extends FutureTask<Long> {
    public MyFutureTask(Callable<Long> c) {
```

```
            super(c);
        }
        public void done() {
            String mes = "DONE. ";
            if (isCancelled()) mes += "Cancelled.";
            else
                try {
                    mes += ("Result OK: " + get());
                } catch(Exception e) {
                    mes += ("Exception: " + e.getCause().toString());
                }
            System.err.println(mes);
        }
};

class CallableFibo implements Callable<Long> {
    long arg;
    public CallableFibo(long arg) {
        this.arg = arg;
    }
    public Long call() throws Exception {
        return fibo(arg);
    }
    private long fibo(long n) {
        if (n < 0)
            throw new IllegalArgumentException("From fibo");
        if (n <= 1) return n;
        return fibo(n-1)+fibo(n-2);
    }
}
```

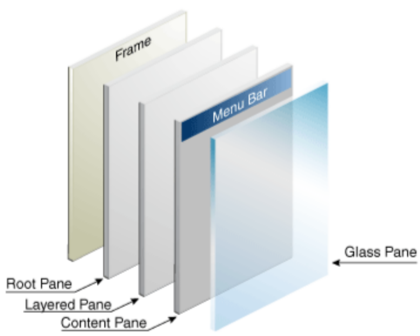
Section 12

GUI - introduction

12.1 Components and containers

Java provides relatively simple tools that can be used to build graphical user interfaces in a way independent of the user's platform. They are collected in two main packages, ***javax.swing*** and ***java.awt*** and their subpackages. Classes defined in these packages describe graphical components (the so called *widgets*, as, for example, windows, buttons, lists, menus, tables) and ways of interactions between the GUI and the user, e.g., by means of reacting to mouse movements and clicks or pressing keys on the keyboard. Generally, we work with GUI components according to the following rules:

- Graphical components are created, like any other Java objects, by using **new** and passing some information to constructors — this information determines properties of the components.
- Components have properties (texts appearing on them, fonts, colors, etc.) — they can be set in a constructor but usually can also be modified and examined dynamically at run time by *setters* (**setXXX**) and *getters* (**getXXX**, **isXXX**), where **XXX** is the name of a property (as color, width, etc.).
- Properties are described by values of primitive types (**int**, **double**) or by objects of classes (as **Font**, **Color**).
- Many components are also **containers**, i.e., we can add to them other components (also other containers).
- Swing windows contain the so called contentPane which is the default container (layer) to which other components may be added. As a matter of fact, it contains many more, although used much less frequently, layers that also can contain components; on top of all layers there is a special layer — 'glass pane' (the picture from Oracle documentation)



- Graphical appearance of the components and their behavior are determined by *layout managers* associated with these components.
- Layout managers are object of special classes and they belong to the properties of components — they may be set for each component separately.
- Applications create one or more windows containing various visual components allowing the user to communicate with the running program.

- Hierarchy of components has as its root a window of the highest level; it is an object of class **JFrame**, but can be also a **JWindow**, **JApplet** or **JDialog** — these are the so called **heavy-weight components**.

- Communication between the user and the GUI is based on handling **events** (mouse clicks, pressing keys, etc.).

- Events are fetched from the operating system and organized in a queue of events managed by the special thread — the so called **event dispatch thread**).

Historically, the first graphical library in Java was AWT (*Abstract Windowing Toolkit*). Its capabilities were rather limited, many useful graphical components (for example, tables) were missing. Components were 'heavyweight' — implemented in terms of native components provided by a given platform (and, consequently, had different 'look and feel' on various platforms).

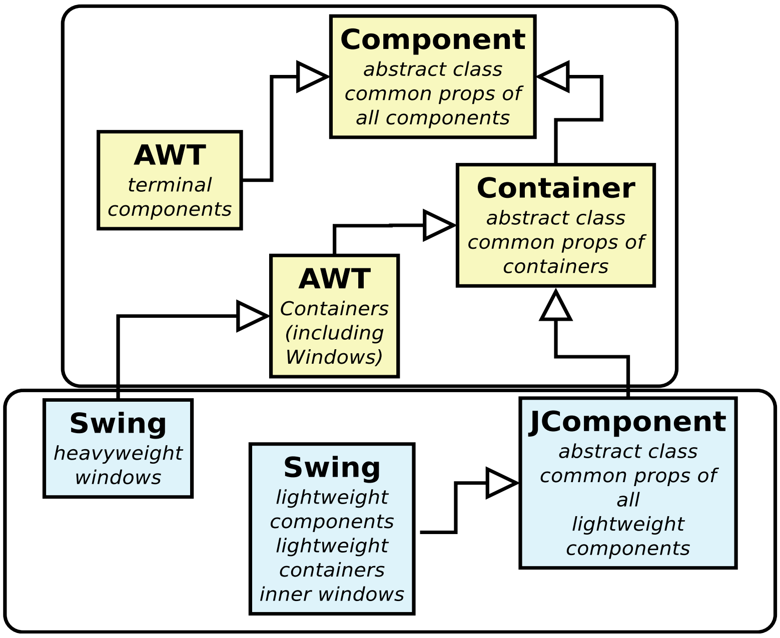
Later, on top of the AWT, a new library was created — the so called **Swing**; it is located in the package ***javax.swing*** and its subpackages. Swing is much richer than AWT, defines much more components with many useful features. Most of them are **lightweight**, i.e., they are implemented in pure Java without referring to native components of the operating system what implies that they *are* platform independent. There are only a few **heavyweight** components describing main windows of applications, inside which all other components are located. These heavyweight components *are* linked with the native graphical system of the operating system, what is understandable, since ultimately this is the window manager of the platform which is responsible for dealing with windows of all applications running at a given moment. The heavyweight components are **JFrame**, **JApplet**, **JDialog** and **JWindow**; we will mainly use **JFrame**.

12.2 Swing components

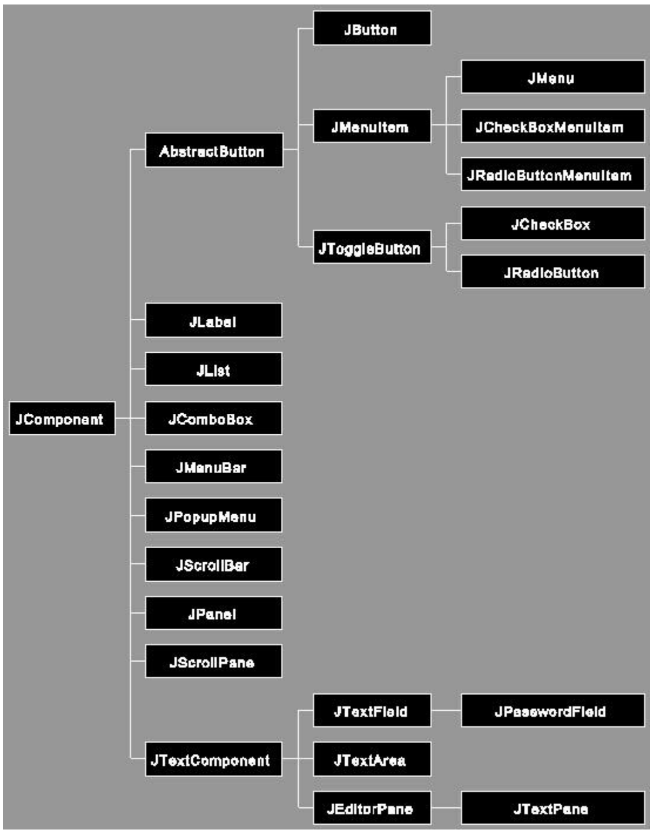
As we said, swing is built on top of the AWT library.

- All components and containers (of both AWT and swing) implement **Component** from ***java.awt***; this interface declares many useful methods for setting and getting properties of all components and containers.
- All containers (both AWT and swing) implement **Container** from ***java.awt***.
- **JComponent** determines common properties of all lightweight swing components.
- Specific properties and functionality of components are defined in classes of these components.
- Heavyweight swing containers are *not* **JComponent**s — they inherit directly from AWT **Container**.

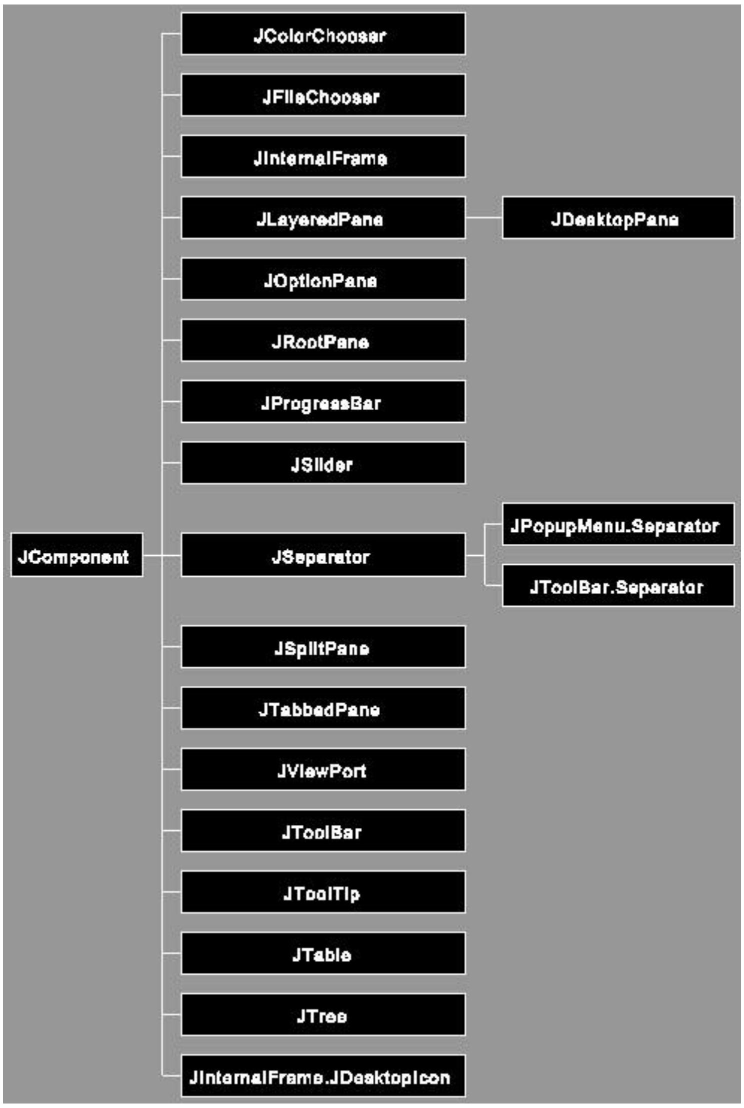
Therefore, the hierarchy looks like this:



The following picture presents **JComponent**s from Swing that extend the corresponding widgets from AWT (from *Magellan Institute Swing Short Course*):



Swing provides much more components; many of them do not have their counterparts in AWT. They are presented in the following figure, also taken from *Magellan Institute Swing Short Course*). Some of them are very elaborated (and often not so easy to use), like **JTable**, **JTree** and some others, but generally it is not difficult to use them, at least on a basic level.



Let us briefly describe swing components and their functionality:

- Buttons: JButton, JToggleButton, JCheckBox, JRadioButton — may have a text and/or an icon with arbitrary positioning, with different text or icon for different states (enabled, disabled), mouse-over effects, may have borders, mnemonics and tips attached, may react to clicks with a mouse or programmatically and so on.
- Labels: JLabel — may have a text and/or an icon with arbitrary positioning, mnemonics, can be linked with another component so clicking its alt-mnemonic transfers the focus to the other component, etc.
- Menus: JMenu, JMenuItem, JCheckBoxMenuItem, JRadioMenuItem — have all properties of buttons. Additionally there are context menus: JPopupMenu.
- Sliders: JSlider — configurable range, description, icons etc.
- Color and file choosers: JColorChooser, JFileChooser — configurable widgets allowing the user to select colors or files (directories); may be embedded in other components.
- One-line editing widgets: JTextField, JPasswordField, JFormattedTextField — entering texts, possible verification; special widget for entering sensitive data (without creating String objects).
- Multi-line text widget: JTextArea, JEditorPane, JTextPane — editing multi-line texts with formatting, embedded graphics etc.

- Lists: **JList** — widget displaying a list of object; fully configurable and dynamic (reflecting modifications of the list at run time).
- Combo boxes: **JComboBox** — similar to **JList** but space-saving.
- Tables: **JTable** — extremely configurable representation of tables; columns of different types, custom rendering of cells and columns, sorting rows etc.
- Trees: **JTree** — configurable representation of data stored in the tree-like form.
- 'Helper' containers: **JPanel**, **JSplitPane**, **JTabbedPane**, **JScrollPane**, — allows the user to group and organize components in various configurable ways.
- Tool bars: **JToolBar** — configurable tool bars for easy launching various actions

Some of them are used in the example below (taken from the Oracle's Swing Tutorial)

Listing 94 MBD-Demo/BasicDnD.java

```java
/*
 * Taken from:
 * https://docs.oracle.com/javase/tutorial/uiswing/examples/
 *          dnd/BasicDnDProject/src/dnd/BasicDnD.java
 * Slightly modified to avoid some warnings
 */
import java.awt.*;
import java.awt.event.*;
import java.awt.datatransfer.*;
import java.text.*;
import java.util.*;
import javax.swing.*;
import javax.swing.table.*;
import javax.swing.text.*;
import javax.swing.tree.*;

public class BasicDnD extends JPanel
        implements ActionListener {
    private static JFrame frame;
    private JTextArea textArea;
    private JTextField textField;
    private JList<String> list;
    private JTable table;
    private JTree tree;
    private JColorChooser colorChooser;
    private JCheckBox toggleDnD;

    public BasicDnD() {
        super(new BorderLayout());
        JPanel leftPanel = createVerticalBoxPanel();
        JPanel rightPanel = createVerticalBoxPanel();

        //Create a table model.
        DefaultTableModel tm = new DefaultTableModel();
        tm.addColumn("Column 0");
        tm.addColumn("Column 1");
```

```java
        tm.addColumn("Column 2");
        tm.addColumn("Column 3");
        tm.addRow(new String[]{"Table 00", "Table 01",
                               "Table 02", "Table 03"});
        tm.addRow(new String[]{"Table 10", "Table 11",
                               "Table 12", "Table 13"});
        tm.addRow(new String[]{"Table 20", "Table 21",
                               "Table 22", "Table 23"});
        tm.addRow(new String[]{"Table 30", "Table 31",
                               "Table 32", "Table 33"});

        //LEFT COLUMN
        //Use the table model to create a table.
        table = new JTable(tm);
        leftPanel.add(
                createPanelForComponent(table, "JTable"));

        //Create a color chooser.
        colorChooser = new JColorChooser();
        leftPanel.add(createPanelForComponent(
                    colorChooser, "JColorChooser"));

        //RIGHT COLUMN
        //Create a textfield.
        textField = new JTextField(30);
        textField.setText("Favorite foods:" +
                          "\nPizza, Moussaka, Pot roast");
        rightPanel.add(createPanelForComponent(
                          textField, "JTextField"));

        //Create a scrolled text area.
        textArea = new JTextArea(5, 30);
        textArea.setText("Favorite shows:" +
                         "\nBuffy, Alias, Angel");
        JScrollPane scrollPane = new JScrollPane(textArea);
        rightPanel.add(createPanelForComponent(
                          scrollPane, "JTextArea"));

        //Create a list model and a list.
        DefaultListModel<String> listModel =
                                new DefaultListModel<>();
        listModel.addElement("Martha Washington");
        listModel.addElement("Abigail Adams");
        listModel.addElement("Martha Randolph");
        listModel.addElement("Dolley Madison");
        listModel.addElement("Elizabeth Monroe");
        listModel.addElement("Louisa Adams");
        listModel.addElement("Emily Donelson");
        list = new JList<>(listModel);
        list.setVisibleRowCount(-1);
```

```
        list.getSelectionModel().setSelectionMode(
                ListSelectionModel.
                MULTIPLE_INTERVAL_SELECTION);

        list.setTransferHandler(new TransferHandler() {
            public boolean canImport(
                TransferHandler.TransferSupport info) {
                // we only import Strings
                if (!info.isDataFlavorSupported(
                        DataFlavor.stringFlavor)) {
                    return false;
                }

                JList.DropLocation dl = (JList.DropLocation)
                                      info.getDropLocation();
                if (dl.getIndex() == -1) {
                    return false;
                }
                return true;
            }

            public boolean importData(
                    TransferHandler.TransferSupport info) {
                if (!info.isDrop()) {
                    return false;
                }

                // Check for String flavor
                if (!info.isDataFlavorSupported(
                            DataFlavor.stringFlavor)) {
                    displayDropLocation(
                            "List doesn't accept a " +
                            "drop of this type.");
                    return false;
                }
                JList.DropLocation dl = (JList.DropLocation)
                                     info.getDropLocation();
                DefaultListModel<String> listModel =
                    (DefaultListModel<String>)
                    list.getModel();
                int ind = dl.getIndex();
                boolean insert = dl.isInsert();
                // Get the current string under the drop.
                String value = listModel.getElementAt(ind);

                // Get the string that is being dropped.
                Transferable t = info.getTransferable();
                String data;
                try {
                    data = (String)t.getTransferData(
```

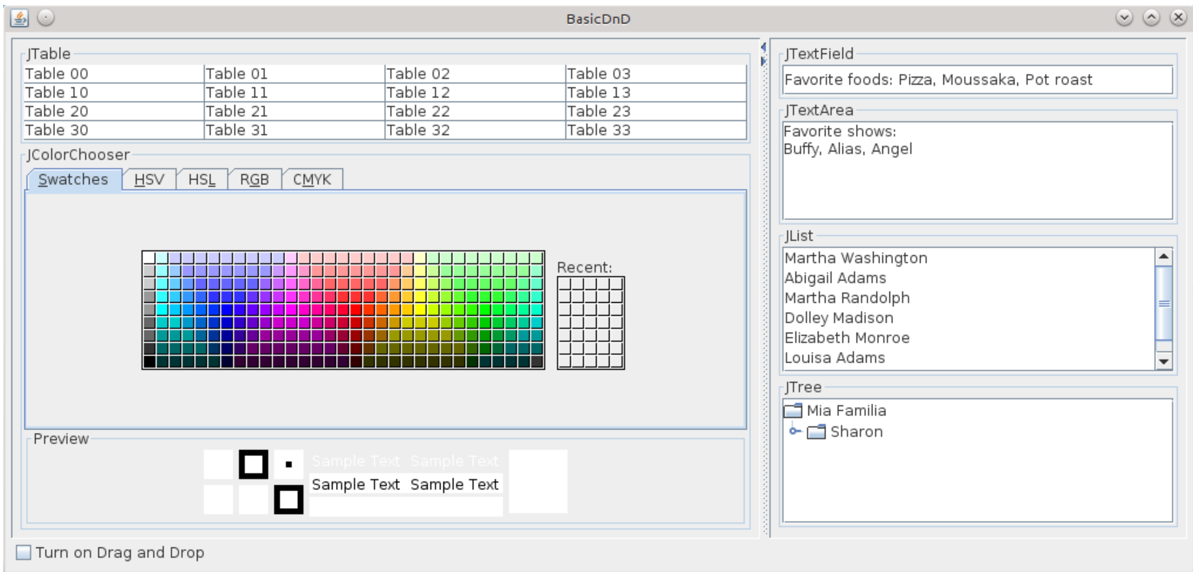
```
                        DataFlavor.stringFlavor);
            }
            catch (Exception e) { return false; }

            // Display a dialog with drop information.
            String dropValue = "\"" +
                                   data + "\" dropped ";
            if (dl.isInsert()) {
                if (dl.getIndex() == 0) {
                    displayDropLocation(dropValue +
                            "at beginning of list");
                } else if (dl.getIndex() >=
                           list.getModel().getSize()) {
                    displayDropLocation(
                            dropValue +
                            "at end of list");
                } else {
                    String value1 =
                        list.getModel()
                       .getElementAt(dl.getIndex() - 1);
                    String value2 = list.getModel()
                                  .getElementAt(dl
                                  .getIndex());
                    displayDropLocation(dropValue +
                            "between \"" + value1 +
                            "\" and \"" + value2 +
                            "\"");
                }
            } else {
                displayDropLocation(dropValue +
                        "on top of " + "\"" +
                        value + "\"");
            }
                    return false;
        }

        public int getSourceActions(JComponent c) {
            return COPY;
        }

        @SuppressWarnings("unchecked")
        protected Transferable createTransferable(
                JComponent c) {
            JList<String> list = (JList<String>)c;
            Object[] values =
                list.getSelectedValuesList().toArray();

            StringBuffer buff = new StringBuffer();

            for (int i = 0; i < values.length; i++) {
```

```
                    Object val = values[i];
                    buff.append(val == null
                            ? ""
                            : val.toString());
                    if (i != values.length - 1) {
                        buff.append("\n");
                    }
                }
                return new StringSelection(buff.toString());
            }
        });
        list.setDropMode(DropMode.ON_OR_INSERT);

        JScrollPane listView = new JScrollPane(list);
        listView.setPreferredSize(new Dimension(300, 100));
        rightPanel.add(createPanelForComponent(listView,
                                                "JList"));

        //Create a tree.
        DefaultMutableTreeNode rootNode =
            new DefaultMutableTreeNode("Mia Familia");
        DefaultMutableTreeNode sharon =
            new DefaultMutableTreeNode("Sharon");
        rootNode.add(sharon);
        DefaultMutableTreeNode maya =
            new DefaultMutableTreeNode("Maya");
        sharon.add(maya);
        DefaultMutableTreeNode anya =
            new DefaultMutableTreeNode("Anya");
        sharon.add(anya);
        sharon.add(new DefaultMutableTreeNode("Bongo"));
        maya.add(new DefaultMutableTreeNode("Muffin"));
        anya.add(new DefaultMutableTreeNode("Winky"));
        DefaultTreeModel model =
            new DefaultTreeModel(rootNode);
        tree = new JTree(model);
        tree.getSelectionModel().setSelectionMode
              (TreeSelectionModel
              .DISCONTIGUOUS_TREE_SELECTION);
        JScrollPane treeView = new JScrollPane(tree);
        treeView.setPreferredSize(new Dimension(300, 100));
        rightPanel.add(createPanelForComponent(treeView,
                                                "JTree"));

        //Create the toggle button.
        toggleDnD = new JCheckBox("Turn on Drag and Drop");
        toggleDnD.setActionCommand("toggleDnD");
        toggleDnD.addActionListener(this);

        JSplitPane splitPane = new JSplitPane(
```

```java
                JSplitPane.HORIZONTAL_SPLIT,
                leftPanel, rightPanel);
        splitPane.setOneTouchExpandable(true);

        add(splitPane, BorderLayout.CENTER);
        add(toggleDnD, BorderLayout.PAGE_END);
        setBorder(BorderFactory.createEmptyBorder(5,5,5,5));
    }

    protected JPanel createVerticalBoxPanel() {
        JPanel p = new JPanel();
        p.setLayout(new BoxLayout(p, BoxLayout.PAGE_AXIS));
        p.setBorder(BorderFactory
                    .createEmptyBorder(5,5,5,5));
        return p;
    }

    public JPanel createPanelForComponent(JComponent comp,
                                          String title) {
        JPanel panel = new JPanel(new BorderLayout());
        panel.add(comp, BorderLayout.CENTER);
        if (title != null) {
            panel.setBorder(
                BorderFactory.createTitledBorder(title));
        }
        return panel;
    }

    private void displayDropLocation(final String string) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                JOptionPane.showMessageDialog(null, string);
            }
        });
    }

    public void actionPerformed(ActionEvent e) {
        if ("toggleDnD".equals(e.getActionCommand())) {
            boolean toggle = toggleDnD.isSelected();
            textArea.setDragEnabled(toggle);
            textField.setDragEnabled(toggle);
            list.setDragEnabled(toggle);
            table.setDragEnabled(toggle);
            tree.setDragEnabled(toggle);
            colorChooser.setDragEnabled(toggle);
        }
    }

    private static void createAndShowGUI() {
        frame = new JFrame("BasicDnD");
```

```java
        frame.setDefaultCloseOperation(
                JFrame.EXIT_ON_CLOSE);

        JComponent newContentPane = new BasicDnD();
        newContentPane.setOpaque(true);
        frame.setContentPane(newContentPane);

        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(
            new Runnable() {
                public void run() {
                        UIManager.put("swing.boldMetal",
                                   Boolean.FALSE);
                createAndShowGUI();
            }
        });
    }
}
```

which produces



All components are ultimately derived from the abstract class **Component** which defines many methods common for all components. These are mainly getters and setters for properties, such as sizes, colors etc. They are named according to the following convention: for property **prop** the getter is named **getProp** while the setter will be **setProp**. If a property is of logical type (**true** or **false**), then instead of **getProp**, we rather use **isProp**.

Let us mention some of these properties:

- size — determined by an object of type Dimension (from *java.awt*) with fields describing width and height (getWidth, getHeight);
- minimumSize, maximumSize, preferredSize — determined by an object of type Dimension and taken into account by layout managers (not always, though);
- width — width of the component;
- height — height of the component;
- bounds — determined by an object of type Rectangle (from *java.awt*) with fields describing coordinates x and y of the upper-left corner and width and height of the component;
- location — determined by an object of type Point (from *java.awt*) with fields describing coordinates x and y of the upper-left corner of the component;
- alignmentX, alignmentY — determined by a float; indicates alignment along the axes;
- font — determined by an object of type Font from *java.awt* (see below);
- background, foreground — determined by objects of type Color from *java.awt* (see below);
- parent — a Container which contains a given component (read only);
- name — the name of this component; if not set, the system will provide a default one;
- visible — boolean value indicating if the component is visible;
- lightweight — boolean value indicating if this component is lightweight;
- opaque — boolean value indicating if this component is opaque;
- enabled — boolean value indicating if this component can react to events (as, for example, mouse clicks).

One has to remember that all sizes of components are not known until they are 'realized' — this usually happens when methods setSize or pack or setVisible is called on the frame window.

Coordinates of components are alway expressed in the coordinate system where the point $(0, 0)$ corresponds to the upper-left corner; x-coordinate goes from left to right, while y-coordinate goes *downwards* (*sic!*).

Locations and sizes of components inside a container are normally calculated by a layout manager — we can suggest our preferences by setting them 'by hand', but that is only a suggestion... However, we *can* set the size of the main frame window by invoking, e.g., `frame.setSize(200, 200)` on it. The sizes of all components inside it will then be determined by a layout manager. Or, we can pack the main window (`frame.pack()`) and its size will be determined by its contents.

Fonts are specified by objects of type Font (from *java.awt*); its main constructor takes three arguments

```
new Font(String name, int style, int size)
```

where

- name is the font name (case insensitive). However, specifying a concrete font name may be a little bit risky, because we don't know if such a font is available on the user's system. Therefore, we can only specify a generic (logical) name of the font we want — there are five such names: Dialog, DialogInput, Monospaced, SansSerif and Serif. The most appropriate font from those installed on a given system will then be selected.

- style is a static final integer defined in class **Font**: Font.PLAIN, Font.BOLD or Font.ITALIC. They can be “OR'ed”, e.g. `Font.BOLD | Font.ITALIC`.
- size is an integer specifying the size (in points, where point is 1/72 of an inch).

Colors are described by objects of class **Color** (from ***java.awt***). The main constructor takes three integers specifying red, green and blue components (or four integers, with transparency, the so called α-channel, added). They all should have values from the range $[0, 255]$. We can create such object like this:

```
new Color(255,33,33)
```

However, the most popular colors are already defined as static fields of class **Color** that are themselves objects of this class: these are BLACK, BLUE, CYAN, DARK_GRAY, GRAY, GREEN, LIGHT_GRAY, MAGENTA, ORANGE, PINK, RED, WHITE, and YELLOW.

Components can react to events like mouse clicks or pressing a key when they have the focus. This feature can be dynamically disabled or enabled (by invoking `comp.setEnabled(false)` or `comp.setEnabled(true)`).

12.3 Swing program

Let us now consider a very simple example:

Listing 95 MBC-HelloG/HelloWorldG.java

```
import java.awt.Color;
import java.awt.Font;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class HelloWorldG {
    public static void main(String[] args) {
        //
        // Should be on the EDT!
        //
        JFrame fr = new JFrame("HELLO");
        fr.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JLabel label = new JLabel("Hello, World");
          // all properties have reasonable defaults, but...
        label.setFont(new Font("Serif",Font.BOLD,70));
        label.setBackground(Color.ORANGE);
        label.setOpaque(true);
        label.setForeground(new Color(0,0,102));

        fr.add(label); // frame.getContentPane().add(label)

        fr.pack();
          // fr.setSize(600,400);
        fr.setLocationRelativeTo(null);
```

```
        fr.setVisible(true);
    }
}
```

which displays

Main points to notice here

- fr represents the frame window: it is a heavyweight container of type JFrame into which we will put all components.
- By invoking setDefaultCloseOperation we specify what will happen when the window is closed. This is determined by an integer (it should be an enum...) defined in class JFrame (strictly speaking inherited by implementing the interface WindowConstants)): EXIT_ON_CLOSE, DISPOSE_ON_CLOSE, DO_NOTHING_ON_CLOSE or HIDE_ON_CLOSE.
- Components may be created in any order; when created, they are just objects in memory, not related to other objects.
- Each existing object can be configured separately (color, font, etc.).
- To put one component into another, we use add on the parent component, passing a child component as the argument.
- Exact sizes of all components are unspecified until pack (or setSize) is invoked on the main window. At this moment, all sizes are calculated and components are arranged inside the frame window (normally this is performed by a layout manager).
- In order to make the window with its contents visible on the screen, we have to call setVisible on it.

12.4 Delegation Event model

Normally, we want our GUI to be responsive — we would like something to happen, for example, when the user clicks the mouse on a button visible on the screen. For this to be possible, any Swing application has to 'listen' to events generated by the system: mouse clicks or moves, key presses, etc. These events are provided by the operating system and can be intercepted by the JVM, which wraps them into objects of a type derived from EventObject and enqueues them on a special FIFO queue for the **Event-Dispatch Thread** (EDT) to process. When the time comes, the EDT pops them from the queue and passes them to listeners.
Events are handled by invoking call-back methods on objects which have been registered as **listeners** of events originating from a given source (e.g., a graphical component, like a button) and of a specified type.

In order to handle events, we need

- a source of events; this can be a graphical component of our GUI or an object representing some data structure. The source holds a collection of its 'listeners'.
- a way to add (and remove) listeners of events of a specified type and taking place on a given source;
- listeners — objects of classes implementing an appropriate interface and therefore providing definitions of its abstract methods; these methods will be invoked by source objects on listeners as a reaction to an event.

The scheme as described above is called the **delegation event model**.

To one source, we can attach many listeners, and the other way around: one listener can be attached to many sources. In order to delegate a listener as a handler of events, we usually invoke a special method

```
source.addXXXListener(listener);
```

where XXX specifies the type of events we are interested in; it can be, for example, Action, Mouse, MouseMotion, Key etc. The reference source refers to an object that has the ability to fire events of the specified type. As the argument, we pass an object which can handle events of the given type: its class has to implement a special interface which declares (as abstract methods) actions that are to be executed after an event of the given type occurred on the given source. Event-handling methods are **public void** and they accept one argument: an object which carries information on the event, as, for example, the source of the event, time of occurrence, and other properties depending on the type of this particular event.

To avoid conflicts, the EDT should be the *only* thread which directly interacts (and therefore can modify) the GUI: it is very important to remember that after the EDT has been started

(almost) all operations modifying the GUI should be executed on the event dispatch thread.

As the standard doesn't state clearly when exactly the EDT is launched, it is recommended to perform *all* operations on Swing components, even before displaying them on the screen, on the EDT. We can do it by invoking the static method **invokeLater(Runnable)** from class **SwingUtilities** (or **EventQueue**). The method takes a **Runnable** and the operations we want to perform have to be contained in its **run** method: we can do it by passing an object of our own class implementing **Runnable**, or an object of an anonymous class, or a lambda, because **Runnable**, having only one abstract method **run**, is a functional interface:

```
SwingUtilities.invokeLater(new MyRunnable(...));

SwingUtilities.invokeLater(new Runnable() {
    @Override
    public void run() {
        // ...
    }
});

SwingUtilities.invokeLater( () -> {
    // ...
});
```

Our **Runnable** will be wrapped in an object representing an event and inserted into the event queue, from where it will be popped and executed *on the EDT*.

Let us consider an example:

Listing 96 MBH-Events/Events.java

```java
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;

public class Events {
    public static void main(String[] args) {
        SwingUtilities.invokeLater( () -> createGUI() );
    }
    private static void createGUI() {
        JFrame f = new JFrame("Events");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

          // these will be sources
        JButton b1 = new JButton("Button one");
        JButton b2 = new JButton("Button two");
        JButton ex = new JButton("EXIT");

          // listener (as a lambda) implementing
          // public void actionPerformed(ActionEvent)
        ActionListener lis = e -> {
            String s = ((JButton)e.getSource()).getText();
            System.out.println(s + " clicked");
        };

          // registering one listener with two buttons
        b1.addActionListener(lis);
        b2.addActionListener(lis);
          // registering listener of 'exit' button
        ex.addActionListener(e -> System.exit(0));

        JPanel p = new JPanel();
          // adding buttons to the panel
        p.add(b1);
        p.add(b2);
        p.add(ex);
          // adding panel to the frame
        f.add(p);
          // now all sizes will be calculated;
          // do not add anything after packing!
        f.pack();
          // the window will be centerd on the screen
        f.setLocationRelativeTo(null);
```

```
            // show the window
            f.setVisible(true);
        }
    }
```

which displays a simple GUI

reacting to clicks on the buttons.

12.5 Layouts

Each container has, associated with it, a layout manager which is responsible for arranging components contained in the container — initially and also after resizing it. The managers are objects of classes implementing the interface **LayoutManager** from *java.awt*. They can (but do not have to) take into account sizes suggested by the user who may invoke, on any component, methods **setPreferredSize**, **setMaximumSize** and **setMinimumSize** passing an object of class **Dimension** (with only two fields: width and height).

It can happen that sizes and locations of components in a given container change or a new child components is added: in such situation calling **revalidate** may help (if not, try also **repaint**). Also, calling **pack** on the parent widow will recalculate all sizes and locations of the child components.

On any container, we can invoke **setLayout** passing an object representing a layout manager. There are five main layout managers that we can use. In fact there are more, but others are harder to use; they are extensively used by graphical tools which automate the process of building the GUI. However, the five basic managers that we will present are quite sufficient in vast majority of cases and are easy to use in programs written 'by hand'.

12.5.1 FlowLayout

Components added to a container (by invoking **add(component)** on it) will be arranged in one row, from left to right; if there is no room for a component in the current row, the second row will be added, and so on. The **FlowLayout** class has three constructors:

- `FlowLayout(int align, int hgap, int vgap)` — the first argument determines the alignment; it is an integer constant from class **FlowLayout**: LEFT, CENTER (the default) or RIGHT. The other two integers specify horizontal and vertical (if there is more than one row) gaps between components, and also gaps between the components and the borders;

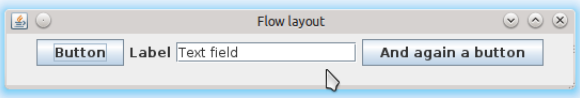
- `FlowLayout(int align)` — equivalent to `FlowLayout(align, 5, 5)`;
- `FlowLayout()` — equivalent to `FlowLayout(CENTER, 5, 5)`.

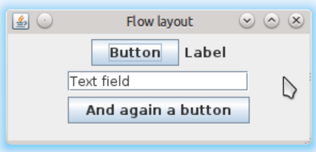
A simple example:

Listing 97 MBI-Flow/FlowEx.java

```
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class FlowEx extends JFrame {
    public static void main (String[] args) {
        SwingUtilities.invokeLater(() -> new FlowEx());
    }
    FlowEx() {
        super("Flow layout");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        add(new JButton("Button"));
        add(new JLabel("Label"));
        add(new JTextField("Text field", 15));
        add(new JButton("And again a button"));
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
}
```

displaying, depending on the width of the window, the components in one or more rows





12.5.2 GridLayout

In this layout, components will be added to a grid of rectangular cells of the same size (in the order row by row, in each row from left to right). The class **GridLayout** has three constructors:

- `GridLayout(int rows, int cols, int hgap, int vgap)` — the grid will have rows rows and cols columns with the specified horizontal and vertical gaps in-between. One (but not both), of rows and cols can be zero, which means 'as many as needed';

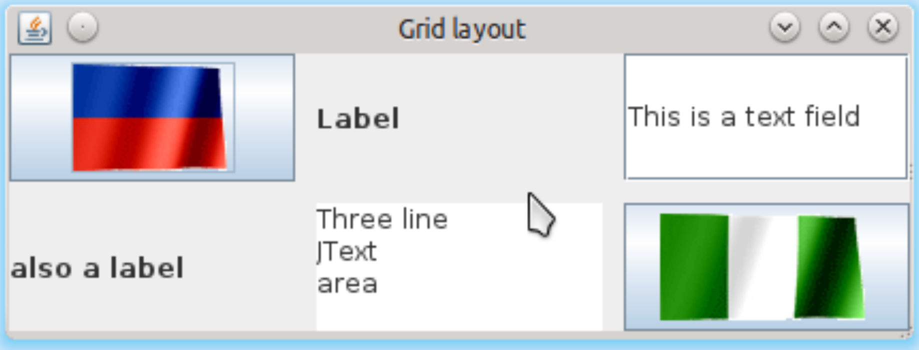
- `GridLayout(int rows, int cols)` — equivalent to `GridLayout(rows, cols, 0, 0);`
- `GridLayout()` — equivalent to `GridLayout(1, 0, 0, 0)` — all components will be added to one row (but they will be of equal sizes).

An example:

Listing 98 MBJ-Grid/GridEx.java

```java
import java.awt.GridLayout;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class GridEx extends JFrame {
    public static void main (String[] args) {
        SwingUtilities.invokeLater(() -> new GridEx());
    }
    GridEx() {
        super("Grid layout");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new GridLayout(2, 3, 10, 10));
        add(new JButton(new ImageIcon("haiti.gif")));
        add(new JLabel("Label"));
        add(new JTextField("This is a text field"));
        add(new JLabel("also a label"));
        add(new JTextArea("Three line\nJText\narea", 3, 10));
        add(new JButton(new ImageIcon("nigeria.gif")));
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
}
```

displays



12.5.3 BorderLayout

A container with this layout is divided into five areas: NORTH, SOUTH, WEST, EAST and CENTER (corresponding to integer constants in the class **GridLayout** with these names). We add components by calling **add(component,where)**, where where is one of the constants just mentioned, or CENTER if not specified. Each area can hold only one component, but this component may contain inside it other components. The class **BorderLayout** has only two constructors:

- `BorderLayout(int hgap, int vgap)` — the arguments specify horizontal and vertical gaps between components;
- `BorderLayout()` — equivalent to `BorderLayout(0, 0)`.

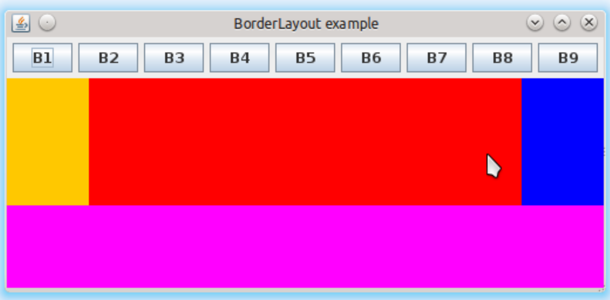
When a container with the border layout is resized, the *height* of the NORTH and SOUTH areas are kept constant, while for WEST and EAST their *width* is constant. The CENTER area is scaled in both directions and 'swallows' all empty areas.
This is illustrated in the example below:

Listing 99 MBG-BrdLay/BorderLayoutEx.java

```java
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class BorderLayoutEx extends JFrame {
    public static void main (String[] args) {
        new BorderLayoutEx("BorderLayout example");
    }
    BorderLayoutEx(String title) {
        super(title);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new BorderLayout());
        JPanel north = new JPanel();
        north.setLayout(new FlowLayout(FlowLayout.CENTER));
```

```java
        for (int i = 1; i <= 9; ++i)
            north.add(new JButton("B" + i));
        add(north, BorderLayout.NORTH);
        add(getPanel(Color.RED),     BorderLayout.CENTER);
        add(getPanel(Color.MAGENTA), BorderLayout.SOUTH);
        add(getPanel(Color.ORANGE),  BorderLayout.WEST);
        add(getPanel(Color.BLUE),    BorderLayout.EAST);
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
    JPanel getPanel(Color c) {
        JPanel p = new JPanel();
        p.setBackground(c);
        p.setPreferredSize(new Dimension(70, 70));
        return p;
    }
}
```

The program displays



Note that when resizing the window, heights of northern and southern areas do not change and the same applies to widths of west and east areas — the center area consumes all available space.
Note also that you can send only one component to each of the five regions. This is not a problem, however, as you can, as we did in the example above, collect several components in one (very often a **JPanel**) and send it as single component.

12.5.4 Box layout

The box layout arranges components in one row or one column. Unlike **GridLayout**, it takes into account the preferred, minimum and maximum sizes of the components, as well as its X- or Y- alignments. In order to set the box layout for a component comp, we call

```java
comp.setLayout(new BoxLayout(comp, BoxLayout.X_AXIS)); // horizontal
comp.setLayout(new BoxLayout(comp, BoxLayout.Y_AXIS)); // vertical
```

and to set its alignment

```java
comp.setAlignmentX(align);
comp.setAlignmentY(align);
```

where align is of type **float** and may assume, for X-alignment, three values predefined as constants in class **Component**: LEFT_ALIGNMENT (0.0), CENTER_ALIGNMENT (0.5) and RIGHT_ALIGNMENT (1.0); for Y-alignment these are TOP_ALIGNMENT (0.0), CENTER_ALIGNMENT (0.5) and BOTTOM_ALIGNMENT (1.0). Components will be arranged in the order they have been added to a container (left to right or top to bottom). One can also add special ‘filling’ components between, below or above them (leftmost and rightmost area for for horizontal boxes). One of these ‘fillers’ is a rigid area which can be added by invoking

```
comp.add(Box.createRigidArea(new Dimension(x,y));
```

It represents fixed-sized gap between components: it will not change its size when the window is resized. On the other hand, one can add a ‘glue’ components which will all shrink or expand in the same way when resizing the window.

```
comp.add(Box.createGlue());
```

Let us see an example:

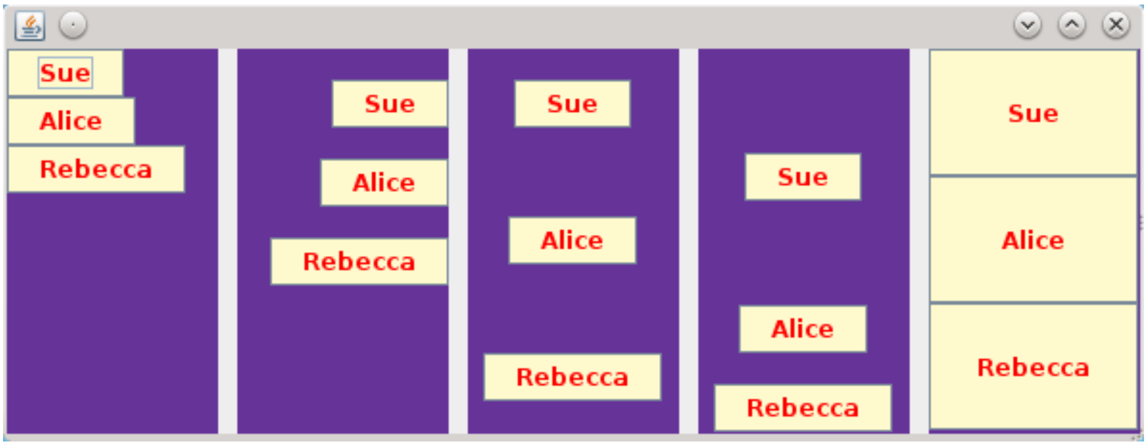
Listing 100 MCQ-BoxLay/Boxes.java

```java
import java.awt.Color;
import java.awt.Component;
import java.awt.Dimension;
import java.awt.GridLayout;
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import static java.awt.Component.LEFT_ALIGNMENT;
import static java.awt.Component.CENTER_ALIGNMENT;

public class Boxes extends JFrame {
    public static void main (String[] args) {
        new Boxes();
    }
    private Boxes() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new GridLayout(1,0,10,5));
        add(new MyBox(0));
        add(new MyBox(1));
        add(new MyBox(2));
        add(new MyBox(3));
        add(new MyBox(4));
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
}

```

```java
class MyBox extends JPanel {
    private static final Color rebecca =
                            new Color(0x66,0x33,0x99);
    private static final Color lfore = Color.RED;
    private static final Color lback =
                            new Color(0xFF,0xFA,0xCD);
    private static final int xsize = 110, ysize = 200;
    private static final float[] align =
        {LEFT_ALIGNMENT, RIGHT_ALIGNMENT, CENTER_ALIGNMENT,
         CENTER_ALIGNMENT, CENTER_ALIGNMENT};
    private static final Dimension rig =
                                new Dimension(0,ysize/12);
    String small = "Sue", medium = "Alice", big = "Rebecca";

    MyBox(int num) {
        setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
        setPreferredSize(new Dimension(xsize,ysize));
        setBackground(rebecca);

          // before the small button
        switch (num) {
            case 0:                                break;
            case 1:
            case 2: add(Box.createRigidArea(rig)); break;
            case 3: add(Box.createGlue());         break;
            case 4:                                break;
        }

        add(addButton(small, num));

          // between the small and medium buttons
        switch (num) {
            case 0:                                break;
            case 1: add(Box.createRigidArea(rig)); break;
            case 2:
            case 3: add(Box.createGlue());         break;
            case 4:                                break;
        }

        add(addButton(medium, num));

          // between the medium and big buttons
        switch (num) {
            case 0:                                break;
            case 1: add(Box.createRigidArea(rig)); break;
            case 2: add(Box.createGlue());         break;
            case 3: add(Box.createRigidArea(rig)); break;
            case 4:                                break;
        }

```

```java
        add(addButton(big, num));

          // after the big button
        switch (num) {
            case 0:
            case 1:                                     break;
            case 2: add(Box.createRigidArea(rig)); break;
            case 3:
            case 4:                                     break;
        }
    }

    private JButton addButton(String txt, int num) {
        JButton but = new JButton(txt);
        but.setForeground(lfore);
        but.setBackground(lback);
        but.setOpaque(true);
        but.setAlignmentX(align[num]);
        if (num == 4)
            but.setMaximumSize(new Dimension(xsize,ysize));
        return but;
    }
}
```

The program displays



Notice, that if the maximum size is not defined or is sufficiently big, the component will occupy the whole available area (see the last column in the figure above).

12.5.5 GridBagLayout

Layout of type GridBagLayout, as those of type GridLayout, divide the area of the component with this layout into rectangular grid of cells. However, unlike GridLayout, it allows to put components into subareas consisting of a rectangular set of neighbouring cells.

Suppose comp is a component with GridBagLayout installed:

```java
comp.setLayout(new GridBagLayout());
```

To add a component into comp, we need an object of type **GridBagConstraints** which carries information where and how this component is to be added. So we create object of type **GridBagConstraints**, configure it, and then we pass this object when adding components to comp:

```
comp.setLayout(new GridBagLayout);
GridBagConstraints  cnstr = new GridBagConstraints();
  // configuring cnstr
comp.add(anotherComponent, cnstr);
```

Objects of type **GridBagConstraints** expose several public, modifiable fields, so configuring them consists of a series of assignments. For example

```
GridBagConstraints c = new GridBagConstraints();
c.fill = GridBagConstraints.BOTH;
c.gridx = 0;
c.gridy = 1;
c.weightx = 0.5; // important when resizing
c.weighty = 0.5;
c.gridheight = 2;
// ...
```

Some of the most important fields that we can set are:

- gridx, gridy — specify the x- and y-coordinates of the upper-left cell of the component's display area; coordinates are counted from zero to the right for x-coordinate and downwards for y-coordinate.
- gridwidth, gridheight — specifies the number of columns (gridwidth) and rows (gridheight) in the component's display area. The default values are 1, what corresponds to one cell at position specified by gridx and gridy. One can use GridBagConstraints.REMAINDER to specify that the component's display area will be from gridx to the last cell in the row (for gridwidth) or from gridy to the last cell in the column (for gridheight).
- fill — will be used if the component's display area (a cell or rectangular group of cells) is larger than the component's size. Possible values are defined in **GridBagConstraints** as constants NONE (leave the size of the components as is, the default), HORIZONTAL (the component is resized to fill its display area horizontally), VERTICAL (the component fills its display area vertically) and BOTH (the component fills its display area in both directions).
- ipadx, ipady — specify the padding around the component (in pixels). By setting non-zero values, we can make some cells larger; remember, however, that all cells of one row have always the same height and all cells in one column have the same width — setting non-zero padding for one cell, therefore, affects widths and heights of other cells.
- weightx, weighty — determine how to distribute space occupied by rows and columns when the whole grid is resized. If the values are 0 (which is the default), the grid will not be rescaled, but will be displayed in the center of the surrounding component. The values of this fields are of type **double** and lie in the range $[0, 1]$. Their rations determine how rows and columns are scaled. Larger values indicate that the component's row (or column) should get more space when resizing. For each column, its (horizontal) weight corresponds to the highest weightx of its cells; for each row its (vertical) weight is determined by the highest weighty of

its cells. Therefore, to make all cells scale uniformly, one can assign the same non-zero value (0.5, say) to weighty of all cells in one column, and to weightx of all cells in one row.

For the example of using the **GridBagLayout**, see the program in Listing 113 on page 198.

12.6 Using icons

In the next example, we show how to create icons from an existing graphic files:

Listing 101 — MBA-IntroSwing/IntroSwing.java

```java
package intro;

import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.Font;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.SwingConstants;
import javax.swing.SwingUtilities;

class IntroSwing  {

    public static void main(String[] args)  {
        SwingUtilities.invokeLater(() -> createGUI());
    }

    private static void createGUI()  {
        Class<IntroSwing> clz = IntroSwing.class;

          // Icons from directory img of the application
        Icon[] icon = {
              // root '/' is dir containing 'intro' package
            new ImageIcon(clz.getResource("/img/pl.gif")),
              // or relative to .class file
            new ImageIcon(clz.getResource("/img/fr.gif")),
            new ImageIcon(clz.getResource("/img/uk.gif")),
        };
          // text on buttons
        String[] descr = {"Poland", "France", "UK" };

        JFrame frame = new JFrame("Swing"); // main window
        frame.setLayout(new FlowLayout());  // layout of its
                                            // contentPane
        for (int i=0; i < icon.length; ++i) {
            JButton b = new JButton(descr[i], icon[i]);
            b.setFont(new Font("Dialog",
```

```
                         Font.BOLD | Font.ITALIC, 18));
            b.setForeground(Color.BLACK);
            b.setBackground(Color.WHITE);
              // position of text relative to icon
            b.setVerticalTextPosition(
                    SwingConstants.BOTTOM);
            b.setHorizontalTextPosition(
                    SwingConstants.CENTER);
            frame.add(b);
        }
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        frame.pack();
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }
}
```

The program displays

12.7 Drawing

All swing components inherit the method **paintComponent** from **JComponent**. We never call it directly — it is invoked automatically when a component must be repainted (e.g., after resizing, when it is exposed after being hidden behind other windows etc.). There are also two other functions which will be invoked automatically when a component is repainted: **paintChildren** and **paintBorder**, but we seldom have to override them. All these methods take as the argument a so called **graphic context** — object of class **Graphics** from ***java.awt*** (in fact **Graphics2D** extending **Graphics**). It represents, in a sense, an output device on which we can draw geometrical figures or strings — normally, whatever we paint on it, will appear on the component (but can be redirected to memory or a file). When specifying positions of graphic elements, we have to remember that the point with coordinates $(0, 0)$ is in the upper-left corner, and y-coordinate increases *downwards!*

Object of type **Graphics** allow us to:

- set properties of the graphic context;
- draw lines and simple geometrical figures;
- draw strings;
- insert pictures and images (objects of type **Image**).

Geometrical figures can be drawn by invoking

- `void drawLine(int x1, int y1, int x2, int y2)` — draws a line between (x_1, y_1) and (x_2, y_2);

- `void drawOval(int x, int y, int width, int height)` — draws an ellipse (circle) inscribed in a rectangle with upper-left vertex at (x, y) and given width and height;
- `void drawRect(int x, int y, int width, int height)` — draws a rectangle with upper-left vertex at (x, y) and given width and height.

Analogously, we can fill ovals and rectangles by

- `void fillOval(int x, int y, int width, int height)` — fills an ellipse (in parcular a circle) inscribed in a rectangle with upper-left vertex at (x, y) and given width and height;
- `void fillRect(int x, int y, int width, int height)` — fills a rectangle with upper-left vertex at (x, y) and given width and height.

Before each drawing or filling, the color can be changed by **setColor**; if not set, the current foreground color will be used.

The coordinates of pixels should be understood as coordinates of points *between* pixels; when we paint a pixel with coordinates (x, y), the pixel to the right and below this point is painted. For example, to draw a diagonal, we should invoke

```
drawLine(0, 0, getWidth()-1, getHeight()-1);
```

because the pixel referred to by coordinates (width, height) would be outside of the picture! On the other hand, when we fill an oval or a rectangle, pixels *inside* a region specified by coordinates is painted, so to fill the whole area of a component, we would invoke

```
fillRect(0, 0, getWidth(), getHeight());
```

Let us see an example: we draw small squares at the corners of a button. Notice that

when overriding the **paintComponent** function, it is necessary to invoke, in the first line, the same method from the superclass.

Listing 102 MBB-PrettyButton/PrettyButton.java

```java
import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;

class MyButton extends JButton {
    public MyButton(String txt) {
        super(txt);
        setFont(new Font("Dialog", Font.PLAIN, 24));
    }
    @Override
    public void paintComponent(Graphics g) {
```

```java
        super.paintComponent(g); // IMPORTANT!!!
        int w = getWidth();
        int h = getHeight();
        g.setColor(Color.red);
          // drawing the squares
        g.fillRect(0, 0, 10, 10);
        g.fillRect(w-10, 0, 10, 10);
        g.fillRect(0, h-10, 10, 10);
        g.fillRect(w-10, h-10, 10, 10);
    }
}

public class PrettyButton extends JFrame {
    public PrettyButton() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        add(new MyButton("A very pretty button"));
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }

    public static void main(String args[]) {
        SwingUtilities.invokeLater(() -> new PrettyButton());
    }
}
```

which produces

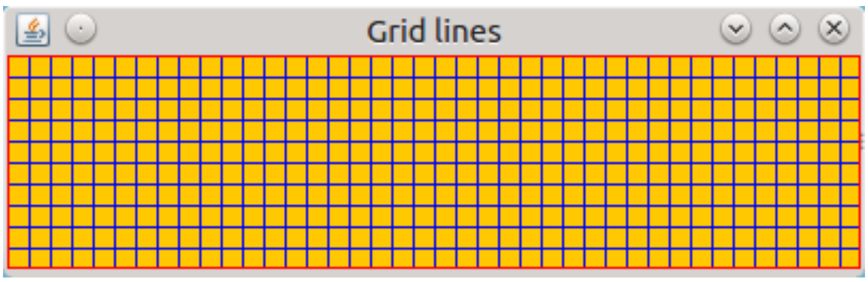
Another example demonstrates how to draw lines and rectangles:

Listing 103 MBE-GridLines/GridLines.java

```java
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;

public class GridLines extends JFrame {
    public GridLines() {
        super("Grid lines");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        add(new MyComponent(400, 100));
        pack();
```

```
        setLocationRelativeTo(null);
        setVisible(true);
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new GridLines());
    }
}

class MyComponent extends JComponent {
    public MyComponent(int w, int h) {
        Dimension d = new Dimension(w, h);
        setMinimumSize(d);
        setPreferredSize(d);
        setMaximumSize(d);
    }
    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int w = getWidth();
        int h = getHeight();
        g.setColor(Color.ORANGE);
        g.fillRect(0, 0, w, h);
        g.setColor(Color.RED);
        g.drawRect(0, 0, w-1, h-1);
        g.setColor(Color.BLUE);
        for (int y = 10; y < h-1; y += 10)
            g.drawLine(1, y, w-2, y);
        for (int x = 10; x < w-1; x += 10)
            g.drawLine(x, 1 , x, h-2);
    }
}
```

The program displays



and the window may be resized.

In the next example, we demonstrate how an image can be loaded, scaled and displayed (in fact, it can be done in several ways). The example also shows how to draw a string:

Listing 104 MCM-Drawing/Drawing.java

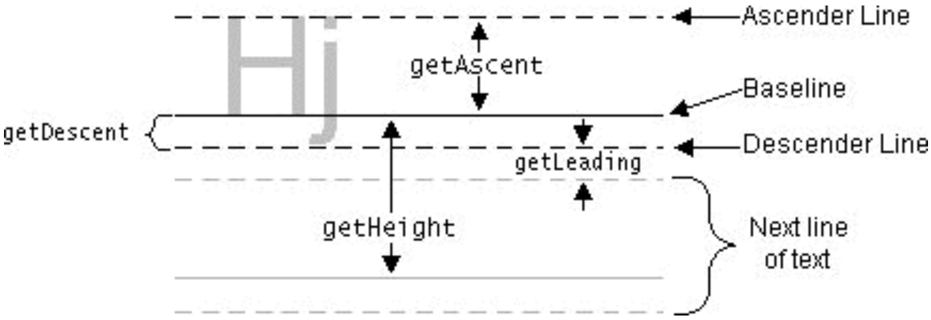
```java
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.RenderingHints;
import java.awt.image.BufferedImage;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Rectangle2D;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;

public class Drawing {
    public static void main (String[] args) {
        SwingUtilities.invokeLater(() -> new Drawing());
    }

    private Drawing() {
        JFrame fr = new JFrame("VERMEER");
        fr.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel panel = new MyPanel();
        //fr.setContentPane(panel);
        fr.add(panel);
        fr.pack();
        fr.setLocationRelativeTo(null);
        fr.setVisible(true);
    }
}

class MyPanel extends JPanel {
    int counter = 0;
    BufferedImage img = null;

    MyPanel() {
        setBackground(new Color(23,9,8));
        setForeground(Color.YELLOW);
        setOpaque(true);
        setFont(new Font("Sans Serif",
                    Font.BOLD | Font.ITALIC,40));
        setPreferredSize(new Dimension(600,350));
        try {
              // loading the image
```

```java
            img = ImageIO.read(new File("vermeer.png"));
        } catch (IOException e) {
            System.out.println("Image file not found");
            System.exit(1);
        }
    }
    @Override
    protected void paintComponent(Graphics g) {
          // not necessary, just to make things prettier
        Graphics2D g2 = (Graphics2D)g;
        g2.setRenderingHint(
                RenderingHints.KEY_STROKE_CONTROL,
                RenderingHints.VALUE_STROKE_PURE);
        g2.setRenderingHint(
                RenderingHints.KEY_ANTIALIASING,
                RenderingHints.VALUE_ANTIALIAS_ON);
        g2.setRenderingHint(
                RenderingHints.KEY_TEXT_ANTIALIASING,
                RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
          // could have been just
          // super.paintComponent(g);
        super.paintComponent(g2);

        String str = "Invoked " + ++counter + " times";
        FontMetrics fm = g2.getFontMetrics();
        Rectangle2D r = fm.getStringBounds(str,g2);

        int w = getWidth();
        int h = getHeight();
        int x = (w-(int)r.getWidth())/2;
        int y = h/6-(int)r.getHeight()/2+fm.getAscent();
        g2.drawString(str, x, y);
        g2.drawOval(0, 0, w, h/3);
        Image im = img.getScaledInstance( -1, h/2,
                               Image.SCALE_SMOOTH);
        g2.drawImage(im,(w-im.getWidth(null))/2,2*h/5,null);
    }
}
```

The program produces

The picture will be rescaled when the window is resized. Note that the counter shows how many times the **paintComponent** function has been invoked.

In the above program we used class **FontMetrics** to get information on properties of the string treated as a graphics; for example we can get sizes of the string rendered in a given font (as an object of type **Rectangle2D**). Strings rendered as a graphics have also other characteristics, shown in the picture



and available by invoking several methods of **FontMetrics**, like **getHeight**, **getAscent**, **getDescent**, **getLeading** and others. Note, that we cast **Graphics** object into **Graphics2D** — this is always safe, because the object passed to **paintComponent** *is* of this type. We did it in order to use methods of **Graphics2D** that are *not* inherited from **Graphics**: those methods yield a better quality of the displayed graphics. The program belows demonstrates the difference:

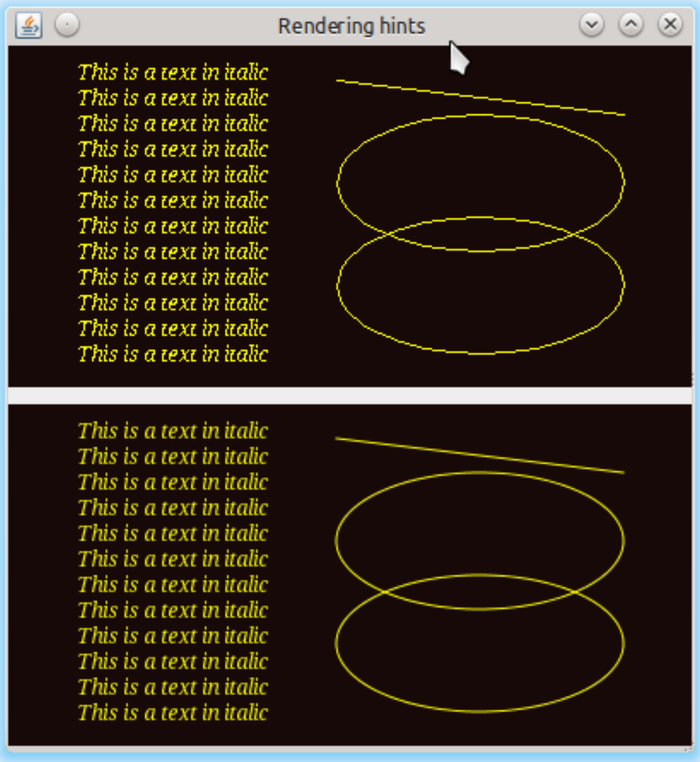
Listing 105 MCY-RenderHints/RenderHints.java

```java
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.GridLayout;
import java.awt.RenderingHints;
import java.awt.geom.Rectangle2D;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;

```

```
public class RenderHints {
    public static void main (String[] args) {
        SwingUtilities.invokeLater(() -> new RenderHints());
    }
    private RenderHints() {
        JFrame fr = new JFrame("Rendering hints");
        fr.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        fr.setLayout(new GridLayout(2,1,10,10));
        fr.add(new MyPanel(false));
        fr.add(new MyPanel(true));
        fr.pack();
        fr.setLocationRelativeTo(null);
        fr.setVisible(true);
    }
}

class MyPanel extends JPanel {
    boolean hintsOn;

    MyPanel(boolean hints) {
        hintsOn = hints;
        setBackground(new Color(23,9,8));
        setForeground(Color.YELLOW);
        setOpaque(true);
        setPreferredSize(new Dimension(400,200));
    }
    @Override
    //protected void paintComponent(Graphics g) {
    public void paintComponent(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        if (hintsOn) {
            g2.setRenderingHint(
                    RenderingHints.KEY_STROKE_CONTROL,
                    RenderingHints.VALUE_STROKE_PURE);
            g2.setRenderingHint(
                    RenderingHints.KEY_ANTIALIASING,
                    RenderingHints.VALUE_ANTIALIAS_ON);
            g2.setRenderingHint(
                    RenderingHints.KEY_TEXT_ANTIALIASING,
                    RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
        }
        super.paintComponent(g2);

        String str = "This is a text in italic";
        g2.setFont(new Font("Serif",Font.ITALIC,12));
        FontMetrics fm = g2.getFontMetrics();
        Rectangle2D r = fm.getStringBounds(str,g2);

        int w = getWidth();
        int h = getHeight();
```

```
            int x = w/10;
            int y = h/10;
            while (y < 0.95*h) {
                g2.drawString(str, x, y);
                y += (int)(r.getHeight()*1.1);
            }
            x = (int)(2*w/10+r.getWidth());
            y = h/10;
            g2.drawLine(x, y, 9*w/10, 2*h/10);
            g2.drawOval(x, 2*h/10, 9*w/10-x, 4*h/10);
            g2.drawOval(x, 5*h/10, 9*w/10-x, 4*h/10);
        }
    }
```

and the difference becomes apparent



12.8 Windows

Windows are containers of the highest level in hierarchy of containers/components — they are heavyweight and contain all other components (usually lightweight) of the whole GUI.
Some windows may have an owner (they are then called *secondary window*), some — *primary windows* — do not: they are always 'parents' for other containers/components.

- Closing a primary window closes all its children (secondary windows);
- Minimizing a primary window minimizes all its contents;

- Moving a primary window moves it with all its contents.

Only secondary windows can (but need not to) be modal; when a window is modal, interaction with the parent window is blocked until this modal window has been closed.

Let us mention some of the most important methods that we can call on windows:

- `void pack()` — 'packs' the window, i.e., calculates all sizes and locations of the child components taking into account their preferred sizes;
- `void setLocationRelativeTo(Component c)` — set location of this window on a specified component; center of the screen if `c` is **null**;
- `setDefaultCloseOperation(int)` specifies what will happen when the window is closed. This is determined by an integer defined in class JFrame: EXIT_ON_CLOSE, DISPOSE_ON_CLOSE, DO_NOTHING_ON_CLOSE or HIDE_ON_CLOSE;
- `Toolkit getToolkit()` — returns Toolkit which contains many useful methods describing the graphical environment of the application. The class constitutes a 'glue' between platform independent classes and native operating system;
- `boolean isShowing()` — tests whether the window is displayed on the screen;
- `void setCursor(Cursor)` — set the type of the cursor;
- `dispose()` — releases resources related to the widow;
- `Window getOwner()` — returns the owner of this window (or **null**);
- `Window[] getOwnedWindows()` — returns an array of owned (child) windows;
- `Component getFocusOwner()` — returns the component inside the widow that has the focus (if the focus is somewhere in this window);
- `Component getMostRecentFocusOwner()` — returns the component of the window that will receive the focus when the whole window will get it;

> The most important kind of window is the frame window (in Swing it is JFrame) which has borders, title bar, control icons, menu bar, tool bar etc.

A frame window has no owner and cannot be modal. It can be created with the default constructor or with a String argument specifying its title (that can be then dynamically modified).

Frames have many properties that can be get/set by appropriate methods; some of them are

- iconImage (setIconImage/getIconImage) of type Image — icon used on the task bar when the window is minimized;
- menuBar of type JMenuBar;
- title of type String — a string appearing on the title bar of the window;
- resizable of type boolean — specifies, if the window can be resized; may be modified dynamically;
- undecorated of type boolean — if **true**, the window has no 'decorations' (title bar, borders etc.);
- extendedState of type int — specifies the current state of the window as an integer constant from class JFrame: NORMAL, ICONIFIED, MAXIMIZED_HORIZ, MAXIMIZED_VERT, MAXIMIZED_BOTH. We can check whether a particular state is available on our platform by invoking Toolkit.isFrameStateSupported(int).

A very close cousin of **JFrame** is **JDialog**. It is intended to display some sort of a dialog allowing the user, for example, to enter some data. It alway has a parent, can (and usually should) be modal. As it represents some sort of a 'helper' widget, you cannot set its default close operations to EXIT_ON_CLOSE. Such dialogs are often created by invocation of static methods from class **JOptionPane** (many overloaded versions of **showInputDialog** or **showMessageDialog**). Modality can be set by invoking **setModalityType** — the corresponding property has enum type **Dialog.ModalityType**; use MODELESS to make the dialog non-modal, DOCUMENT_MODAL to make it modal (its parents will be blocked) or APPLICATION_MODAL (all root windows of an application will be blocked).

There are also windows of type **JInternalFrame** that are *lightweight*: they always have a parent and are contained inside another container. As they are lightweight, they can be completely platform independent and have additional properties lacking in heavyweight windows.

12.9 More examples

The example below illustrates various components, in particular a text area with scroll bars, and also **invokeLater** which is called from the main thread to modify the displayed GUI (append a line to the text area and possibly add a vertical scroll bar):

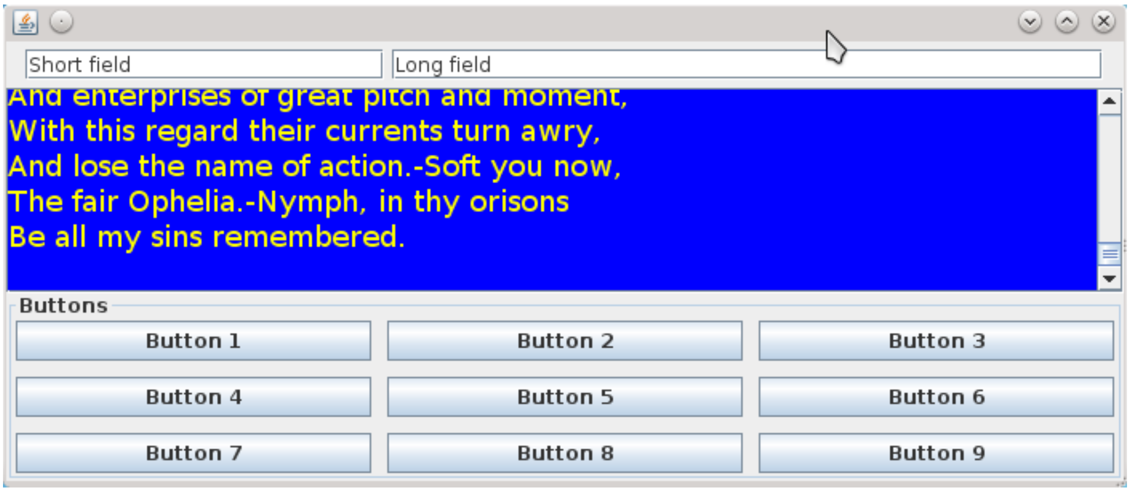
Listing 106 MCJ-Layouts1/Layouts.java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.GridLayout;
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class Layouts extends JFrame {
    static JTextArea area = null;

    public static void main (String[] args) {
        new Layouts();

        try (BufferedReader br =
                Files.newBufferedReader(
```

```java
                    Paths.get("hamlet.txt"))) {
            String line = null;
            while ( (line = br.readLine()) != null ) {
                String s = line;
                SwingUtilities.invokeLater(
                        () -> area.append(s+"\n"));
                Thread.sleep(500);
            }
            SwingUtilities.invokeLater( () -> {
                area.setBackground(Color.BLUE);
                area.setForeground(Color.YELLOW);
            });
        } catch(IOException | InterruptedException e) {
            e.printStackTrace();
            return;
        }
    }

    private Layouts() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
          // setting layout of the whole contentPane
          // of the frame window (it is border layout
          // by default anyway...)
        setLayout(new BorderLayout());

        JPanel southPanel = new JPanel();
        southPanel.setBorder(
            BorderFactory.createTitledBorder("Buttons"));
        southPanel.setLayout(new GridLayout(3,3,10,10));
        for (int i = 1; i < 10; ++i)
            southPanel.add(new JButton("Button " + i));
        add(southPanel,BorderLayout.SOUTH);

        JPanel northPanel = new JPanel();
        northPanel.setLayout(
                        new FlowLayout(FlowLayout.CENTER));
        northPanel.add(new JTextField("Short field",20));
        northPanel.add(new JTextField("Long field",40));
        add(northPanel,BorderLayout.NORTH);

        area = new JTextArea(15,40);
        area.setFont(
            new Font("Sans_Serif",Font.PLAIN,18));
        area.setBackground(Color.WHITE);
        area.setForeground(Color.BLACK);
        JScrollPane scroll = new JScrollPane(area,
                JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
                JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);
        add(scroll,BorderLayout.CENTER);

```

```
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
}


```

The program displays



Another example is similar: here we create more complex borders, in particular compound borders:

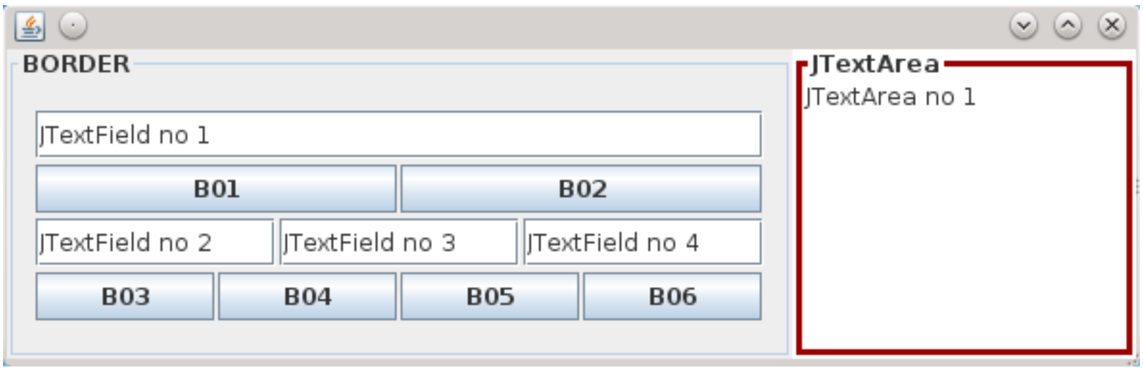
Listing 107 MCK-Layouts/Layouts.java

```java
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.GridLayout;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class Layouts extends JFrame {
    public static void main(String[] args) {
        new Layouts();
    }

    Layouts() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setContentPane(new MainPanel());
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
```

```
    }
}

class MainPanel extends JPanel {
    MainPanel() {
        JTextField[] tft = new JTextField[4];
        for (int i = 0; i < tft.length; i++) {
            tft[i]=new JTextField(11);
            tft[i].setText("JTextField no "+(i+1));
        }

        JButton[] bt = new JButton[6];
        for (int i = 0; i < bt.length; i++)
            bt[i]=new JButton(String.format("B%02d",i+1));

        JPanel[] panels = new JPanel[6];
        for (int i = 0; i < panels.length; i++)
            panels[i] = new JPanel();

        JTextArea tat = new JTextArea(3,15);
        tat.setText("JTextArea no 1");
        tat.setBorder(
            BorderFactory.createTitledBorder(
                BorderFactory.createLineBorder(
                    new Color(0x99,0,0),3
                ),
                "JTextArea"
            )
        );

        // two buttons in a row
        panels[0].setLayout(new GridLayout(1,2,2,2));
        panels[0].add(bt[0]);
        panels[0].add(bt[1]);

        // four buttons in a row
        panels[1].setLayout(new GridLayout(1,4,2,2));
        for (int i = 2; i < 6; i++)
            panels[1].add(bt[i]);

        // one text field in a row
        panels[2].setLayout(new BorderLayout());
        panels[2].add(tft[0],BorderLayout.CENTER);

        // three text fields in a row
        panels[3].setLayout(new GridLayout(1,3,2,2));
        for (int i = 1; i < 4; i++)
            panels[3].add(tft[i]);

        // text area
```

```java
        panels[4].setLayout(new BorderLayout());
        panels[4].add(tat,BorderLayout.CENTER);

        // buttons and text fields
        panels[5].setLayout(new GridLayout(4,1,5,3));
        panels[5].add(panels[2]);
        panels[5].add(panels[0]);
        panels[5].add(panels[3]);
        panels[5].add(panels[1]);
        panels[5].setBorder(
            BorderFactory.createCompoundBorder(
                  // outer
                BorderFactory.createTitledBorder("BORDER"),
                  // inner
                BorderFactory.createEmptyBorder(15,10,15,10)
            )
        );

        setLayout(new BorderLayout());
        add(panels[5], BorderLayout.CENTER);
        add(panels[4], BorderLayout.EAST);
    }
}
```

The program displays



In the example below, we build even more complex GUI

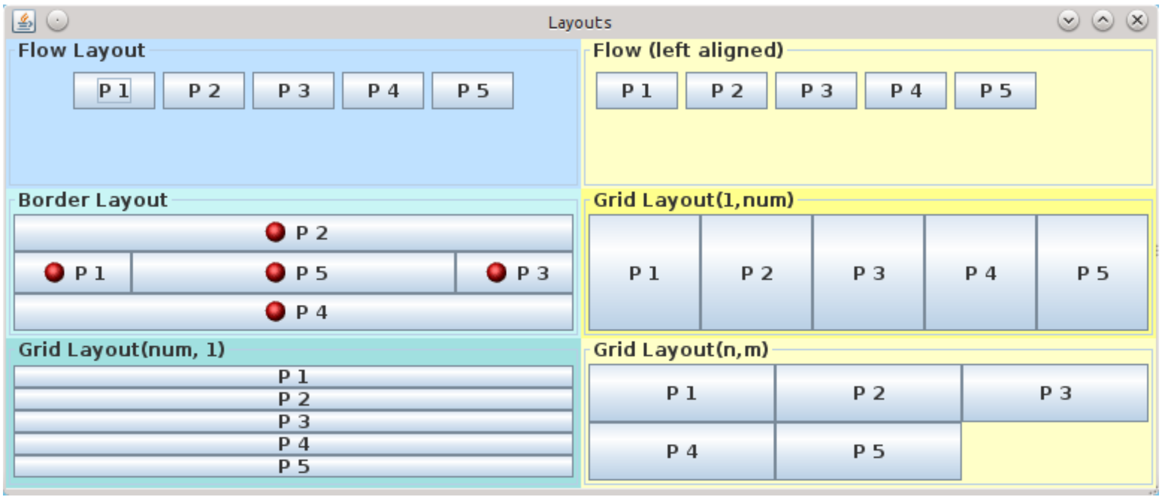
Listing 108 MCL-Layouts2/MiscLayouts.java

```java
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.LayoutManager;
import java.awt.Color;
import javax.swing.BorderFactory;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JButton;
```

```java
import javax.swing.JFrame;
import javax.swing.JPanel;

public class MiscLayouts {
    public static void main(String[] args) {
          // number of comonents in panels
        final int CNUM = 5;
          // descriptions
        String lmNames[] = {
            "Flow Layout", "Flow (left aligned)",
            "Border Layout", "Grid Layout(1,num)",
            "Grid Layout(num, 1)", "Grid Layout(n,m)"
        };
          // layouts
        LayoutManager lm[] = {
            new FlowLayout(),
            new FlowLayout(FlowLayout.LEFT),
            new BorderLayout(),
            new GridLayout(1, 0),
            new GridLayout(0, 1),
            new GridLayout(2, 0)
        };

          // for BorderLayout
        String gborders[] = {
                BorderLayout.WEST,
                BorderLayout.NORTH,
                BorderLayout.EAST,
                BorderLayout.SOUTH,
                BorderLayout.CENTER
        };
          // panel colors
        Color colors[] = {
            new Color(191, 225, 255),
            new Color(255, 255, 200),
            new Color(201, 245, 245),
            new Color(255, 255, 140),
            new Color(161, 224, 224),
            new Color(255, 255, 200)
        };

          // icon on a button
        Icon redDot = new ImageIcon("red.gif");

        JFrame frame = new JFrame("Layouts");
        frame.setLayout(new GridLayout(0, 2));

        for (int i = 0; i < lmNames.length; i++) {
            JPanel p = new JPanel();
            p.setBackground(colors[i]);
```

```
                p.setBorder(BorderFactory
                        .createTitledBorder(lmNames[i]));
                p.setLayout(lm[i]);
                Icon icon = null;

                if (lm[i] instanceof BorderLayout) icon = redDot;
                for (int j = 0; j < CNUM; j++) {
                    JButton b = new JButton("P " + (j+1), icon);
                    p.add(b, gborders[j]);
                }
                frame.add(p);
            }
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.pack();
            frame.setLocationRelativeTo(null);
            frame.setVisible(true);
        }
    }
```

The program displays



The next example shows how to set mnemonics and how to connect a label with another component:

Listing 109 MCH-LabsFor/LabelsFor.java

```
import java.awt.BorderLayout;
import java.awt.GridLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

class LabelsFor extends JFrame {
    JPanel panel = new JPanel(new GridLayout(0, 2, 10, 5));

```

```
    public static void main(String args[]) {
        new LabelsFor();
    }

    LabelsFor() {
        setLayout(new BorderLayout()); // not needed here
        String html = "<html><center>Please<br>"
            + "<b><font color=red>enter</font></b><br>"
            + "<font color=blue>your personal data"
            + "</font></center><br></html>";
        JLabel head = new JLabel(html, JLabel.CENTER);
        add(head, BorderLayout.NORTH);
        addLabAndTxtFld("Name", 'n', "Enter your name");
        addLabAndTxtFld("Year of birth", 'y', "YYYY/MM/DD");
        addLabAndTxtFld("Address", 'a', "Your address");
        add(panel, BorderLayout.CENTER);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }

    void addLabAndTxtFld(String txt, char mnem, String tip){
        JLabel lab = new JLabel(txt, JLabel.RIGHT);
        JTextField tf = new JTextField(20);
        tf.setToolTipText(tip);
        lab.setLabelFor(tf);
        lab.setDisplayedMnemonic(mnem);
        panel.add(lab);
        panel.add(tf);
    }
}
```

The program displays

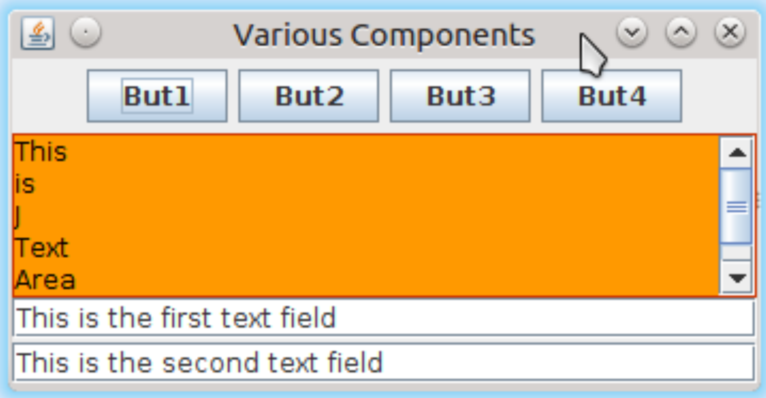
And one more example with **JScrollPane**:

Listing 110 MCG-Components/VariousComponents.java

```java
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class VariousComponents extends JFrame {
    public static void main(String[] args) {
        new VariousComponents();
    }

    VariousComponents() {
        super("Various Components");
        setDefaultCloseOperation(EXIT_ON_CLOSE);

          // this is the default anyway
        setLayout(new BorderLayout());

        JPanel lower = new JPanel();
        lower.setLayout(new GridLayout(2,1,5,2));
        JTextField tup = new JTextField(30);
        JTextField tdn = new JTextField(30);
        tup.setText("This is the first text field");
        tdn.setText("This is the second text field");
        lower.add(tup);
        lower.add(tdn);
        add(lower,BorderLayout.SOUTH);

        JPanel upper = new JPanel();
        upper.setLayout(new FlowLayout());
        JButton b1 = new JButton("But1");
        JButton b2 = new JButton("But2");
        JButton b3 = new JButton("But3");
        JButton b4 = new JButton("But4");
        upper.add(b1);
        upper.add(b2);
        upper.add(b3);
        upper.add(b4);
        add(upper,BorderLayout.NORTH);

        JTextArea ja = new JTextArea(5,30);
```

```java
        ja.setBackground(new Color(255,153,0));
        ja.setForeground(Color.BLACK);
        ja.setText("This\nis\nJ\nText\nArea\n!");
        JScrollPane sc = new JScrollPane(ja,
                JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
                JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);
        sc.setBorder(
            BorderFactory.createLineBorder(
                                          new Color(204,51,0)));
        add(sc,BorderLayout.CENTER);

        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                pack();
                setLocationRelativeTo(null);
                setVisible(true);
            }
        });
    }
}
```

The program displays



The next example just shows various borders:

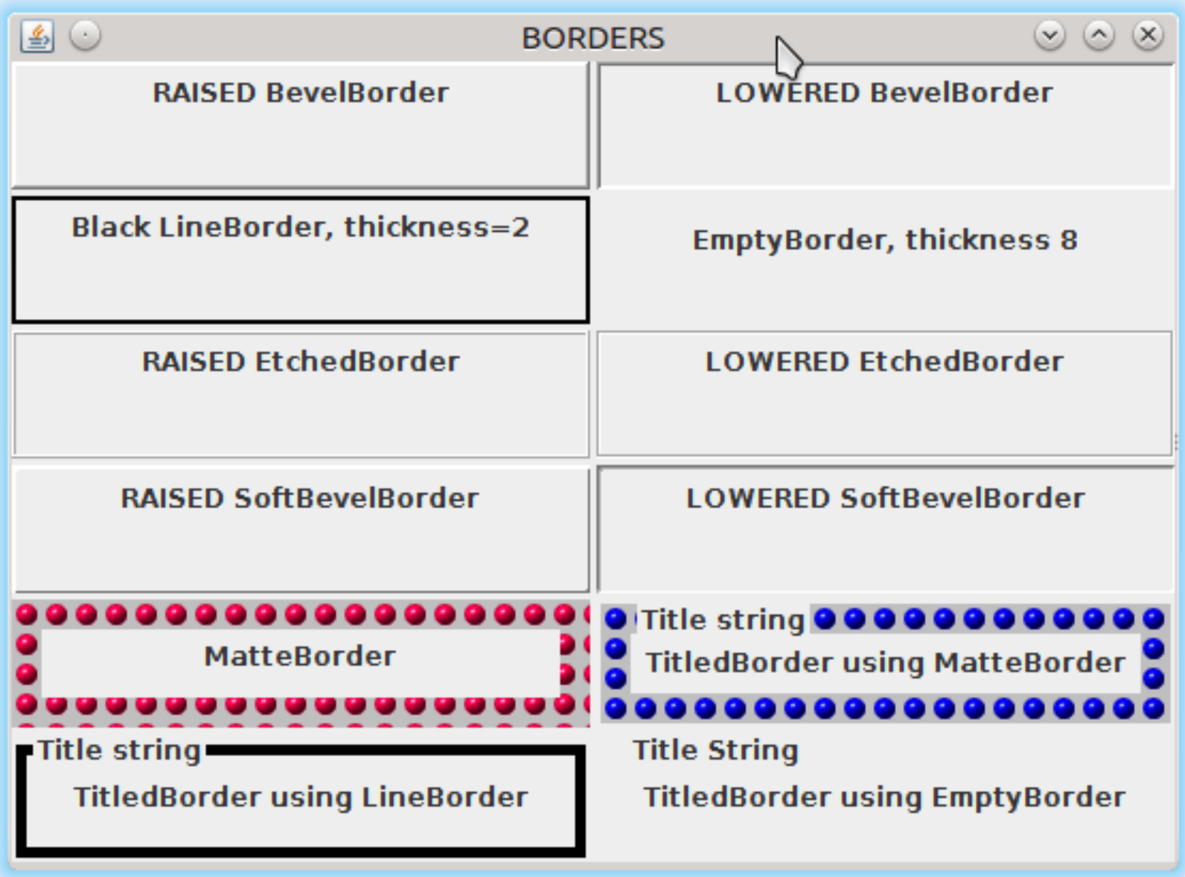
Listing 111 MCP-Borders/Borders.java

```java
import java.awt.Color;
import java.awt.GridLayout;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.border.BevelBorder;
```

```
import javax.swing.border.EmptyBorder;
import javax.swing.border.EtchedBorder;
import javax.swing.border.LineBorder;
import javax.swing.border.MatteBorder;
import javax.swing.border.SoftBevelBorder;
import javax.swing.border.TitledBorder;

public class Borders extends JFrame {
    public static void main(String args[]) {
        new Borders();
    }

    public Borders() {
        super("BORDERS");
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        JPanel content = new JPanel();
        content.setLayout(new GridLayout(6,2,3,3));

        JPanel p = new JPanel();
        p.setBorder(new BevelBorder(BevelBorder.RAISED));
        p.add(new JLabel("RAISED BevelBorder"));
        content.add(p);

        p = new JPanel();
        p.setBorder(new BevelBorder(BevelBorder.LOWERED));
        p.add(new JLabel("LOWERED BevelBorder"));
        content.add(p);

        p = new JPanel();
        p.setBorder(new LineBorder(Color.black, 2));
        p.add(new JLabel("Black LineBorder, thickness=2"));
        content.add(p);

        p = new JPanel();
        p.setBorder(new EmptyBorder(8, 8, 8, 8));
        p.add(new JLabel("EmptyBorder, thickness 8"));
        content.add(p);

        p = new JPanel();
        p.setBorder(new EtchedBorder(EtchedBorder.RAISED));
        p.add(new JLabel("RAISED EtchedBorder"));
        content.add(p);

        p = new JPanel();
        p.setBorder(new EtchedBorder(EtchedBorder.LOWERED));
        p.add(new JLabel("LOWERED EtchedBorder"));
        content.add(p);

        p = new JPanel();
```

```
        p.setBorder(new SoftBevelBorder(
                    SoftBevelBorder.RAISED));
        p.add(new JLabel("RAISED SoftBevelBorder"));
        content.add(p);

        p = new JPanel();
        p.setBorder(new SoftBevelBorder(
                    SoftBevelBorder.LOWERED));
        p.add(new JLabel("LOWERED SoftBevelBorder"));
        content.add(p);

        p = new JPanel();
        p.setBorder(new MatteBorder(
                    new ImageIcon("redball.gif")));
        p.add(new JLabel("MatteBorder"));
        content.add(p);

        p = new JPanel();
        p.setBorder(new TitledBorder(new MatteBorder(
                        new ImageIcon("blueball.gif")),
                            "Title string"));
        p.add(new JLabel("TitledBorder using MatteBorder"));
        content.add(p);

        p = new JPanel();
        p.setBorder(new TitledBorder(
                        new LineBorder(Color.black, 5),
                            "Title string"));
        p.add(new JLabel("TitledBorder using LineBorder"));
        content.add(p);

        p = new JPanel();
        p.setBorder(new TitledBorder(
                        new EmptyBorder(10, 10, 10, 10),
                            "Title String"));
        p.add(new JLabel("TitledBorder using EmptyBorder"));
        content.add(p);

        add(content);
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
}
```

The program displays



In the following example, we demonstrate, how one can set icons for buttons, different for different states of the button. Moreover, icons are custom made — we draw them manually, so we don't need any additional graphic files. Components JToggleButton are by themselves not very interesting, but the class is the base for much more useful JRadioButton and JCheckBox.

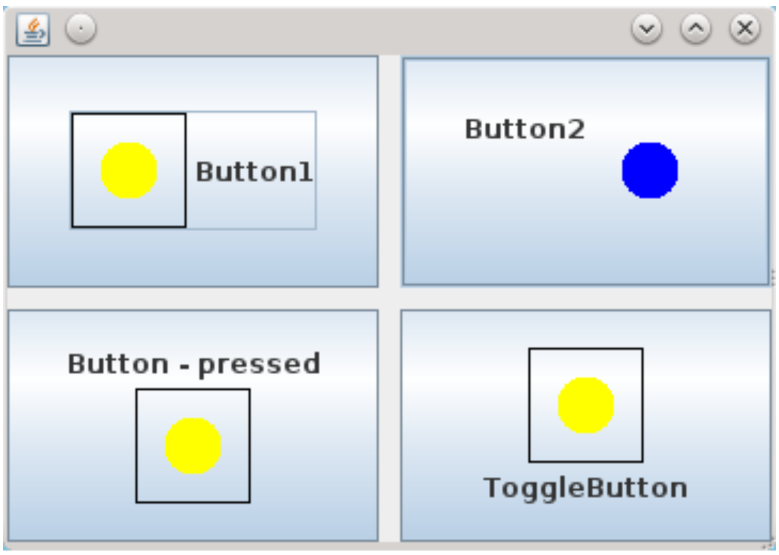
Listing 112 MCU-Buttons/Buttons.java

```java
import java.awt.Color;
import java.awt.Component;
import java.awt.Graphics;
import java.awt.GridLayout;
import javax.swing.AbstractButton;
import javax.swing.Icon;
import javax.swing.JButton;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JToggleButton;
import javax.swing.SwingUtilities;
import static javax.swing.SwingConstants.*;


public class Buttons extends JFrame {
```

```java
    private Icon[] icons = {
        new MyIcon(Color.YELLOW, true),
        new MyIcon(Color.BLUE, false),
        new MyIcon(Color.RED, true),
        new MyIcon(Color.BLACK, false)
    };

    // will be programatically pressed
    private JButton bpre = new JButton("Button - pressed");

    public static void main(String args[]) {
        new Buttons();
    }

    Buttons() {
        setLayout(new GridLayout(2, 2, 10, 10));
        JButton b = new JButton("Button1");
        setButt(b, icons, RIGHT, CENTER);
        JButton bmov = new JButton("Button2");
        setButt(bmov, icons, LEFT, TOP);
        setButt(bpre, icons, CENTER, TOP);

        JToggleButton tb = new JToggleButton(
                "ToggleButton");
        setButt(tb, icons, CENTER, BOTTOM);

        setDefaultCloseOperation(DISPOSE_ON_CLOSE);
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
        try {
            Thread.sleep(2500);
            SwingUtilities.invokeLater(() -> {
                bpre.doClick(500); // pressed for 500 ms
            });
        } catch(Exception ignore) { }
    }

    void setButt(AbstractButton b, Icon[] i,
            int horPos, int vertPos) {
        b.setFocusPainted(true);
        b.setIcon(i[0]);
        b.setRolloverIcon(i[1]);
        b.setPressedIcon(i[2]);
        b.setSelectedIcon(i[3]);
        b.setHorizontalTextPosition(horPos);
        b.setVerticalTextPosition(vertPos);
        add(b);
    }
}
```

```java

class MyIcon implements Icon {

    private Color color;
    private int w = 80;
    private boolean frame;

    MyIcon(Color c, boolean frame) {
        color = c;
        this.frame = frame;
    }

    @Override
    public void paintIcon(Component c,
            Graphics g, int x, int y) {
        Color old = g.getColor();
        g.setColor(color);
        w = ((JComponent) c).getHeight()/2;
        int p = w/4, d = w/2;
        g.fillOval(x + p, y + p, d, d);
        if (frame) {
            g.setColor(Color.BLACK);
            g.drawRect(x, y, w-1, w-1);
        }
        g.setColor(old);
    }

    @Override
    public int getIconWidth()  { return w; }
    @Override
    public int getIconHeight() { return w; }
}
```

The program displays



The last example demonstrates GridBagLayout applied to a calculator-like application, which displays the following interface

The code is presented below. Note that for each component added to the main **JPanel**, we create a separate object **GridBagConstraints**. This is not needed, we could reuse existing objects for several components. However, such approach can be rather error-prone, as we would have to remember which fields are set and reset them to other values before applying to another components.

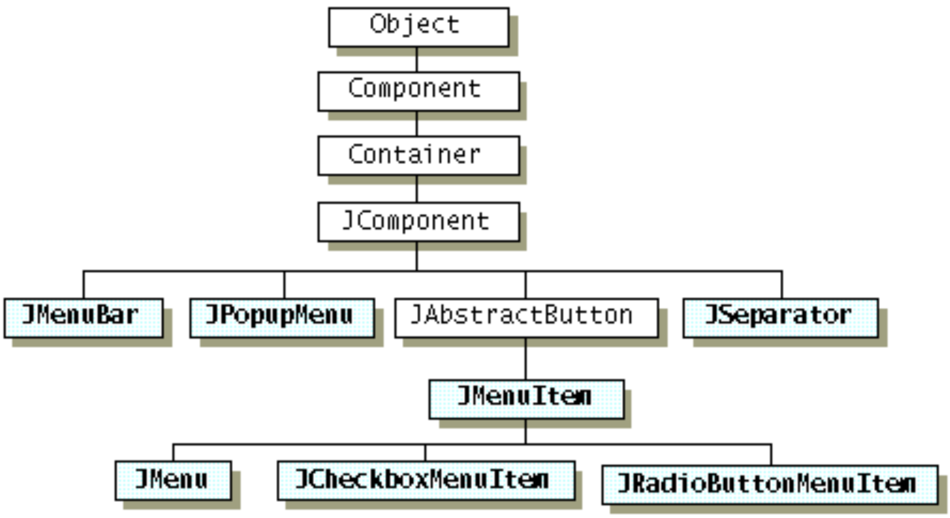
Listing 113 MDB-GridBag/GridBag.java

```java
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class GridBag {
    public static void main (String[] args) {
        JFrame f = new JFrame("GBL");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.add(new MyPanel());
        f.pack();
        f.setLocationRelativeTo(null);
        f.setVisible(true);
    }
}

class MyPanel extends JPanel {
    MyPanel() {
        setLayout(new GridBagLayout());
          // upper row
        String[] ur = {"%","\u00f7","\u00d7","\u2212"};
        for (int i = 0; i < ur.length; ++i) {
            GridBagConstraints c = new GridBagConstraints();
            c.fill = GridBagConstraints.BOTH;
            c.gridx = i;
            c.gridy = 0;
            c.weightx = 0.5; // important when resizing
            c.weighty = 0.5;
            add(new JButton(ur[i]),c);
        }
          // numeric pad
```

```
        for (int i = 1; i <= 9; ++i) {
            GridBagConstraints c = new GridBagConstraints();
            c.fill = GridBagConstraints.BOTH;
            c.gridx = (i-1)%3;
            c.gridy = 3 - (i-1)/3;
            c.weighty = 0.5;
            c.ipadx = 10; // digit buttons will be larger
            c.ipady = 10;
            add(new JButton(""+i),c);
        }
          // plus and == at rhs
        String[] rr = {"\u002b","\u003d"};
        for (int i = 0; i < 2; ++i) {
            GridBagConstraints c = new GridBagConstraints();
            c.fill = GridBagConstraints.BOTH;
            c.gridx = 3;
            c.gridy = 1 + 2*i;
            c.gridheight = 2;
            add(new JButton(rr[i]),c);
        }
          // zero
        GridBagConstraints czero = new GridBagConstraints();
        czero.fill = GridBagConstraints.BOTH;
        czero.gridx = 0;
        czero.gridy = 4;
        czero.gridwidth = 2;
        add(new JButton("0"),czero);
          // dot
        GridBagConstraints cdot = new GridBagConstraints();
        cdot.fill = GridBagConstraints.BOTH;
        cdot.gridx = 2;
        cdot.gridy = 4;
        cdot.weighty = 0.5;
        add(new JButton("\u2022"),cdot);
    }
}
```

12.10 Menus

Menus belong to the most useful elements of almost any graphical interface. In Java, one can easily create even quite complex menus. The classes involved are presented in the figure below



As we can see, the parent type here is **JMenuItem**, representing a selectable menu item. Selectable, because **JMenuItem** inherits from **JAbstractButton** and therefore can be clicked to fire an event that we can somehow handle — exactly as buttons. Also as buttons, menu items can be equipped with texts and icons that can be configured. In order to create a menu, one

- creates **JMenuBar**;
- creates menus (**JMenu**);
- to each menu, adds other menus which will constitute its submenus;
- to each menu or submenu, adds other menus or, finally, menu items — **JMenuItem** — which therefore are leaves of the tree-like structures corresponding to each highest level menu;
- adds the highest level menus to **JMenuBar**;
- sets this menu bar as the menu bar of a window (usually **JFrame**).

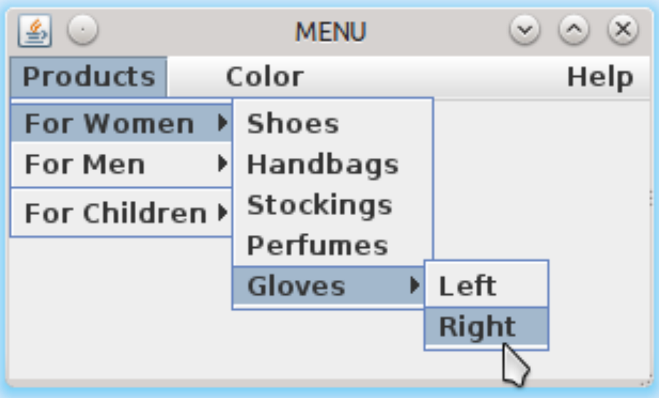
Let us look at an example:

Listing 114 MDA-MenuSimple/MenuSimple.java

```java
import java.awt.Dimension;
import javax.swing.Box;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;

public class MenuSimple {
    public static void main (String[] args) {
        JFrame f = new JFrame("MENU");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel panel = new JPanel();
        panel.setPreferredSize(new Dimension(300,130));

          // menu Products
        JMenu productMenu = new JMenu("Products");
          // submenu For Women
```

```java
        JMenu women = new JMenu("For Women");
          // adding items to the submenu For Women
        women.add(new JMenuItem("Shoes"));
        women.add(new JMenuItem("Handbags"));
        women.add(new JMenuItem("Stockings"));
        women.add(new JMenuItem("Perfumes"));
        JMenu gloves = new JMenu("Gloves");
        gloves.add(new JMenuItem("Left"));
        gloves.add(new JMenuItem("Right"));
        women.add(gloves);
          // submenu For  Men
        JMenu men = new JMenu("For Men");
          // adding items to the submenu For Men
        men.add(new JMenuItem("Beer"));
          // submenu For Children (will be empty)
        JMenu children = new JMenu("For Children");
          // adding submenus to menu Products
        productMenu.add(women);
        productMenu.add(men);
        productMenu.addSeparator();
        productMenu.add(children);

          // menu Color
        JMenu colorMenu = new JMenu("Color");
          // adding menu items
        colorMenu.add(new JMenuItem("Red"));
        colorMenu.add(new JMenuItem("Blue"));

          // menu Help - here wil be empty...
        JMenu helpMenu  = new JMenu("Help");

          // adding menus to menu bar
        JMenuBar menuBar = new JMenuBar();
        menuBar.add(productMenu);
        menuBar.add(Box.createHorizontalStrut(20));
        menuBar.add(colorMenu);
        menuBar.add(Box.createGlue());
        menuBar.add(helpMenu);

          // setting menu bar
        f.setJMenuBar(menuBar);

        f.add(panel);
        f.pack();
        f.setLocationRelativeTo(null);
        f.setVisible(true);
    }
}
```

The program displays



Notice that

- By default, the view of the JMenuBar is laid out by Box layout. Therefore, one can add 'glues' and horizontal struts (rigid gaps) between menus: in the example above, there is a strut between 'Products' and 'Color' menus and a glue between 'Color' and 'Help' menus (and therefore 'Help' is shifted to the right).
- In the submenu lists, one can also add separators by invoking **addSeparator**; in the example there is such a separator between 'For Men' and 'For Children' submenus.

More examples are provided in section **??** (p. **??**).

12.11 Dialogs

It is quite common that our application needs to read some data from the user, or display some kind of a message. This can be done by displaying a special type of a window — JDialog, which is somewhat similar to JFrame but with limited functionality. Objects of this class have an owner (parent window) that one can specify explicitly. The most convenient way of creating and using dialogs is by invoking one of the many static methods of class JOptionPane. These are:

- showConfirmDialog — asks the for confirmation (e.g., something like *'Do you really want to quit?'*); the possible answers are then 'yes', 'no', or 'cancel'.
- showInputDialog — prompts the user for some input data; returns a String typed by the user, or in some cases a user-selected Object.
- showMessageDialog — displays some kind of a message.
- showOptionDialog — combines functionality of the above three.

All these methods come in many overloaded flavors with different number of parameters. The meaning of these parameters is as follows:

- parentComponent — specifies the parent of the dialog (so it will appear on or just below its parent. Setting it to null will result in the dialog appearing in the center of the screen.

- message — a message to be placed in the dialog box (usually some kind of a prompt explaining the user what is expected). It does not to be a **String** — its interpretation depends on its type:
  - **Object[]** — an array of objects is interpreted as a series of messages arranged vertically.
  - **Component** — is displayed 'as is'.
  - **Icon** — the icon is displayed wrapped in a label.
  - for arguments of others types, **toString** is invoked and the returned string is displayed.
- messageType — defines the type of the message. Predefined types are specified by one of the integer constants from class **JOptionPane**: ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE and PLAIN_MESSAGE. It will be taken into account for selecting, for example, an appropriate icon (if not set explicitly in the program).
- optionType — for dialogs displaying a set of buttons, specifies, if not set explicitly by the program, what buttons will appear. This is also an integer constants defined in class **JOptionPane**: DEFAULT_OPTION, YES_NO_OPTION, YES_NO_CANCEL_OPTION and OK_CANCEL_OPTION. However, by specifying explicitly options (see below), one can provide any set of buttons.
- options — specifies what buttons will appear in the dialog box: normally, this is an array of **String**s, but can also be an array of any **Object**s. Then buttons will be created depending on the type of elements:
  - **Component** — the component will be used directly instead of a button;
  - **Icon** — the icon will be used instead of a button;
  - for other types, **toString** will be invoked and the returned string will be used on the button.
- icon — specifies a decorative icon to be placed in the dialog box. The default (if there is no corresponding argument or it is **null**) will be determined by the messageType parameter (see above).
- title — a title to appear on the title bar of the dialog.
- initialValue — the default selection (input value) if there are several options.

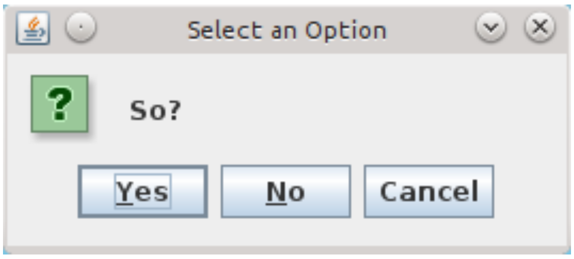
Some static methods of **JOptionPane** return an **int**; it is equal to one of the predefined constants from class **JOptionPane**: YES_OPTION, NO_OPTION, CANCEL_OPTION, OK_OPTION and CLOSED_OPTION and corresponds to a button that was clicked by the user (or, if the user just closed the dialog without pressing any button, CLOSED_OPTION is returned).

For example, when we want the user to confirm some decision, we can use a **showConfirmDialog** function. The following snippet

```
int a = JOptionPane.showConfirmDialog(
            null, // parent
            "So?" // message
        );
if (a == JOptionPane.YES_OPTION)
    System.out.println("Yes!");
else if (a == JOptionPane.NO_OPTION)
    System.out.println("No!");
```

```
    else if (a == JOptionPane.CANCEL_OPTION)
        System.out.println("Canceled!");
    else if (a == JOptionPane.CLOSED_OPTION)
        System.out.println("Closed!");
    else
        System.out.println("Not possible...!");
```

will display a dialog



and return an **int** which may then be examined, as shown in the example. Actually, there are four overloaded versions of **showConfirmDialog** (see documentation).

The **showConfirmDialog** function can give us only a simple yes/no answer. To request some data, **showInputDialog** may be used instead. It has six overloaded versions, five of which return a **String**. For example

```
    String b = JOptionPane.showInputDialog(
                   null, // parent
                   "Enter an int...", // message
                   "42" // initial value
               );
    int i = 0;
    if (b == null || b.trim().equals("")) {
        System.out.println("No value returned");
        i = -1;
    } else {
        try {
            i = Integer.parseInt(b);
        } catch(NumberFormatException e) {
            System.out.println("This is not an int!");
            i = -2;
        }
    }
    System.out.println("i = " + i);
```

returns a **String** which can be then converted to an **int**. When such a dialog is displayed, the default value is already shown in the text field, so the user can just press 'enter' to select it.

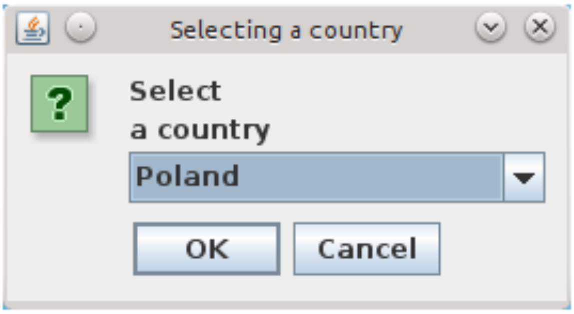
There is one version of **showInputDialog** which returns an **Object**, not necessarily a **String**. Let us consider an example. Here we pass an array of references to object of type **CountryLabel** (which extends **JLabel**, but it could be any type). Options represented by the elements of the array, converted to strings by means of **toString** method, are shown in a combo box for the user to select from. After a selection has been made, the corresponding element of the array is returned as an **Object**, but can be safely cast to its real type:

Listing 115 MCW-Options/CountryDialog.java

```java
import java.awt.Dimension;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import static javax.swing.JFrame.EXIT_ON_CLOSE;

public class CountryDialog {
    public static void main(String[] args) {
        JLabel[] opts = {
            new CountryLabel("Poland","Warsaw"),
            new CountryLabel("France","Paris"),
            new CountryLabel("UK","London")};
          // This version returns Object, not String!!!
          // Options will be displayed in a combo box.
        Object c = // will be of type CountryLabel
            JOptionPane.showInputDialog(
                null, // parent component
                "Select\na country",    // message
                "Selecting a country", // title
                JOptionPane.QUESTION_MESSAGE,// message type
                null,   // icon
                opts,   // options (objects of any type)
                opts[0] // default selection
            );
        CountryLabel cl = (CountryLabel)c;
        if (cl == null) System.out.println("Cancelled");
        else {
            JFrame f = new JFrame(cl.toString());
            f.setDefaultCloseOperation(EXIT_ON_CLOSE);
            f.add(cl);
            f.setSize(new Dimension(200,150));
            f.setLocationRelativeTo(null);
            f.setVisible(true);
        }
    }
}

class CountryLabel extends JLabel {
    String name;
    String capital;
    CountryLabel(String n, String c) {
        super(n + ", capital: " + c,
              new ImageIcon(n + ".gif"), JLabel.CENTER);
        setHorizontalTextPosition(JLabel.CENTER);
        setVerticalTextPosition(JLabel.BOTTOM);
        name=n;
        capital=c;
```

```
    }
    String getCountryName()  { return name; }
    String getCapital()      { return capital; }
    @Override
    public String toString() { return name; }
}
```

The program displays first a dialog with options in the form of a combo box, as shown on the left hand side of the figure below



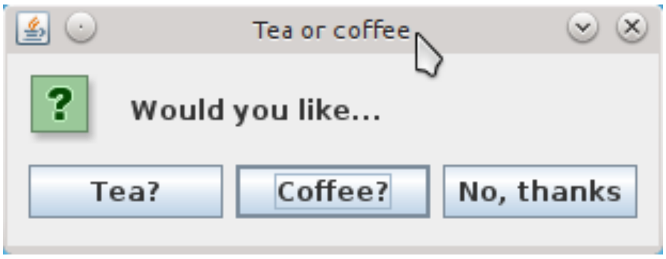
Note that pressing the space bar unfolds the options of the combo box; you can use ‘up’ and ‘down’ keys to move around the options, press ‘enter’ to select one and again ‘enter’ to finish the selection process. To jump directly to an option, you can just press the key corresponding to the first letter of an option. Therefore, you can do everything without even touching the mouse (power users abhor mice). In the example, after selecting ’UK’, you should see a little window shown on the right hand side of the figure.

Using **showOptionDialog**, one can customize texts on the buttons with options:

```
Object[] opts = {"Tea?", "Coffee?", "No, thanks"};
int a = JOptionPane.showOptionDialog(
            null, // parent
            "Would you like...", // message
            "Tea or coffee",     // title
                // option type
            JOptionPane.YES_NO_CANCEL_OPTION,
                // message type
            JOptionPane.QUESTION_MESSAGE,
            null, // icon
            opts, // options
            opts[1] // initial value
        );
if (a == JOptionPane.YES_OPTION)
    System.out.println("Tea");
else if (a == JOptionPane.NO_OPTION)
    System.out.println("Coffee");
else if (a == JOptionPane.CANCEL_OPTION)
    System.out.println("Nothing");
else if (a == JOptionPane.CLOSED_OPTION)
    System.out.println("Closed!");
```

```
    else
        System.out.println("Not possible...!");
```

For example, if option type is set to YES_NO_CANCEL_OPTION, instead of 'yes', 'no', 'cancel', we will see our texts passed as an array:



However, the returned value still will be one of YES_OPTION, NO_OPTION, CANCEL_OPTION or CLOSED_OPTION.

Another form of a dialog is presented below. Here we display a message dialog, but the function **showMessageDialog** doesn't return anything. However, the 'message' contains a group of radio buttons, and we can find which of them has been selected (after returning from the function) by querying this group of buttons:

Listing 116 MDE-RadioBut/RadioBut.java

```java
import java.util.Enumeration;
import javax.swing.AbstractButton;
import javax.swing.ButtonGroup;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JLabel;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JRadioButton;

public class RadioBut {
    public static void main(String[] args) {
        Object[] mess =
            new Object[3+Stars.values().length];
        mess[0] = "Select one";
        mess[1] = "(and only one)";
        mess[2] = " ";
        ButtonGroup bgroup = new ButtonGroup();
        int i = 0;
        for (Stars s : Stars.values()) {
            JRadioButton b = new JRadioButton(s.getFirst());
            b.putClientProperty("star",s);
            mess[3+i] = b;
            bgroup.add(b);
            ++i;
        }

        JOptionPane.showMessageDialog(
```

```
                 null,mess, // <-- array of Objects
                 "Hard choice...",
                 JOptionPane.QUESTION_MESSAGE, // ignored
                 new ImageIcon("stars.png"));

         Stars star = null;
         Enumeration<AbstractButton> buttons =
                                     bgroup.getElements();
         while (buttons.hasMoreElements() && star == null) {
             AbstractButton b = buttons.nextElement();
             if (b.isSelected()) star =
                     (Stars)b.getClientProperty("star");
         }
         if (star != null) {
             JOptionPane.showMessageDialog(null,
                 "You have selected " + star.getFirst() +
                 " (" + star + ")", "Your selection",
                 JOptionPane.INFORMATION_MESSAGE);
             display(star.getIcon());
         } else {
             JOptionPane.showMessageDialog(null,
                 "No star selected!","No selection",
                 JOptionPane.ERROR_MESSAGE);
         }
     }

     private static void display(Icon icon) {
         JFrame f = new JFrame("Your star");
         f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         f.add(new JLabel(icon));
         f.pack();
         f.setLocationRelativeTo(null);
         f.setVisible(true);
     }
 }

 enum Stars {
     MONICA("monica") {
         @Override
         public String toString() {return "Monica Bellucci";}
     },
     PENELOPE("penelope") {
         @Override
         public String toString() {return "Penelope Cruz";}
     },
     CINDY("cindy") {
         @Override
         public String toString() {return "Cindy Crawford";}
     };
     Icon icon;
```

```
    Stars(String fname) {
        icon = new ImageIcon(fname + ".jpg");
    }
    Icon getIcon() { return icon; }
    String getFirst() {
        return this.toString().split(" ")[0];
    }
}
```

(the example also illustrates non-trivial use of enums, see sect. 6 on p. 76).

A few more examples of dialogs will be presented in sect. **??** on p. **??**; in particular **JFileChooser** (see **??**, p. **??**) and **JColorChooser** (see **??**, p. **??**).

List of listings

Index