8 Miscellaneous

- ‘for each’ Loops
- File I/O
- Enumeration types
- Methods with parameter lists of variable length
- Java Archives
- Regular Expressions

‘**for each**’ loops simplify processing of individual elements of arrays and collections:

Simple example with arrays:

```
1  public class K8B01E_ForEachArray {
2
3   public static void main (String[] args) {
4
5    int day = 0;
6
7    System.out.println("Days of the week:");
8
9    String [] week = { "Monday", "Tuesday", "Wednesday",
10                      "Thursday", "Friday", "Saturday", "Sunday"};
11
12   for (String str : week) //for each String str in week do ...
13     System.out.println( ++day +"th day of the week  = " + str);
14
15  }
16 }
```

‘for each’ loops with multidimensional arrays:

```
1  class K8B02E_ForEachMatrix {
2    public static void main(String[] args) {
3      int [] [] array = { {22, 45, 57, 33},
4                          {64, 28, 19},
5                          {},
6                          {97},
7                          {88, 73, 44, 35, 84} };
8      System.out.println("array (row-wise:");
9      for ( int [] row: array ) {
10       for ( int element : row )
11          System.out.print( element + "␣␣");
12       System.out.println();
13     }
14   }
15 }
```

‘for each’ loop with collections:

```
1  import java.util.*;
2
3  public class K8B03E_ForEachCollection {
4    public static <E> void printCollection
5                (Collection <E> c, String s) {
6      System.out.println(s + ":");
7      for (E e : c) System.out.println(e);
8      System.out.println("-----");
9    }
10
11   public static void main(String[] args) {
12     LinkedList <Integer> list = new LinkedList<>();
13     list.add(1);
14     list.add(22);
15     list.add(333);
16
17     printCollection(list, "data");
18   }
19 }
```

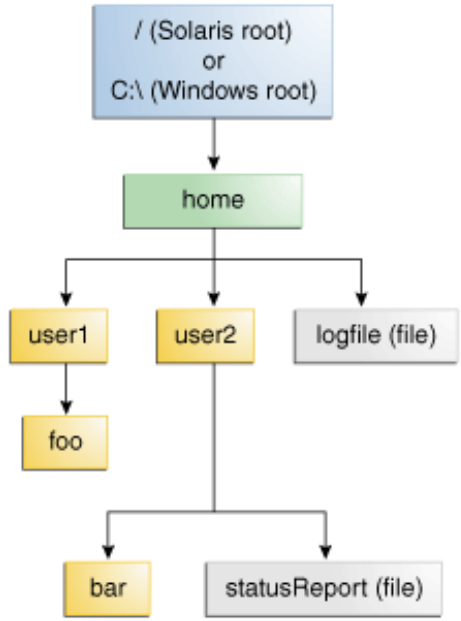
‘for each’ can only be applied for reading:

```
 1 public class K8B04E_ForEachRW {
 2
 3    public static void main (String args[]) {
 4
 5       String output = "";
 6       int [] iArray = new int [5];
 7       int count = 0;
 8
 9       for (int i = 0; i < 5 ; i ++)
10          iArray [i] = 2*i;  // modification
11
12       for (int i : iArray)
13          System.out.println(i); // read-only access
14
15    }
16 }
```

8 Miscellaneous

- ‘for each’ Loops
- File I/O
- Enumeration types
- Methods with parameter lists of variable length
- Java Archives
- Regular Expressions

Tree structure of the file system



source: `https://docs.oracle.com/javase/tutorial/essential/io/path.html`

A **file system** consists of at least one **root**, a hierarchical tree structure of **directories** and **files** (as leaf nodes without further children).

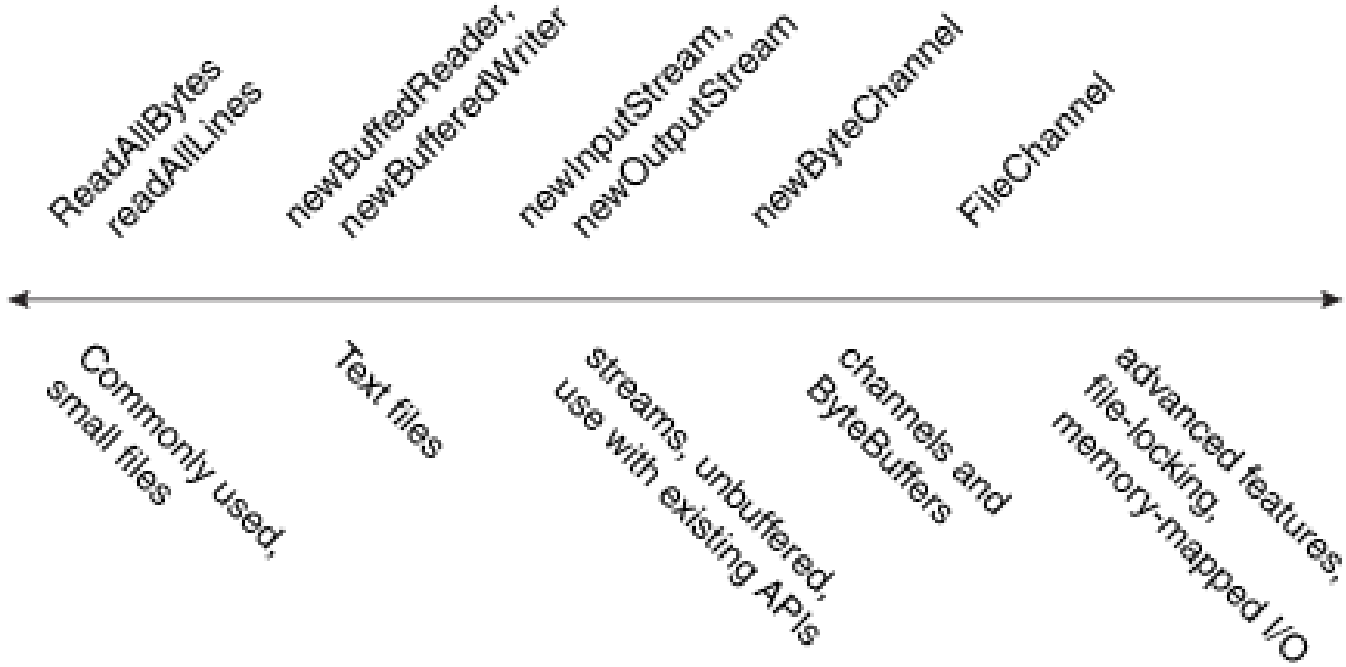
Files and directories are identified using their **path name**, e.g. `/home/sally/statusReport` (Solaris, Linux) or `C:\home\sally\statusReport` (Windows) – separator character is system-specific in Java!

For the file system, files are sequences of bytes without any further semantics.

In Java, files are abstracted as a **stream** of bytes that are read or written. Additional methods allow to write more complex objects, especially Strings.

“*Prior to Java SE 7, File IO involved a fair amount of code, some of it nonintuitive and difficult to remember even for Java veterans. For instance, I would often find myself searching my older code and cutting/pasting into new projects.*”

(Eric Bruno, www.drdobbs.com/jvm/java-se-7-new-file-io/231600403)



https://docs.oracle.com/javase/tutorial/essential/io/file.html

In the following: recipe for **`Java SE 7 – New File IO`**

Reading files

- Open an existing file using a path

```
Path path = FileSystems.getDefault().getPath(".", name);
```

- Small files can be read completely:

```
byte[] filearray = Files.readAllBytes(path);
```

- Alternative: read them row-wise, e.g. into **`List<String>`**:

```
List<String> lines = Files.readAllLines(path,
        Charset.defaultCharset() );
```

- For big files it is better to read them in pieces:

```
BufferedReader nbr =
  Files.newBufferedReader(path, Charset.defaultCharset() );
...
String line = null;
while ( (line = nbr.readLine()) != null ) { /* ... */ }
nbr.close(); // but: see below!
```

the file should be closed, see below!

Writing files

- Creating and writing a new file based on existing data:

```
String content = ...
Files.write( path, content.getBytes(),
                      StandardOpenOption.CREATE);
// creates new file.
```

- If the file should be created from small pieces, one can use the following approach:

```
BufferedWriter nbw =
    Files.newBufferedWriter( path, Charset.defaultCharset(),
         StandardOpenOption.CREATE);
...
while ( ... ) {
  String content = ...
  nbw.write(content, 0, content.length());
}
...
nbw.close(); // but: see below!
```

The file must be closed! Alternative to **`close`**: see below
There are many more possible values for `StandardOpenOption` like `APPEND`,`TRUNCATE_EXISTING` (see documentation!)

Excursion: `try-with-resources`

Alternative to **`try-catch-finally`** with slightly handling of exceptions in **`try`** and in **`finally/close`**

```
try ( BufferedReader nbr = ...)
{ ... }
catch(IOException e)
{...}
```

Here, **`nbr.close()`** is called automatically at the end of the **`try`** block, even if an exception is thrown before.

Example program **`K8B05E_FileIO.java`**:

- Runneable program that uses the methods above
- including **`try-catch`** where useful
- **`try-with-resources`** makes sure that the used file is closed at the end of the block.

Used imports:

```
1  import java.io.BufferedReader;
2  import java.io.BufferedWriter;
3  import java.io.IOException;
4  import java.nio.charset.Charset;
5  import java.nio.file.FileSystems;
6  import java.nio.file.Files;
7  import java.nio.file.Path;
8  import java.nio.file.StandardOpenOption;
9  import java.util.ArrayList;
10 import java.util.List;
```

Methods for reading small files:

```
1   public static byte[] readSmallFileBytes(String name) {
2     try {
3       Path path = FileSystems.getDefault().getPath(".", name);
4       return Files.readAllBytes(path);
5     }
6     catch ( IOException ioe ) { ioe.printStackTrace(); }
7     return null;
8   }
9
10  public static List<String> readSmallFileLines(String name) {
11    try {
12      return Files.readAllLines(
13            FileSystems.getDefault().getPath(".", name),
14            Charset.defaultCharset() );
15    }
16    catch ( IOException ioe ) { ioe.printStackTrace(); }
17    return null;
18  }
```

Reading a large file (here into a **`List<String>`**, close via **`try-with-resources`**):

```
1   public static List<String> readLargeFileLines(String name) {
2     try ( BufferedReader nbr =
3             Files.newBufferedReader(
4               FileSystems.getDefault().getPath(".", name),
5               Charset.defaultCharset() )
6     ){
7       List<String> lines = new ArrayList<>();
8       while (true){
9         String line = nbr.readLine();
10        if ( line == null ) return lines;
11        lines.add(line);
12      }
13
14    }
15    catch ( IOException ioe ) { ioe.printStackTrace(); }
16    return null;
17  }
```

Writing small files in one shot:

```
1 public static void writeFileBytes(String name, String content){
2  try {
3    Files.write(
4       FileSystems.getDefault().getPath(".", name),
5       content.getBytes(),
6       StandardOpenOption.CREATE);
7   }
8   catch ( IOException ioe ) { ioe.printStackTrace(); }
9 }
```

Writing large files piecewise (here from a **`List<String>`**, again closed using **`try-with-resources`**):

```
1   public static void writeLargeFileLines(String name,
2                       List<String> lines) {
3     try ( BufferedWriter nbw =
4             Files.newBufferedWriter(
5               FileSystems.getDefault().getPath(".", name),
6               Charset.forName("UTF-8"),
7               StandardOpenOption.CREATE)
8     ){
9       for ( String line : lines ) {
10          nbw.write(line, 0, line.length());
11          nbw.newLine();
12      }
13    }
14    catch ( IOException ioe ) { ioe.printStackTrace(); }
15  }
```

Corresponding **`main`** method:

```
1  public static void main(String[] args) {
2    String outString = "line 1
3  nline 2
4  nline 3";
5    List<String> lines = null;
6  // write bytes in small file
7    writeFileBytes("K8B05a.txt", outString);
8  // read bytes from small file
9    System.out.println("\n-- TEST 1 -------------------------");
10   String inString = new String(readSmallFileBytes("K8B05a.txt"));
11   System.out.println(inString);
12 // read lines from small file
13   System.out.println("\n-- TEST 2 -------------------------");
14   lines = readSmallFileLines("K8B05a.txt");
15   for ( String line: lines ) System.out.println(line);
16 // read lines from large file
17   System.out.println("\n-- TEST 3 -------------------------");
18   lines = readLargeFileLines("K8B05a.txt");
19   for ( String line: lines )System.out.println(line);
20 // write lines in large file with buffer
21   writeLargeFileLines("K8B05b.txt", lines);
22 }
```

8 Miscellaneous

- ‘for each’ Loops
- File I/O
- Enumeration types
- Methods with parameter lists of variable length
- Java Archives
- Regular Expressions

Enumerations (enum) In the simple case, **`enum`** defines a type as a set of constants. Every constant is represented only by its name.

Type definition:

```
enum Weekdays { MONDAY, TUESDAY, WEDNESDAY,
                THURSDAY, FRIDAY, SATURDAY, SUNDAY };
```

Variable definition:

```
Weekdays aDay;
```

Assignment:

```
aDay = Weekdays.TUESDAY;
```

In fact, **enum** defines a class that can also have constructors, variables and methods, in addition to constants.

- **enum** types are implicitly final
- **enum** constants are implicitly static

```
1  public class K8B06_EnumTage {
2    private enum Weekdays { MONDAY, TUESDAY, WEDNESDAY,
3                            THURSDAY, FRIDAY, SATURDAY, SUNDAY };
4
5    public static void main( String args[] ) {
6
7      Weekdays aDay = Weekdays.TUESDAY;
8      System.out.println("single day: " + aDay);
9
10     //values() returns array with all constants of this enum type
11     System.out.println("\nlist of all weekdays:");
12     for ( Weekdays day : Weekdays.values() )
13          System.out.println(day);
14   }
15 }
```

enum types can be translated like normal classes, or as nested static classes:

```
1  import java.util.EnumSet;
2
3  public class K8B07E_EnumBook {
4    public static enum Book {
5      JHTP8( "Java␣How␣to␣Program␣8e", "2015" ),
6      CHTP8( "C␣How␣to␣Program␣8e", "2016" ),
7      CPPHTP9( "C++␣How␣to␣Program␣9e", "2016" ),
8      CSHARPHTP5( "C#␣How␣to␣Program␣5e", "2012" );
9
10     //(constants are objects of their own class!)
11
12     private final String title;
13     private final String copyrightYear;
14
15     Book( String bookTitle, String year ) {
16        title = bookTitle;
17        copyrightYear = year;
18     }
19
20     public String getTitle() { return title; }
21     public String getCopyrightYear() { return copyrightYear; }
22   }
```

```
1   public static void main( String args[] ) {
2     System.out.println("All␣books:");
3
4     for ( Book book : Book.values() )
5       System.out.println(book + ",␣" +
6          book.getTitle() + ",␣" + book.getCopyrightYear());
7
8     System.out.println("range selection:");
9
10    for ( Book book : EnumSet.range(Book.JHTP8,Book.CPPHTP9 ) )
11        System.out.println(book  + ",␣"+
12                           book.getTitle()  + ",␣"+
13                           book.getCopyrightYear());
14  }
15 }
```

8 Miscellaneous

- ‘for each’ Loops
- File I/O
- Enumeration types
- Methods with parameter lists of variable length
- Java Archives
- Regular Expressions

Methods with parameter lists of variable length:

- variable number of parameters of the same type
- implemented by combination of same-type parameters into an array

```
1  import java.util.Arrays;
2  class K8B08E_VarArg {
3
4    public static void sum (int ... numbers) {
5      int total = 0;
6      for (int i : numbers) total += i;
7      System.out.println("Sum of " +
8             Arrays.toString(numbers) +
9             ":␣" + total);
10     return;
11   }
12
13   public static void main (String args[]) {
14     int i1 = 5, i2 = 10, i3 = 15, i4 = 20;
15     sum (i1, i2);
16     sum (i1, i2, i3);
17     sum (i1, i2, i3, i4);
18   }
19 }
```

8 Miscellaneous

- ‘for each’ Loops
- File I/O
- Enumeration types
- Methods with parameter lists of variable length
- Java Archives
- Regular Expressions

- In general one uses **`.jar`** files (Java archives) instead of single classes; usually these archives are signed for security reasons.
- Example `Test` for creating a **`.jar`** file (requires Linux shell):
  - Three java files: `A.java`, `B.java`, `C.java`
  - Each with `main` method; `A.main()` should be started
  - `A.class` uses `B.class` and `C.class`

```
1 javac A.java
2
3 echo "Main-Class: A" > Test.manifest
4 jar cmvf Test.manifest Test.jar A.class B.class C.class
5
6 java -jar Test.jar
```

8 Miscellaneous

- ‘for each’ Loops
- File I/O
- Enumeration types
- Methods with parameter lists of variable length
- Java Archives
- Regular Expressions

Example: consider regular expression for syntactically correct US ZIP Code

$$\{\boldsymbol{a}, \ldots, \boldsymbol{z}, \boldsymbol{A}, \ldots, \boldsymbol{Z}\}^{\mathbf{2}} \circ (\varepsilon \cup \{\text{" "}\} \cup \{-\}) \circ \{\mathbf{1}, \ldots, \mathbf{9}\} \circ \{\mathbf{0}, \ldots, \mathbf{9}\}^{\mathbf{4}}$$

in `Java` syntax: `[a-zA-Z]{2}[ [-]]{0,1}[1-9][0-9]{4}`

- at the beginning: exactly 2 letters

  `[a-zA-Z]{2}`

  (2 lowercase or uppercase letters)
- at the end: exactly 5 digits, the first digit not 0

  `[1-9][0-9]{4}`

  (1 digit not 0, 4 arbitrary digits)
- in-between: nothing, 1 space, or 1 hyphen

  `[ [-]]{0,1}`

  (‘ ’ or ‘-’, 0 or 1 times)

`public boolean matches (String regex)` tests if the String matches the template defined by the regular expression **`regex`**.

```
1  public class K9B09E_ZipCode {
2
3    public static void main(String[] args) {
4
5      String regex = "[a-zA-Z]{2}[␣[-]]{0,1}[1-9][0-9]{4}";
6
7      boolean ok = args[0].matches(regex);
8
9      System.out.println("Input is " +
10                        (ok?"valid":"invalid"));
11     System.exit(ok?0:1);
12   }
13 }
```

The exit code can be processed in a ‘shell’,
e.g. in a Unix bash:

`java K9B09E_ZipCode Ab12345 && echo so it continues...`

`Java` expressions for regular expressions

`[abc]`	**1** character (`a`, `b` or `c`)
`[^abc]`	**1** character (not `a`, not `b` and not `c`)
`[a-dk-o]`	**1** character (in '`a` to `d`' or in '`k` to `o`')
`.`	exactly **1** arbitrary character

`+`	**1** or more characters
`*`	**0** or more characters
`?`	**0** or **1** characters

`{n}`	exactly ***n*** characters
`{n,}`	at least ***n*** characters
`{n,m}`	between ***n*** and ***m*** characters

`[abc[ghi]]`	union (corresponds to `[abcghi]`)
`[a-z&&[def]]`	intersection (corresponds to `[def]`)
`[a-z&&[^def]]`	difference (corresponds to `[a-cg-z]`

`[0-9]+`	at least **1** digit
`[0-9]*`	arbitrary many digits
`.?`	**0** or **1** arbitrary characters

`[a-z]{n}`	***n*** lowercase letters
`a{n,}`	at least ***n*** times '`a`'
`(ab){n,m}`	between ***n*** and ***m*** times '`ab`'

Predefined character classes:

- `\d` — a digit: `[0-9]`
- `\D` — a ‘non-digit’ character: `[^0-9]`
- `\s` — a ‘whitespace’ character: `[ \t\n\x0B\f\r]`
- `\S` — a ‘non-whitespace’ character: `[^\s]`
- `\w` — a word character: `[a-zA-Z_0-9]`
- `\W` — a ‘non-word’ character: `[^\w]`

Usage of escape sequences for example with
`String REGEX = "\\d";` for a single digit

a small selection of ‘boundary matchers’:

- `^` — beginning of a line (for `\n` in a String)
- `$` — end of a line) `\n` in a String)
- `\b` — word boundary
- `\A` — beginning of the input
- `\Z` — end of the input (ignoring the final terminator `\n`, `\r`)
- `\z` — end of input

Examples for regular expressions:

regex	String	match?
`foo`	`foo`	+
`cat.`	`cats`	+
`[rcb]at`	`cat`	+
`[rcb]at`	`hat`	-
`[^bcr]at`	`cat`	-
`[^bcr]at`	`hat`	+
`[a-c]`	`b`	+
`[a-c]`	`d`	-
`foo[1-5]`	`foo5`	+
`foo[^1-5]`	`foo6`	+
`[0-4[6-8]]`	`6`	+
`[0-4[6-8]]`	`5`	-
`[0-9&&[345]]`	`3`	+
`[2-8&&[4-6]]`	`3`	-
`[0-9&&[^345]]`	`6`	+

regex	String	match?
.	@	+
a?		+
a?	a	+
a?	aa	-
a+	aaa	+
a+		-
a*		+
a*	aaaaaa	+
a{3}	aaa	+
a{3}	aaaa	-
a{3,}	aaaa	+
(hi){2}	hihi	+
.*hi	hellohi	+
\d+	12345	+
\D+	12345	-
\w+	hi_ho	+
\s		+

We can also determine parts of a String where a given regular expression matches:

```
1  public class RegexpTest {
2
3    public static void main(String[] args) {
4
5      String regex = "[a-zA-Z]{2}";
6
7      Matcher m =Pattern.compile(regex).matcher(args[0]);
8
9                  while (m.find())
10                 {
11                    System.out.println("match: "+m.group());
12                 }
13   }
14 }
```

regex	String	match found?
ing	singer	+
ing\b	singer	-
ing\b	sing	+
ing\B	singer	+
ing\B	sing	-
\s	␣␣test	+
\s	test	-
\s	test␣␣	+
^\s	␣␣test	+
^\s	test	-
^\s	test␣␣	-
\s$	␣␣test	-
\s$	test	-
\s$	test␣␣	+