

September 2020,
Programming in Java

Core Java Classes & Exception Handling

Trayan Iliev

tiliev@iproduct.org

<http://iproduct.org>

Copyright © 2003-2020 IPT - Intellectual
Products & Technologies

Agenda for This Session

1. **Exception handling (2h)** – the role of exceptions; exception types; try-catch; try with resources; throwing exceptions; handling multiple exceptions; using stack-trace; finally block and its applications.
2. **Core Java classes (8h)**
 - **Strings** - creating and manipulating strings, concatenation, comparing, immutability, string pool; methods: `length()`, `charAt()`, `indexOf()`, `substring()`, `toLowerCase()/UpperCase()`, `equals()` and `equalsIgnoreCase()`, `startsWith()` and `endsWith()`, `contains()`, `replace()`, `trim()`;
 - **StringBuilder** – mutability and chaining; methods: `append()`, `insert()`, `delete()`, `reverse()`, `toString()`, `charAt()`, `indexOf()`, `length()` and `substring()`;
 - **Functional Programming** – lambdas, Stream API, default and static methods in interfaces;
 - **Java Date/Time API** – working with dates and times
 - **NIO2 API** – working with files and directories, Input and Output streams, Readers and Writers.
 - **JDBC** - database drivers, connecting to MySQL/PostgreSQL database using `DriverManager`. Using `Statement` / `PreparedStatement`. Using `ResultSet`.

Exception handling



Where to Find the Code?

Java Web Development projects and examples are available @ GitHub:

<https://github.com/iproduct/course-java-fd>

Exception Handling in Java

- Obligatory exception handling in Java → secure and reliable code
- Separation of concerns: business logic from exception handling code
- Class **Throwable** → classes Error и Exception
- Generating Exceptions – keyword **throw**
- Exception handling:
 - **try – catch – finally** block
 - Delegating the handling to the caller method - **throws**

Try-Catch-Finally Block

- Operator **try** for demarcation of un-reliable code, multiple **catch** blocks for catching exceptions, and **finally** clause for guaranteed clean-up in the end:

try {

// code that can throw Exceptions: Ex1, Ex2, ...

} catch(Ex1 ex) { // excuted only when Ex1 is thrown
// handling problem 1 (Ex1)

} catch(Ex2 ex) { // excuted only when Ex2 is thrown
// handling problem 2 (Ex2)

} finally {

// executed always independently of whether an exception
was thrown or not

}

Exception Handling in Java - II

- Implementing custom exceptions
- Using stack trace
- Unchecked exceptions extending **RuntimeException**
- Guaranteed completion using **finally**

Novelties in Exception Handling since Java 7

- **Multi-catch** clause:

```
catch (Exception1|Exception2 ex) {  
    ex.printStackTrace();  
}
```

- Program block **try-with-resources**

```
String readInvoiceNumber(String myfile) throws IOException  
{  
    try (BufferedReader input = new  
        BufferedReader(new  
            FileReader(myfile))) {  
        return input.readLine();  
    }  
}
```


Strings and Regular Expressions



Strings & StringBulider

- **String** class provides **immutable** objects – i.e. any operation on the string creates a new object in heap.
- Basic operations in the class **String** – `length()`, `charAt()`, `indexOf()`, `substring()`, `toLowerCase()/UpperCase()`, `equals()`, `equalsIgnoreCase()`, `startsWith()` and `endsWith()`, `contains()`, `replace()`, `trim()`
- **StringBulider** – it provides an efficient way to modify the strings, by implementing the Reusable Design Pattern: **Builder** – for incremental string building.
- Basic operations in the class **StringBulider** – `append()`, `insert()`, `delete()`, `reverse()`, `toString()`, `charAt()`, `indexOf()`, `length()` and `substring()`
- Formatted output - method **format()** and class **Formatter**. Specifiers: `%[argument_index$][flags][width][.precision]conversion`

Conversion in Type Formatting

- d – decimal, integral types
- c – character (unicode)
- b - boolean
- s - String
- f – float, double (with decimal point)
- e - float, double (scientific notation)
- x – hexadecimal value of integral types
- h – hexadecimal hash code

Regular Expressions - I

- Symbolic classes:
 - Any character (may or may not match line terminators)
 - **\d** A digit: [0-9]
 - **\D** A non-digit: [^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]

Regular Expressions - II

- Qualifiers:

–**X?** X, once or not at all

–**X*** X, zero or more times

–**X+** X, one or more times

–**X{n}** X, exactly n times

–**X{n,}** X, at least n times

–**X{n,m}** X, at least n but not more than m times

- **Greedy, Reluctant (?) & Possessive (+)** qualifiers
- **Capturing Group - (X)**

Regular Expressions - III

- Class **Pattern** – basic methods:

public static Pattern compile(String regex)

public Matcher matcher(CharSequence input)

public static boolean matches(String regex, CharSequence input)

public String[] split(CharSequence input, int limit)

- Class **Matcher** – basic methods:

public boolean matches()

public boolean lookingAt()

public boolean find(int start)

public int groupCount() и public String group(int group)

Functional Programming & Streams API



Functional Interfaces in Java 8+

- **Functional interface** = interface with **Single Abstract Method (SAM)** => **@FunctionalInterface**
- Examples:

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

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

```
public interface Runnable {  
    public void run();  
}
```

```
public interface Callable<V> {  
    V call() throws Exception;  
}
```

Labmda Expressions – `java.util.function`

Examples:

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

`() -> 42`

`(a, b) -> a * a + b * b;`

`(String s) -> { System.out.println(s); }`

`book -> book.getAuthor().fullName()`

`voter -> voter.getAge() >= legalAgeOfVoting`

`(person1, person2) -> person1.getAge() - person2.getAge()`

`(song1, song2) ->`

`song1.getArtist().compareTo(song2.getArtist())`

Lambda Expressions Formatting Rules

- **Lambda expressions (functions)** can have arbitrary number of **arguments**, which are enclosed in parentheses and separated by commas. Their type can be declared or not inferred by the context of use (**target typing**). If there is exactly one parameter, the parentheses are not required.
- **The body of lambda expressions** is composed by arbitrary language constructs (statements), each finishing with **;** and the body is enclosed in curly braces **}**. If there is a **single expression** that needs to be returned by the function, then we can **skip the curly braces and return** keyword – the value of the expression is automatically returned as function result.

Package **java.util.function**

- **Predicate<T>** – predicate = a boolean expression representing a property of the argument object
- **Function<A,R>**: function which receives an argument **A** and transforms it to a result **R**
- **Supplier<T>** – method **get()** returns a new instance each time it is called – an object factory
- **Consumer<T>** – accepts an argument (**accept()** method) and executes an operation consuming that argument
- **UnaryOperator<T>** – single argument operator $T \rightarrow T$
- **BinaryOperator<T>** – binary operator $(T, T) \rightarrow T$

Stream API (1)

Examples:

```
books.stream().map(book -> book.getTitle())  
    .collect(Collectors.toList());
```

```
books.stream()  
    .filter(w -> w.getDomain() == PROGRAMMING)  
    .mapToDouble(w -> w.getPrice()) .sum();
```

```
document.getPages().stream()  
    .map(doc -> Documents.characterCount(doc))  
    .collect(Collectors.toList());
```

```
document.getPages().stream()  
    .map(p -> pagePrinter.printPage(p))  
    .forEach(s -> output.append(s));
```

Stream API (2)

Examples:

```
document.getPages().stream()
    .map(page -> page.getContent())
    .map(content -> translator.translate(content))
    .map(translated -> new Page(translated))
    .collect(Collectors.collectingAndThen(
        Collectors.toList(),
        pages -> new
Document(translator.translate(document.getTitle()), pages)));
```

Method References

- Static method references – `Class::staticMethod`
- Instance method references – `object::instanceMethod`
- Class constructor references – `Class::new`

```
Comparator<Person> namecomp =  
Comparator.comparing(Person::getName);
```

```
Arrays.stream(pageNumbers)  
    .map(doc::getPageContent)  
    .forEach(Printers::print);
```

```
pages.stream().map(Page::getContent)  
    .forEach(Printers::print);
```

Static and Default Interface Methods

- Method with default implementation are called **virtual extension methods** or **defender methods**, because they allow the interfaces to be extended without breaking existing implementations.
- Static methods allow to provide additional utility methods, such as **factory** methods directly in the interfaces, instead of creating auxiliary utility classes such as **Arrays** and **Collections**.

Default and Static Methods Usage Example

@FunctionalInterface

```
interface Event {  
    Date getDate();  
    default String getDateFormatted() {  
        return String.format("%1$td.%1$tm.%1$tY", getDate());  
    }  
    public static <T, U extends Comparable<? super U>>  
    Comparator<T> comparing(Function<T, U> getKey) {  
        return (c1, c2) -> getKey.apply(c1).compareTo(getKey.apply(c2));  
    }  
}  
Event current = () -> new Date();  
System.out.println(current.getDateFormatted());
```

Resources

- Oracle tutorial – lambda expressions -
<http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>
- Java SE 8: Lambda Quick Start -
<http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html>
- OpenJDK Lambda Tutorial -
<https://github.com/AdoptOpenJDK/lambda-tutorial>

Date & Time API



Novelties in Java 8+: Date-Time API (JSR 310)

- Allows flexible processing (incl. time based calculations) with dates and periods
- Package: **java.time**
- Supported standards: **ISO-8601, Unicode Common Locale Data Repository (CLDR), Time-Zone Database (TZDB)**
- Example:

```
LocalDate today = LocalDate.now();  
LocalDate reportDay =  
today.with(TemporalAdjusters.lastDayOfMonth());  
LocalDate paymentDay = reportDay.plusDays(5);
```

Advantages of Date-Time API

- **Clear** - methods in the API are well defined and their behavior is clear and expected
- **Fluent** - provides a fluent interface, making the code easy to read, most methods do not allow null values => can be chained together:

```
LocalDate today = LocalDate.now();
```

```
LocalDate paymentDay = today  
    .with(TemporalAdjusters.lastDayOfMonth())  
    .minusDays(2);
```


Advantages of Date-Time API

- **Immutable** - after the object is created, it cannot be modified. To alter the value of an immutable object, a new object must be constructed as a modified copy of the original => thread-safe. This affects the API in that most of the methods used to create date or time objects are prefixed with `of`, `from`, or `with`, rather than constructors, and there are no set methods. For example:

```
LocalDate dateOfBirth =  
    LocalDate.of(2012, Month.MAY, 14);  
LocalDate firstBirthday = dateOfBirth.plusYears(1);
```

- **Extensible** - wherever possible. For example, you can define your own time adjusters and queries, or build your own calendar system.

Date and Time API: Main Classes (1)

- **Clock** – allows access to the current moment, date and time for a time zone
- **Instant** – momentary point on the time axis
- **LocalDate** – local date without time and zone: 2014-12-20
- **LocalTime** – local time without date and zone: 14:25:15
- **LocalDateTime** – local date and time without zone: 2014-12-20T14:25:15
- **MonthDay** – day of month –12-20 => December 20
- **Duration** – time period – e.g. 2 minutes and 52 seconds
- **Period** – time period in days – e.g. 3 years 2 months and 4 days

Date and Time API: Main Classes (2)

- **OffsetDateTime** – date and time + time zone:
2014-12-20T09:15:00+02:00
- **OffsetTime** – time + time zone: 09:15:00+02:00.
- **Year** – year - e.g. 2014
- **YearMonth** – month in year – напр. 2014-12
- **ZonedDateTime** – similar to OffsetDateTime + time-zone
ID: 2014-12-20T10:15:30+01:00 Europe/Sofia
- **ZonedId** – time-zone ID – e.g. Europe/Rome
- **ZoneOffset** – time offset from Greenwich/UTC – e.g. +02:00

Date and Time API – Example

```
LocalDate today = LocalDate.now();
LocalDate dateOfBirth = LocalDate.of(1982, Month.MAY, 14);
LocalDateTime now = LocalDateTime.now();
LocalTime timeOfBirth = LocalTime.of(14, 50);
LocalDateTime dateTimeOfBirth = LocalDateTime.of(dateOfBirth,
timeOfBirth);
Period howOld = Period.between(dateOfBirth, today);
Duration age = Duration.between(dateTimeOfBirth, now);
long daysOld = ChronoUnit.DAYS.between(dateOfBirth, today);
System.out.println("Your age are: " + howOld.getYears() + " years, "
+ howOld.getMonths() + " months, and " + howOld.getDays()
+ " days. (" + age.toDays() + " /" + daysOld + "/ days total)");
```

Java IO. New IO (NIO) 2



Agenda for This Session

- I/O basics,
- AutoCloseable,
- Closeable and Flushable interfaces,
- I/O exceptions,
- Serialization,
- java.io. and nio

Java I/O

- Input/Output from/to:
 - ✓ Memory
 - ✓ String
 - ✓ Between different threads
 - ✓ Files
 - ✓ Console
 - ✓ Network sockets
- Different data types – bytes / characters. Encoding.
- Common and extensible architecture of Java I/O system using Decorator design pattern.

Class **File** – Working with Files and Dirs

- Class *File*
- Represents a file or a directory.
- Methods *getName()* and *list()*
- Getting file information
- Creating, renaming and deleting directories.

Input and Output Streams

- Input streams – class ***InputStream*** and its inheritors
- Output streams – class ***OutputStream*** and its inheritors
- Decorator design pattern
- Decorators - class ***FilterInputStream*** and its inheritors, class ***FilterOutputStream*** and its inheritors

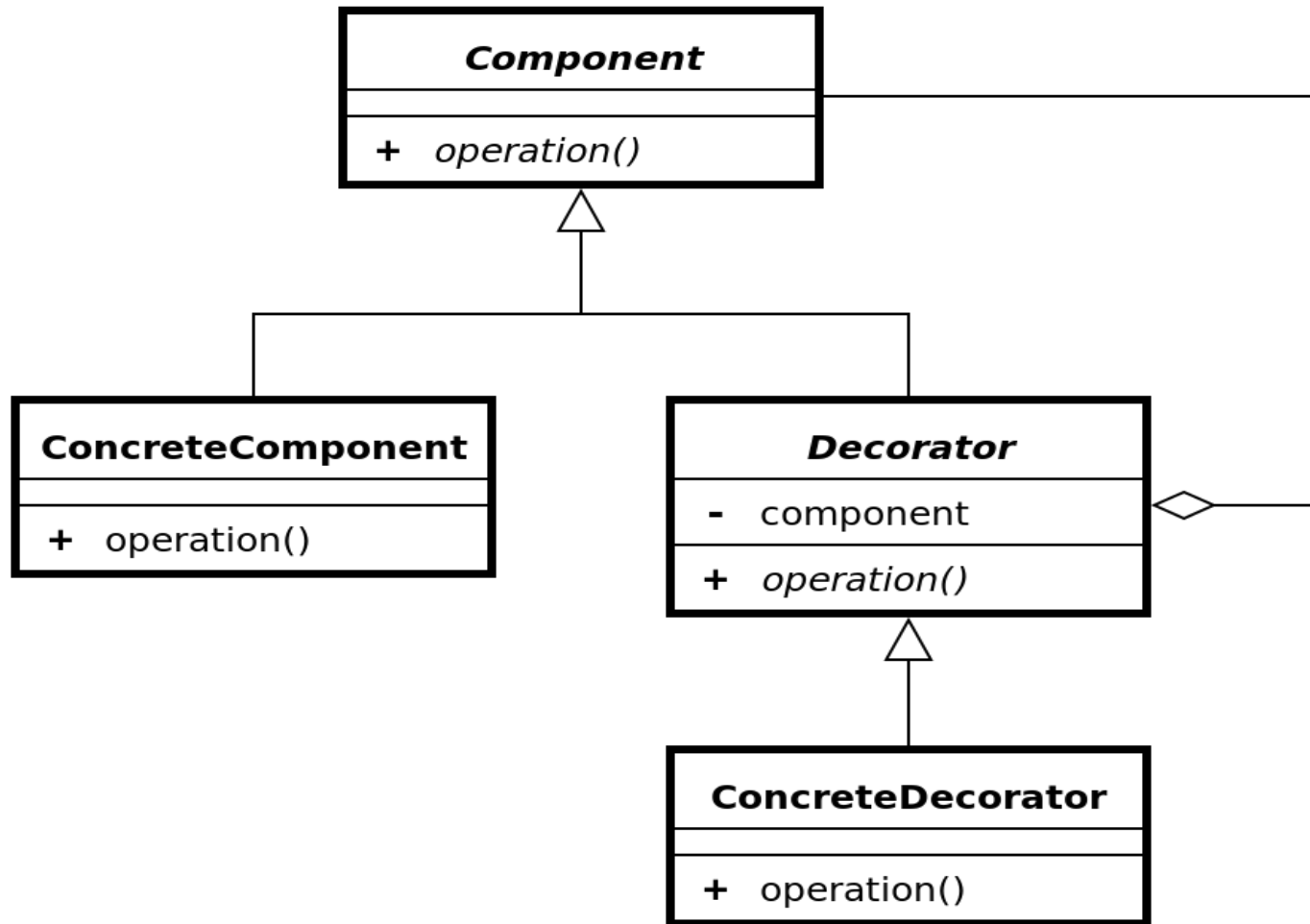
Input Streams: *InputStream*

- **FileInputStream** – reads data from file
- **ByteArrayInputStream** – reads data from memory
- **StringBufferInputStream** – reads data from StringBuffer
- **ObjectInputStream** – de-serializes Objects and primitives
- **PipedInputStream** – receives data from another thread
- **SequenceInputStream** – combines several InputStreams
- **FilterInputStream** – decorates wrapped input streams with additional functionality

Output Streams: *OutputStream*

- **FileOutputStream** – writes data to file
- **ByteArrayOutputStream** – writes data to memory buffer
- **ObjectOutputStream** – serializes objects and primitives
- **PipedOutputStream** – sends data to another thread
- **FilterOutputStream** – decorates wrapped InputStreams with additional functionality

Decorator Design Pattern



Input Stream Decorators

- **DataInputStream** – reads primitive types
- **BufferedInputStream** – buffers the input, allows reading lines instead of characters
- **DigestInputStream** – calculates content hash using algorithms such as: SHA-1, SHA-256, MD5
- **DeflaterInputStream** – data compression
- **InflaterInputStream** – data decompression
- **ChecksumInputStream** – calculates checksum (Adler32, CRC32)
- **CipherInputStream** – deciphers data (using Cipher)

Output Stream Decorators

- **PrintStream** – provides convenient methods for printing different data types, processes exceptions
- **DataOutputStream** – writes primitive data types
- **BufferedOutputStream** – output buffering
- **DigestOutputStream** – calculates content hash using algorithms such as: SHA-1, SHA-256, MD5
- **DeflaterOutputStream** – data compression
- **InflaterOutputStream** – data decompression
- **CheckedOutputStream** – checksum computation
- **CipherInputStream** – encrrips data (using Cipher)

Reading Character Data: *Reader*

Adaptor Class: *InputStreamReader*

- **FileReader** – reads character data from file
- **CharArrayReader** - reads character data from memory
- **StringReader** – reads character data from String
- **PipedReader** – receives character data from a thread
- **FilterReader** – Reader decorator base class

Writing Character Data : *Writer*

Adaptor Class: *OutputStreamWriter*

- **FileWriter** – writes character data to file
- **CharArrayWriter** - writes character data to array
- **StringWriter** – writes character data to StringBuffer
- **PipedWriter** – sends character data to another thread
- **FilterWriter** – base class for Writer decorators
- **PrintWriter** – formatted output in string format, handles all exceptions

Reader / Writer Decorators

- **BufferedReader** – character input buffering
- **PushbackReader** – allows characters to be read without consuming
- **BufferedWriter** – character output buffering
- **StreamTokenizer** – allows parsing of character input (from Reader) token by token

Direct Access Files

- Class ***RandomAccessFile***.
 - Access modes
 - Method ***seek()***
 - Usage examples.
-
- Standard I/O to/from console. Redirecting.

New More Effective I/O Implementation: New I/O

- Java New I/O – package ***java.nio.**** introduced in JDK 1.4
- Uses low level OS mechanisms and structures to allow more effective, faster and non-blocking IO.
- Underlying all types of Streams (FileInputStream, FileOutputStream, RandomAccessFile) as well as network socket streams.

New More Effective I/O Implementation: New I/O

- Buffers for primitive data types: **java.nio.Buffer**, **ByteBuffer**, **CharBuffer**, **DoubleBuffer**, **FloatBuffer**, **IntBuffer**, **LongBuffer**, **ShortBuffer**
- Channels – new low level IO abstraction: **java.nio.channels.Channel**, **FileChannel**, **SocketChannel**
- Supports different encodings: **java.nio.charset.Charset**

New More Effective I/O Implementation: New I/O

- Supports read/write locking of arbitrary sections of a file up to `Integer.MAX_VALUE` байта (2 GiB). Depending on OS can allow shared locking: **`tryLock()`** or **`lock()`** of the class **`java.nio.channels.FileChannel`**
- Allows multiplexing of I/O operations for implementing scalable servers processing multiple sessions using a single thread asynchronously: **`java.nio.channels.Selector`** и **`SelectableChannel`**

Compression: GZIP, ZIP. JAR Files

- File compression – gzip, zip. Check Sum.
- Application deployment using .jar archives. JAR file manifest.
- **jar** [options] archive [manifest] files

c – creates new archive

x / x файл – extracts specific/all files from an archive

t – prints archive content table

f – necessary to specify the file we read/write from/to

m – if we provide a manifest file

M – do not create manifest file automatically

0 – without compression

v – verbose output

Object Serialization

- Interface ***Serializable*** – all fields are serialized except those marked as ***transient***
- Interface ***Externalizable*** – we serialize all fields explicitly
- Methods ***readObject()*** and ***writeObject()*** – ***Serializable*** + customization where necessary
- Examples

Novelties in Java 7 - JSR 203: NIO.2 (1)

- New NIO packages: `java.nio.file`, `java.nio.file.attribute`
- **FileSystem** – allows a unified access to different file systems using URI or the method `FileSystems.getDefault()`. A factory for file system object creation. Methods: `getPath()`, `getPathMatcher()`, `getFileStores()`, `newWatchService()`, `getUserPrincipalLookupService()`.
- **FileStore** – models a drive, partition or a root directory. Can be accessed using `FileSystem.getFileStores()`

Novelties in Java 7 - JSR 203: NIO.2 (1)

- **Path** – represents a file or directory path in the file system. Has a hierarchical structure – a sequence of directories separated using an OS specific separator ('/' или '\'). Provides methods for composing, decomposing, comparing, normalizing, transforming relative and absolute paths, watching for file and directory changes, conversion to/from **File** objects (`java.io.File.toPath()` и `PathToFile()`).
- **Files** – utility class providing static methods for manipulation (creation, deletion, renaming, attributes change, content access, automatic MIME type inference, etc.) of files, directories, symbolic links, etc.

Resources

- Sun Microsystems Java™ Technologies webpage – <http://java.sun.com/>
- New I/O във Wikipedia: http://en.wikipedia.org/wiki/New_I/O
- Уроци за новостите в JSR 310: Date and Time API <http://docs.oracle.com/javase/tutorial/datetime/>
- Уроци за новостите в JSR 203: NIO.2 <http://download.oracle.com/javase/tutorial/essential/io/fileio.html>

Java DataBase Connectivity (JDBC)



Java Database Connectivity (JDBC)

- **Java Database Connectivity (JDBC)** is an **application programming interface (API)** in **Java**, which allows the clients to access, extract, and modify data in a relational database.
- **JDBC-to-ODBC bridge** allows connections to all ODBC-compatible data-sources from Java.

JDBC Example (1)

```
Scanner sc = new Scanner(System.in);
Properties props = new Properties();

System.out.println("Enter username (default root): ");
String user = sc.nextLine().trim();
user = user.length() > 0 ? user : "root";
props.setProperty("user", user);

String password = sc.nextLine().trim();
password = password.length() > 0 ? password : "root";
props.setProperty("password", password);
```


JDBC Example (2)

// 1. Load jdbc driver (optional)

```
try {  
    Class.forName("com.mysql.cj.jdbc.Driver");  
} catch (ClassNotFoundException e) {  
    e.printStackTrace();  
    System.exit(0);  
}  
System.out.println("Driver loaded successfully.");
```

// 2. Connect to DB

```
Connection connection =  
    DriverManager.getConnection(  
        "jdbc:mysql://localhost:3306/employees?useSSL=false",  
        props);  
System.out.println("Connected successfully.");
```

JDBC Example (3)

// 3. Execute query

```
PreparedStatement stmt =  
    connection.prepareStatement("SELECT * FROM employees JOIN  
salaries ON employees.emp_no=salaries.emp_no WHERE  
salaries.salary > ?");
```

```
System.out.println("Enter minimal salary (default 20000): ");  
String salaryStr = sc.nextLine().trim();
```

```
double salary = Double.parseDouble(salaryStr);
```

```
stmt.setDouble(1, salary);  
ResultSet rs = stmt.executeQuery();
```

JDBC Example (4)

```
// 4. Process results
while (rs.next()) {
    System.out.printf("| %-15.15s | %-15.15s | %10.2f |\n",
        rs.getString(2),
        rs.getString("last_name"),
        rs.getDouble("salary")
    );
}

// 5. Close connection and statement
connection.close();
```

Novelties in JDBC™ 4.1: try-with-resources

- `java.sql.Connection`, `java.sql.Statement` and `java.sql.ResultSet` implement the interface **AutoCloseable**:

```
Class.forName("com.mysql.jdbc.Driver");           //Load MySQL DB driver
try (Connection c = DriverManager.getConnection(dbUrl, user, password);
    Statement s = c.createStatement() ) {
    c.setAutoCommit(false);
    int records = s.executeUpdate("INSERT INTO product " //Insert new product
        + "VALUES ('CP-00002', 'Lenovo', 790.0, 'br', 'Laptop')");
    System.out.println("Successfully inserted "+ records + " records.");
    records = s.executeUpdate("UPDATE product "         //Update product price
        + "SET price=470, description='Classic laptop' "
        + "WHERE code='CP-00001'");
    System.out.println("Successfully updated "+ records + " records.");
    c.commit();                                         //Finish transaction
}
```

Resources

- Wikipedia – Free Online Enciclopedia – <http://wikipedia.org/>
- Oracle® Java™ Technologies webpage – <http://www.oracle.com/technetwork/java/>
- Oracle®: The Java Tutorials: Lesson: JDBC Basics – <http://docs.oracle.com/javase/tutorial/jdbc/basics/index.html>
- Oracle®: Новости в JDBC™ 4.1 – http://docs.oracle.com/javase/7/docs/technotes/guides/jdbc/jdbc_41.html
- Joshua Bloch: Automatic Resource Management (V.2) – https://docs.google.com/View?id=ddv8ts74_3fs7483dp

Thank's for Your Attention!



Trayan Iliev

CEO of IPT – Intellectual Products
& Technologies

<http://iproduct.org/>

<https://github.com/iproduct>

<https://twitter.com/trayaniliev>

<https://www.facebook.com/IPT.EACAD>