

# Medii de proiectare și programare

2022-2023

# Conținut

- Build automation tool (Gradle)
- Conectarea la baze de date relaționale
- Inversion of Control - Spring
- Aplicații client-server - Șablonul de proiectare Proxy
- Introducere în apelul metodelor la distanță (Remote Procedure Call) - **Remoting/WCF, RMI, Spring Remoting**
- Enterprise Application Integration - Protocol buffers, Thrift, ActiveMQ, RabbitMQ
- Object Relational Mapping (Strategii, Hibernate, Entity Framework)
- REST Services
- Dezvoltarea aplicațiilor web folosind frameworkuri (React)
- Web sockets
- Securitate Web - Acces bazat pe roluri (JWT Token)

# Bibliografie

- Joseph Albahari, Ben Albahari, *C# 6.0 in a Nutshell*, Sixth Edition, O'Reilley, 2015.
- Craig Larman, *Applying UML and Design Patterns: An Introduction to OO Analysis and Design and Unified Process*, Berlin, Prentice Hall, 2002.
- Martin Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2002.
- Hohpe, G., Woolf, B., *Enterprise integration patterns*, Addison-Wesley, 2003.
- \*\*\*, Microsoft Developer Network, Microsoft Inc., <http://msdn.microsoft.com/>
- \*\*\*, The Java Tutorials, Oracle Java Documentation, Inc. <https://docs.oracle.com/javase/tutorial/>
- Craig Walls, *Spring in Action*, 4th Edition, Ed. O'Reilley, 2015.
- Documentație Spring <http://spring.io/projects>
- Alte tutoriale

# Evaluare

- Întrebări în timpul cursului (**QC**) - **10%**
- Examen (**NE**) - **50%**
- Teme în timpul laboratorului (**TL**) - **10%**
- Laboratoare (**LB**) - **30 %**
- Nota finală **NF=0.1\*QC+0.5\*NE+0.1\*TL+0.3\*LB**
- Condiții pentru promovare:
  - **NE>=5, LB>=4.5**
  - **NF>=4.5**

# Laborator

- Asignarea și proiectarea aplicației. Studenții trebuie să proiecteze și să dezvolte o aplicație **client-server**.
- Configurarea aplicației folosind *Gradle*, IoC.
- Proiectarea și implementarea persistenței (baze de date relaționale, ORM)
- Proiectarea și implementarea serviciilor (*Şablonul Proxy*).
- Enterprise Application Integration (*Protobuf/ Thrift/ gRPC*)
- Servicii REST
- Aplicații web (web sockets / securitate ).

# Notare laborator

- Fiecare temă de laborator are un termen de predare.
- Predare completă a temei la termen: **nota 10**.
- Pentru fiecare săptămână întârziere în care nu s-a predat nimic din temă: **penalizare 3 puncte/săpt.**
- Pentru fiecare săptămână întârziere în care s-a predat ceva (este incompletă, erori): **penalizare 1 punct/săpt,** pâna la predarea completă a temei.
- Tema nepredată: **nota 0.**
- Tema copiată: **nota 0.**



**Titlu proiect:** Azi Student, Mâine Antreprenor!

**Acronim proiect:** ASMA

**Cod proiect:** POCU/379/6/21/123894

**Beneficiar:** Universitatea Babeș-Bolyai din Cluj-Napoca

**Proiect cofinanțat din FSE prin Programul Operațional Capital Uman 2014-2020**

# Curs 1

2022-2023

# Curs 1

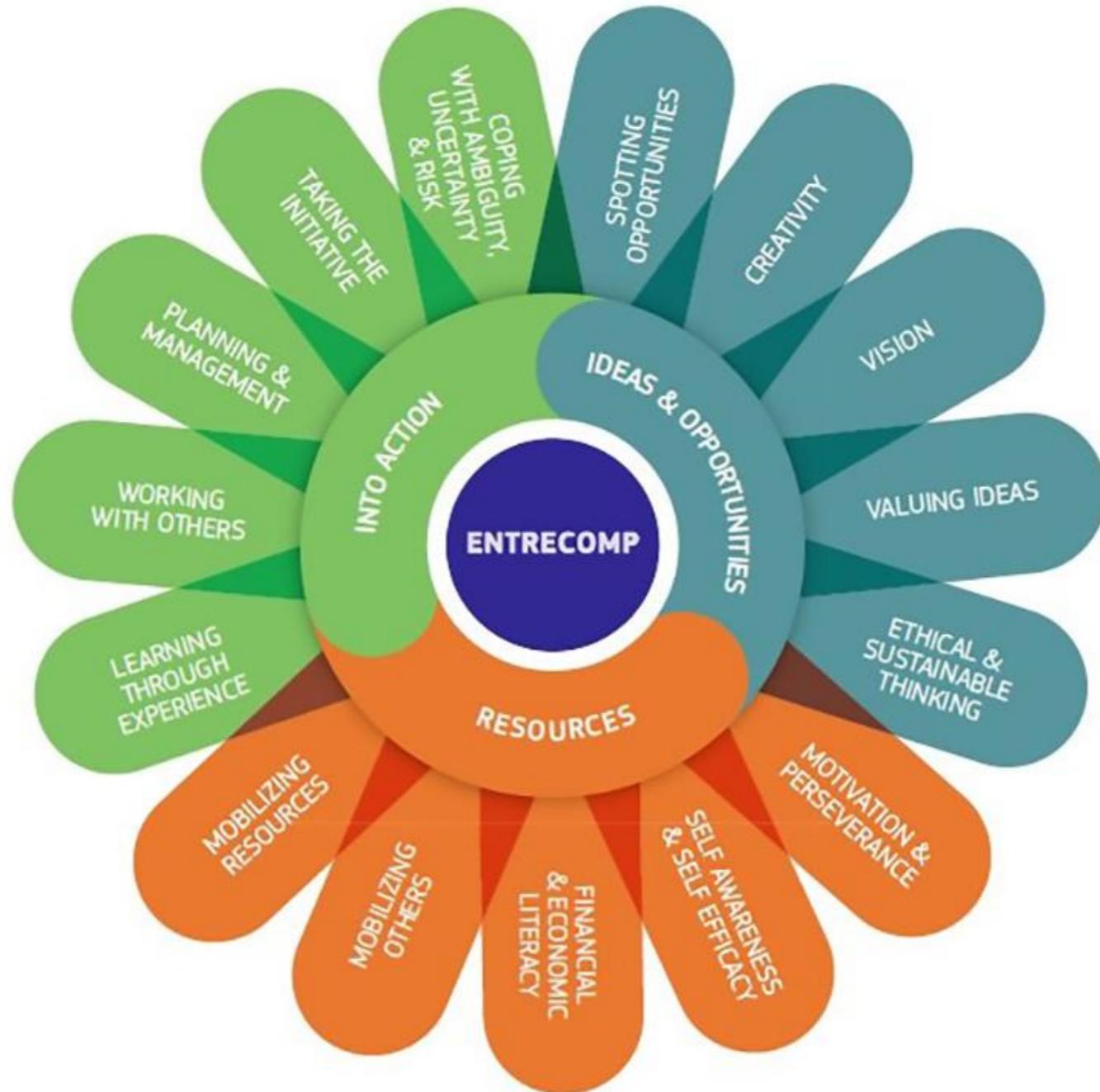
- Competențe antreprenoriale (informatică)
- Instrumente pentru construire automată
  - Gradle

# Competențe antreprenoriale



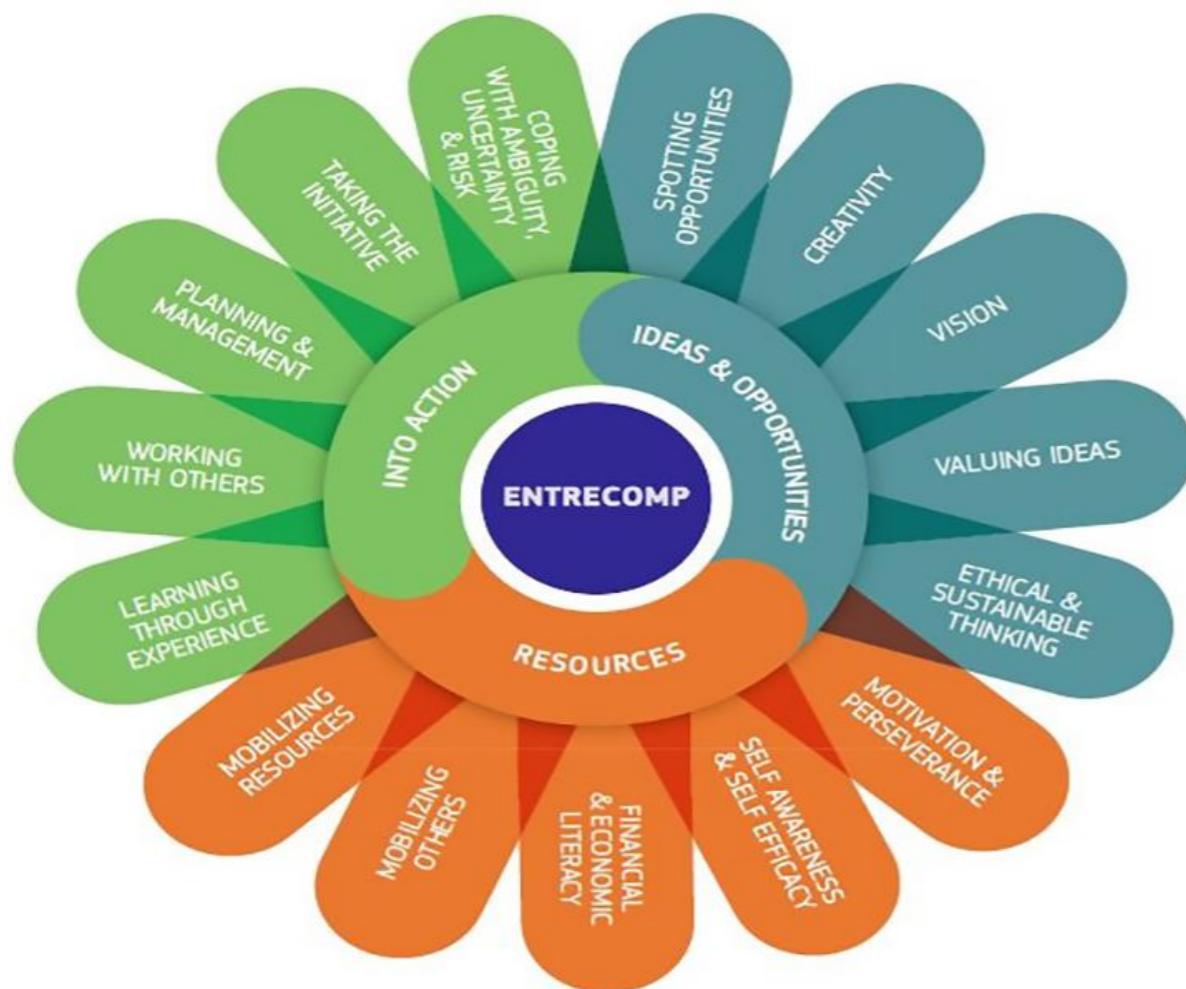
Key competences for lifelong learning, European Commission, Directorate-General for Education, Youth, Sport and Culture, 2019

# Competențe antreprenoriale



**EntreComp** -The European Entrepreneurship Competence Framework, 2018

# Informatică - competențe antreprenoriale



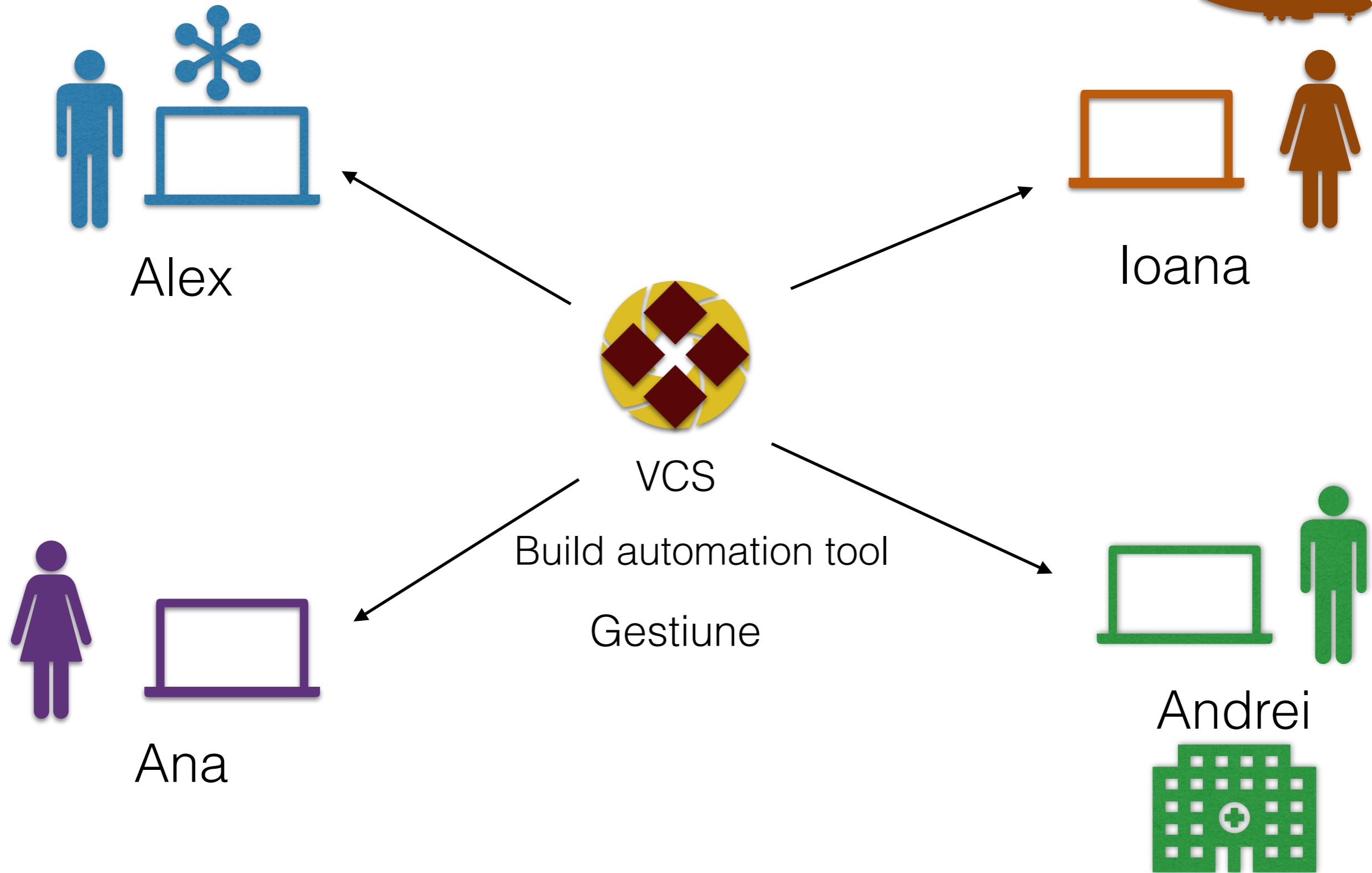
- **Lucrul în echipă**
- **Învățarea prin experiență**
- **Creativitate**
- **Identificarea de oportunități**

**Proiect semestrial facultativ!**

# *Instrumente pentru lucrul în echipă*

- Gestiunea proiectelor soft:
  - Jira, Trello, Atlassian, Zoho, etc.
- Controlul versiunilor și colaborare:
  - GitHub, Git, GitLab, Atlassian Git, BitBucket, etc.
- Construire automată:
  - Gradle, Maven, Jenkins, Travis CI, etc.
- Jurnalizare:
  - Log4j2, SLF4J, Logger.NET, NLog, etc.

# Instrumente pentru lucrul în echipă



# Instrumente pentru construire automată\*

- **Procesul automatizat** corespunzător **construirii** unui sistem soft și a proceselor asociate (compilarea codului sursă, rularea testelor automate, împachetarea codului binar, etc).
  - Makefile
- Categorii:
  - a. **Utilitare** pentru construire automată:
    - Generează artefactele corespunzătoare construirii în timpul compilării și/sau link-editării codului.
    - Make, MS build, Ant, **Gradle**, Maven etc.
  - b. **Servere** pentru construire automată:
    - Sisteme soft care execută/rulează utilitarele de construire automată la perioade de timp predefinite sau în momentul apariției anumitor evenimente.
    - TeamCity, Jenkins, CruiseControl, etc.

\*Eng. *Build automation tools*

# Utilitate pentru construire automată

## Avantaje

- Permit **automatizarea** sarcinilor *simple și repetabile*.
- Permit **atingerea** unui scop/**obiectiv** prin execuția fiecărei sarcini dintr-un set de sarcini specifice în ordinea corectă.

Îmbunătățesc calitatea produselor

Accelerează compilarea și link-editarea

Elimină sarcinile redundante

Elimină dependențele de angajați ‘cheie’

Oferă un **jurnal** al **construcțiilor** ce pot fi folosite când apar **probleme**

Reduc numărul **construcțiilor eșuate**

Economisesc **timp și bani**



# Gradle Build Tool

- Gradle Build Tool, <https://gradle.org/>
- Guides <https://gradle.org/guides/#getting-started>
- Tutoriale:
  - <http://www.vogella.com/tutorials/Gradle/article.html#introduction-to-the-gradle-build-system>
  - <https://www.petrikainulainen.net/getting-started-with-gradle/>
  - Altele ...

**Permite gestiunea avansată a construirii aplicațiilor**

**Oferă suport pentru configurarea și descărcarea automată a bibliotecilor și a altor dependențe**

**Permite refolosirea artefactelor construite folosind alte utilitare**



## Gradle Build Tool

**Oferă suport pentru depozitarele\* Maven și Ivy pentru refacerea dependențelor**

**Oferă suport pentru construiri multi-project și multi-artefacte**

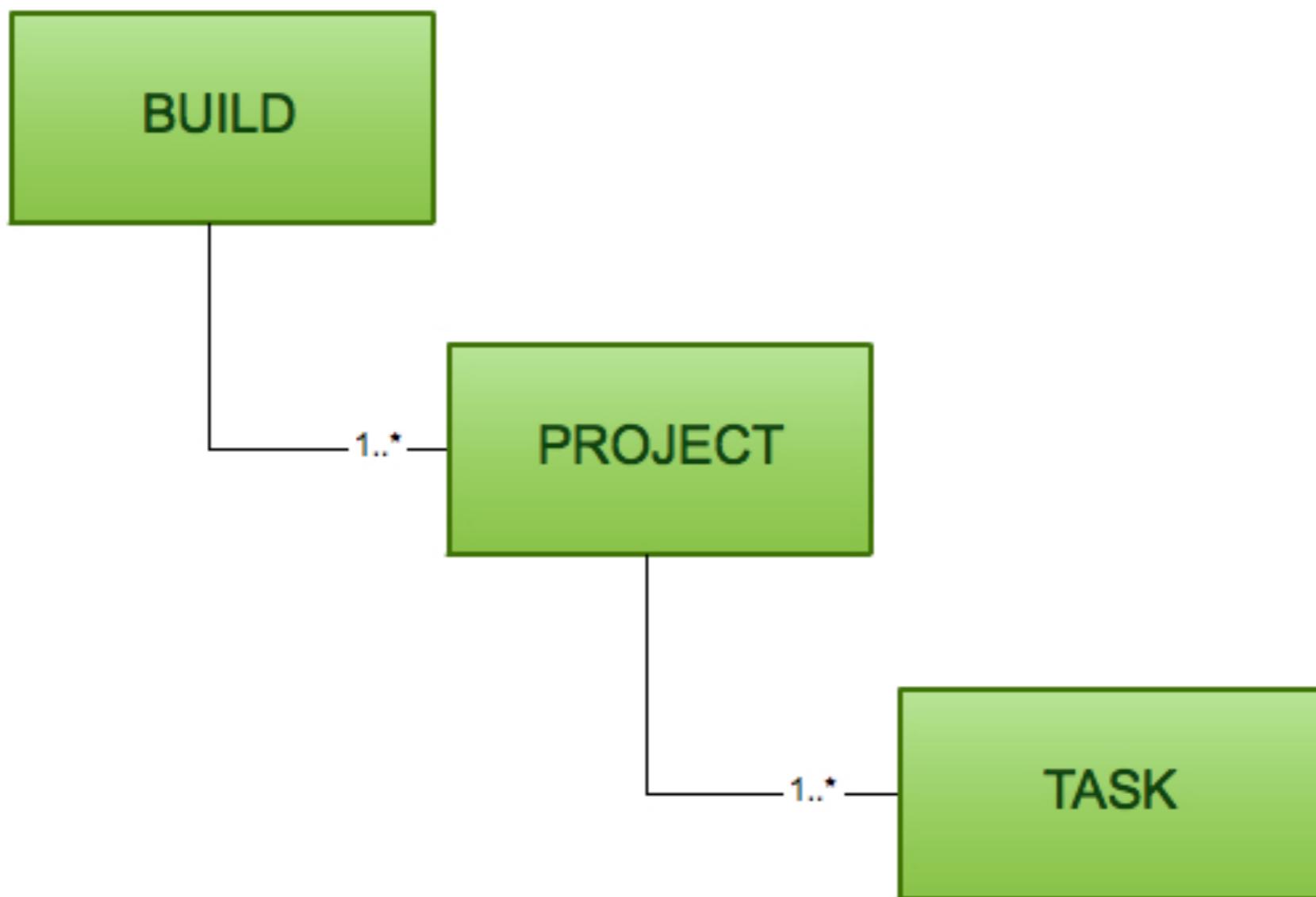
# Gradle Build Tool

- Concepte de bază:
  - proiect (eng. *project*)
  - sarcini (eng. *tasks*).
- ❖ Un project este:
  - ceva ce se dorește a se *construi* (ex. un sistem soft, un fișier jar),
  - ceva ce se dorește a se *face* (instalarea unui aplicații pentru folosirea de către utilizatori).
- ❖ Un proiect are asociate una sau mai multe *sarcini*.
- ❖ O sarcină este o *unitate de lucru* care este efectuată pentru construirea automată:
  - compilarea unui proiect
  - rularea testelor automate
  - ...



# Gradle Build Tool

- Fiecare construire Gradle conține unul sau mai multe proiecte.





- O construire Gradle se configurează folosind fișierele:
  - `build.gradle` (*Gradle build script*) - specifică un proiect și sarcinile sale asociate.
  - `gradle.properties` (*Gradle properties file*) - este folosit pentru configurarea proprietăților construirii.
  - `settings.gradle` (*Gradle Settings file*) - optional la construirile ce conțin un singur proiect.
    - Dacă construirea este formată din mai multe proiecte Gradle, fișierul este obligatoriu și descrie proiectele conținute.
    - Fiecare construire multi-proiect trebuie să aibă un fișier `settings.gradle` în directorul rădăcină al proiectului.



- **Exemplu fișier build.gradle**

```
apply plugin: 'java'

repositories {
    //pentru rezolvarea dependențelor: Maven, JCenter, Ivy
    mavenCentral()
    //jcenter()
}

dependencies {
    //format GAV Grup:Artefact:Versiune
    compile 'com.google.guava:guava:20.0'    //versiuni vechi
    testCompile group: 'junit', name: 'junit', version: '4.+' //versiuni vechi
    //alte dependențe

//versiuni noi Gradle
    implementation 'com.google.guava:guava:20.0'
    testImplementation group: 'junit', name: 'junit', version: '4.+'
```



- settings.gradle

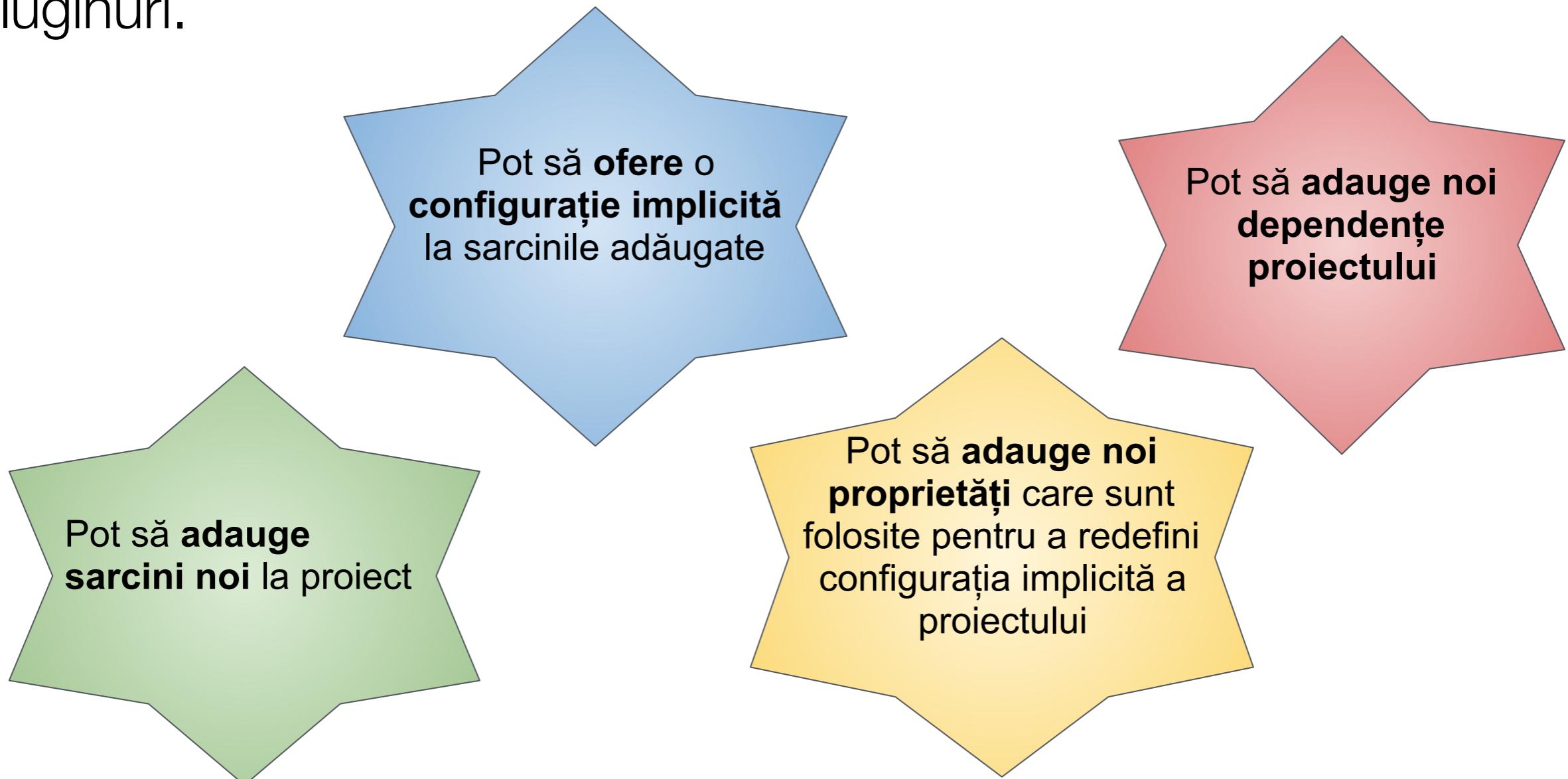
```
rootProject.name = 'NumeProjetc'
```



# Gradle Build Tool

## Pluginuri

Toate caracteristicile/facilitățile utile unui proiect sunt furnizate de pluginuri.





- Un plugin poate fi adăugat unui proiect folosind numele sau tipul acestuia:
- Exemplu folosind numele:

În fișierul build.gradle:

```
apply plugin: 'foo'
```

- Exemplu folosind tipul:

În fișierul build.gradle:

```
apply plugin: 'com.bar.foo'
```



- Blocul plugins (versiuni mai **noi** Gradle):

```
plugins {
```

```
    id 'java'
```

```
    id 'application'
```

```
}
```

- Se poate adăuga și versiunea unui plugin

```
    id 'org.openjfx.javafxplugin' version '0.0.7'
```



- Pluginuri Gradle standard:
  - `java`: adaugă sarcini pentru compilarea, testarea și împachetarea unui proiect Java. Este un plugin de bază pentru alte pluginuri.
  - `groovy`: adaugă suport pentru proiecte Groovy.
  - `cpp`: adaugă sarcini pentru compilarea codului unui proiect C++.
  - `application`: adaugă sarcini pentru rularea și împachetarea unui proiect Java ca și aplicație executabilă.
  - `distribution`: adaugă suport pentru construirea distribuțiilor de tip ZIP și TAR.
  - `signing`: adaugă abilitatea de a semna digital fișierele și artefactelor construite.
  - etc.



- Structura unui proiect Java:
  - *src/main/java* - directorul corespunzător codului sursă.
  - *src/main/resources* - directorul corespunzător resurselor folosite în proiect (fișiere de proprietăți, imagini, fxml etc.).
  - *src/test/java* - directorul corespunzător testelor automate.
  - *src/test/resources* - directorul corespunzător resurselor necesare testelor automate.
  - *build* - directorul ce conține toate artefactele construite folosind Gradle.
    - *classes* - conține fișierele *.class*.
    - *libs* - conține fișierele *jar*, *war*, *ear* create folosind Gradle.
    - etc.



- Crearea unui proiect Java

În fișierul `build.gradle`:

```
apply plugin: 'java'  
  
plugins{  
    id 'java'  
}
```

- Crearea unui proiect Java cu structura implicită\*:

`gradle init --type 'java-library'`

`gradle init --type 'java-application'`

\*În directorul rădăcină al proiectului



# Gradle Build Tool

- Sarcinile asociate unui proiect Java (plugin java):
  - *assemble* - compilează codul sursă al aplicației și creează fișierul jar. Nu rulează testele automate.
  - *build* - construiește toate artefactele asociate proiectului.
  - *clean* - șterge directorul **build** asociat proiectului.
  - *compileJava* - compilează codul sursă al aplicației.
  - etc.



- **gradle tasks** - afișează lista completă a sarcinilor ce pot fi executate pentru un proiect și descrierea acestora:
  - *assemble* - Assembles the outputs of this project.
  - *build* - Assembles and tests this project.
  - *buildDependents* - Assembles and tests this project and all projects that depend on it.
  - *buildNeeded* - Assembles and tests this project and all projects it depends on.
  - *classes* - Assembles main classes.
  - *clean* - Deletes the build directory.
  - *jar* - Assembles a jar archive containing the main classes.
  - *testClasses* - Assembles test classes.
  - *check* - Runs all checks.
  - *test* - Runs the unit tests.
  - etc.



- Împachetarea aplicației (obținerea artefactelor):
  - *gradle assemble*
  - :compileJava*
  - :processResources*
  - :classes*
  - :jar*
  - :assemble*



- Împachetarea aplicației (obținerea artefactelor și rularea testelor automate):

- *gradle build*

- :compileJava*

- :processResources*

- :classes*

- :jar*

- :assemble*

- :compileTestJava*

- :processTestResources*

- :testClasses*

- :test*

- :check*

- :build*

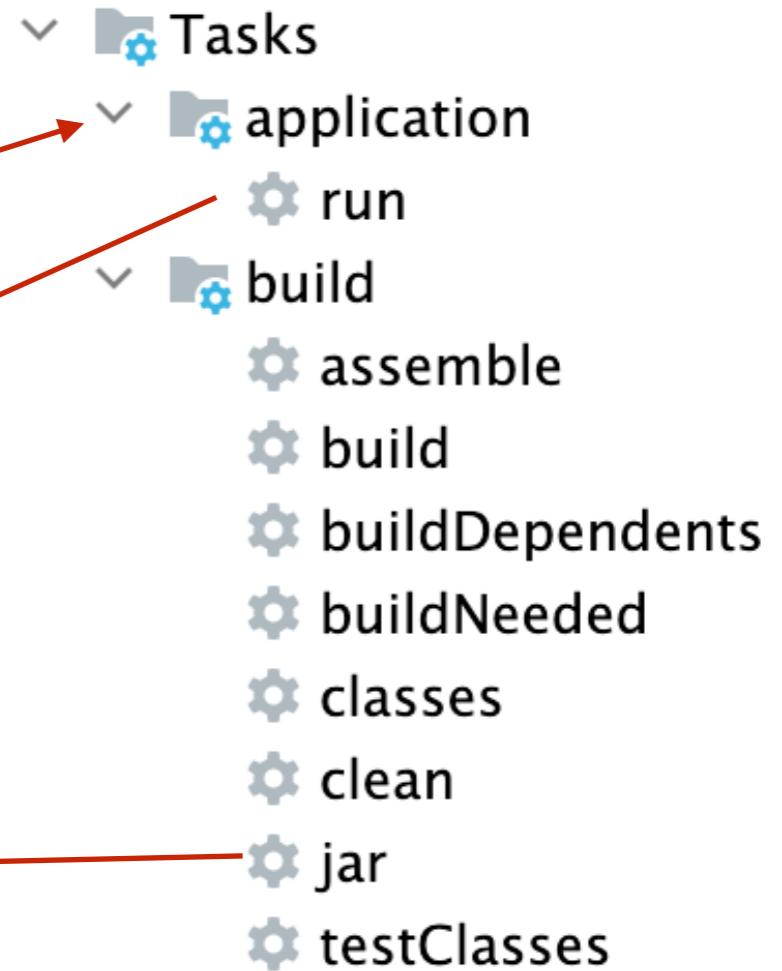


# Gradle Build Tool

- Fișiere *jar executable*:

- *build.gradle*

```
plugins{
    id 'java'
    id 'application'
}
application{
    mainClass='NumeClasaCuMain'
}
jar {
    manifest {
        attributes('Main-Class':'NumeClasaCuMain')
    }
    from {
        configurations.runtimeClasspath.collect { it.isDirectory() ? it : zipTree(it) }
    }
}
```





- Proiecte multiple:
  - Fiecare proiect (subproiect) are aceeași structură - corespunzătoare proiectelor Gradle Java.
  - Fiecare proiect (subproiect) va conține fișierul `build.gradle` propriu, cu configurațiile specifice.
  - Proiectul rădăcină (root) conține obligatoriu și fișierul `settings.gradle`:

`include 'A'`

`include 'B'`



# Gradle Build Tool

- Dependențe între (sub)proiecte: Subproiectul B depinde de subproiectul A:
  - build.gradle corespunzător subproiectului B:

```
dependencies {  
    implementation project(':A')  
}
```



- Proiectul A: build.gradle

```
plugins{
    id 'java'

}

repositories {
    mavenCentral()

}

dependencies {
    implementation 'com.google.guava:guava:20.0'
    testImplementation 'junit:junit:4.11'

}
```



- Proiectul B: build.gradle

```
plugins{
    id 'application'
    id 'java'
}

repositories {
    mavenCentral()
}

application{
    mainClass='StartApp'
}

dependencies {
    testImplementation 'junit:junit:4.11'
    implementation project(':A')
}
```



- Project Root: build.gradle

```
allprojects {  
    plugins{  
        id 'java'  
    }  
    repositories {  
        mavenCentral()  
    }  
    dependencies {  
        testImplementation 'junit:junit:4.11'  
    }  
}
```



# Gradle Build Tool

- Proiectul A: build.gradle modificat

```
dependencies {  
    implementation 'com.google.guava:guava:20.0'  
}
```

- Proiectul B: build.gradle modificat

```
plugins{  
    id 'application'  
}  
  
application{  
    mainClass='StartApp'  
}  
  
dependencies {  
    implementation project(':A')  
}
```



- Proiectul Root: build.gradle

```
subprojects {  
    //Configurări comune tuturor subproiectelor  
}
```

```
project(':A') {  
    //Configurări specifice proiectului A  
}
```

```
project(':B') {  
    //Configurări specifice proiectului B  
}
```

# Medii de proiectare și programare

2022-2023

Curs 2

# Conținut curs 2

- Gradle (cont. curs 1)
- Jurnalizare
- SQLite, SQLiteStudio, MySQL
- Accesul la baze de date relaționale
  - Java: JDBC
  - C#: ADO.NET - curs 3
- Configurarea (Java properties, C# app.config - curs 3)
- Ierarhia repository - curs 3



- Proiecte multiple:
  - Fiecare proiect (subproiect) are aceeași structură - corespunzătoare proiectelor Gradle Java.
  - Fiecare proiect (subproiect) va conține fișierul `build.gradle` propriu, cu configurațiile specifice.
  - Proiectul rădăcină (root) conține obligatoriu și fișierul `settings.gradle`:

`include 'A'`

`include 'B'`



# Gradle Build Tool

- Dependențe între (sub)proiecte: Subproiectul B depinde de subproiectul A:
  - build.gradle corespunzător subproiectului B:

```
dependencies {  
    implementation project(':A')  
}
```



- Proiectul A: build.gradle

```
plugins{
    id 'java'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'com.google.guava:guava:20.0'
    testImplementation 'junit:junit:4.11'
}
```



- Proiectul B: build.gradle

```
plugins{
    id 'java'
    id 'application'
}
repositories {
    mavenCentral()
}
application{
    mainClass='StartApp'
}
dependencies {
    testImplementation 'junit:junit:4.11'
    implementation project(':A')
}
```



- Project Root: build.gradle

```
allprojects {  
    plugins{  
        id 'java'  
    }  
    repositories {  
        mavenCentral()  
    }  
    dependencies {  
        testImplementation 'junit:junit:4.11'  
    }  
}
```



# Gradle Build Tool

- Proiectul A: build.gradle modificat

```
dependencies {  
    implementation 'com.google.guava:guava:20.0'  
}
```

- Proiectul B: build.gradle modificat

```
plugins{  
    id 'application'  
}  
  
application{  
    mainClass='StartApp'  
}  
  
dependencies {  
    implementation project(':A')  
}
```



- Proiectul Root: build.gradle

```
subprojects {  
    //Configurări comune tuturor subproiectelor  
}  
  
project(':A') {  
    //Configurări specifice proiectului A  
}  
  
project(':B') {  
    //Configurări specifice proiectului B  
}
```

# Instrumente pentru jurnalizare

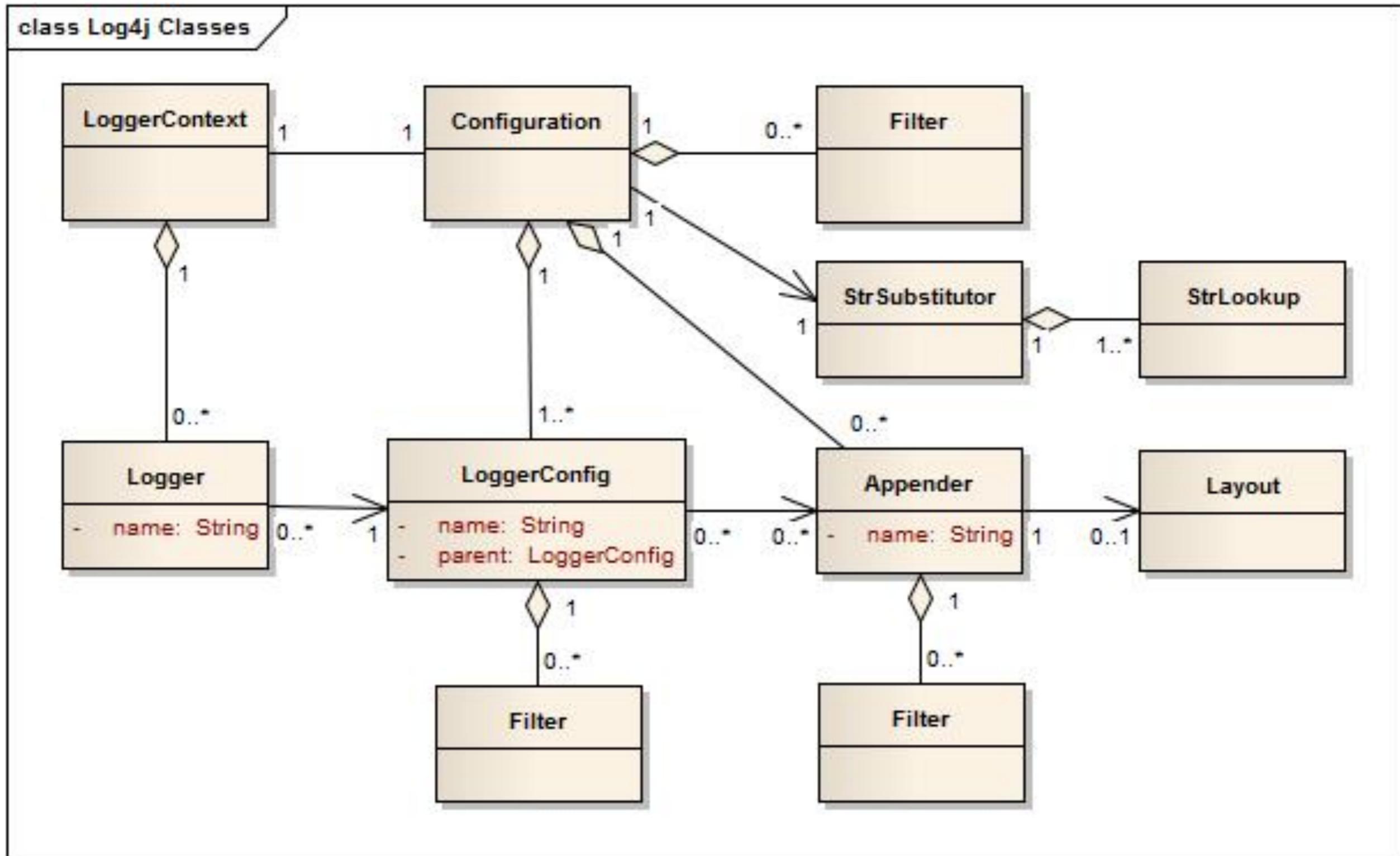
- Un **instrument pentru jurnalizare** permite programatorilor să înregistreze diferite tipuri de mesaje din codul sursă cu diverse scopuri: depanare, analiza ulterioară, etc.
- Majoritatea instrumentelor definesc diferite nivele pentru mesaje: *debug*, *warning*, *error*, *information*, *sever*, etc.
- Configurarea instrumentelor se face folosind fișiere text de configurare și pot fi oprite sau pornite la rulare.
- Apache Log4j, Logging SDK, slf4j, etc.



- Proiect open source dezvoltat de Apache Foundation.  
<http://logging.apache.org/log4j/2.0/>
- Log4j 2 are 3 componente principale:
  - *loggers*,
  - *appenders* (pentru stocare),
  - *layouts* (pentru formatare).



# Arhitectura





- Aplicațiile care folosesc Log4j 2 cer o referință către un obiect de tip *Logger* cu un anumit nume de la *LogManager*.
- *LogManager* va localiza obiectul *LoggerContext* corespunzător numelui și va obține referința către obiectul *Logger* de la el.
- Dacă obiectul de tip *Logger* corespunzător încă nu a fost creat, se va crea unul nou și va fi asociat cu un obiect de tip *LoggerConfig* care fie:
  - are același nume ca și *Logger*,
  - are același nume ca și pachetul părinte,
  - este rădăcina *LoggerConfig*.
- Obiectele de tip *LoggerConfig* sunt create folosind declarațiile din fișierul de configurare.
- Fiecare *LoggerConfig* îi sunt asociate unul sau mai multe obiecte de tip *Appender*.



- Fișierul de configurare (XML, JSON, proprietăți Java, yaml)
- Dacă nu este configurat, log4j 2 afișează doar mesajele de tip *error* la consolă

Exemplu fișier de configurare în format XML: *log4j2.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="TRACE">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="TRACE">
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```



- Log level - fiecare mesaj are asociat un anumit nivel
    - TRACE, DEBUG, INFO, WARN, ERROR și FATAL



- Numele asociat unui logger: structura ierarhică asemănătoare structurii pachetelor Java.

```
public class LogTest {  
    //a  
    private static final Logger logger = LogManager.getLogger(LogTest.class);  
    //b  
    private static final Logger logger =  
        LogManager.getLogger(LogTest.class.getName());  
    //c  
    private static final Logger logger = LogManager.getLogger();  
}
```



- Clasa `Logger` conține metode ce permit urmărirea fluxului execuției unei aplicații.
  - `entry(...)` - 0 ..4 parametrii
  - `traceEntry(String, ...)` - String și o lista variabilă de parametri
  - `exit(...), traceExit(String, ...)`
  - `throwing (...)` - când se aruncă o excepție
  - `catching(...)` când se prinde o excepție
  - `trace(...)`
  - `error(...)`
  - `log(...)`
  - etc.
- Exemplu



- Salvarea mesajelor: fișier, consolă, baze de date, etc.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="TRACE">
    <Appenders>
        <File name="FisierLog" fileName="logs/app.log">
            <PatternLayout pattern="%d{DATE} [%t] %class{36} %L %M - %msg%xEx%n"/>
        </File>
    </Appenders>
    <Loggers>
        <Root level="TRACE">
            <AppenderRef ref="FisierLog"/>
        </Root>
    </Loggers>
</Configuration>
```

# SGBD

- Sqlite
  - SQLiteStudio
- MySQL/MariaDB



- <https://www.sqlite.org/>
- Bază de date relațională
- Nu necesită configurări adiționale
- Nu necesită pornirea unui proces separat
- Toate informațiile sunt păstrate într-un singur fișier
- Formatul fișierului este independent de platformă
- Open source, gratuit.

**sqlite\_dir> sqlite3**



- <https://sqlitestudio.pl/>
- Sistem de gestiune a unei baze de date Sqlite
- Interfață grafică ușor de folosit
- Independent de platformă
- Gratuit
- Open source



- <https://www.mysql.com/> <https://mariadb.com/>
- Sistem de gestiune a bazelor de date relationale
- Rapid, scalabil, ușor de folosit
- Sistem de tip client-server/ embedded
- Gratuit
- Open source (MariaDb)

# JDBC

- Java Database Connectivity (JDBC) API conține o mulțime de clase ce asigură accesul la date.
- Se pot accesa orice surse de date: baze de date relationale, foi de calcul (*spreadsheets*) sau fișiere.
- JDBC oferă și o serie de interfețe ce pot fi folosite pentru construirea instrumentelor specializate.
- Pachete:
  - `java.sql` conține clase și interfețe pentru accesarea și procesarea datelor stocate într-o sursă de date (de obicei bază de date relațională).
  - `javax.sql` - adaugă noi funcționalități pentru partea de server.

# Stabilirea unei conexiuni

- Conectarea se poate face în două moduri:
  - Clasa **DriverManager**: Conexiunea se creează folosind un URL specific.
    - Necesita încărcarea unui driver specific bazei de date (JDBC<4).
    - Începând cu JDBC 4.0 nu mai este necesară încărcarea driverului.
  - Interfața **DataSource**: Este recomandată folosirea interfeței pentru aplicații complexe, deoarece permite configurarea sursei de date într-un mod transparent.
- Stabilirea unei conexiuni se realizează astfel:
  - Încărcarea driverului (versiuni JDBC <4.0)

**Class.forName(<DriverClassName>);**

- **Class.forName** creează automat o instanță a driverului și o înregistrează la **DriverManager**.
  - Nu este necesară crearea unei instanțe a clasei.
- Crearea conexiunii.

# Crearea unei conexiuni

- Folosind clasa **DriverManager** :
  - Colaborează cu interfața Driver pentru gestiunea driverelor disponibile unui client JDBC.
  - Când clientul cere o conexiune și furnizeaza un URL, clasa DriverManager este responsabilă cu găsirea driverului care recunoaște URL și cu folosirea lui pentru a se conecta la sursa de date.
  - Sintaxa URL-ului corespunzator unei conexiuni este:

**jdbc:subprotocol:<numeBazaDate>[listaProprietati]**

```
Connection conn = DriverManager.getConnection("jdbc:sqlite:users.db");

String url = "jdbc:mysql:Test";

String url = "jdbc:mariadb://localhost:3306/Test"

Connection conn = DriverManager.getConnection(url, <user>, <passwd>);
```

# Crearea unei conexiuni

- Folosind interfața **DataSource**:

```
InitialContext ic = new InitialContext()  
//a)  
DataSource ds = ic.lookup("java:comp/env/jdbc/myDB") ;  
Connection con = ds.getConnection() ;  
//b)  
  
DataSource ds =  
    (DataSource)org.apache.derby.jdbc.ClientDataSource()  
ds.setPort(1527) ;  
ds.setHost("localhost") ;  
ds.setUser("APP")  
ds.setPassword("APP") ;  
  
Connection con = ds.getConnection() ;
```

# Clasa Connection

- Reprezintă o sesiune cu o bază de date specifică.
- Orice instrucțiune SQL este executată și rezultatele sunt transmise folosind contextul unei conexiuni.
- Metode:
  - `close()`, `isClosed() :boolean`
  - `createStatement(...)` :`Statement` //overloaded
  - `prepareCall(...)` :`CallableStatement` //overloaded
  - `prepareStatement(...)` :`PreparedStatement` //overloaded
  - `rollback()`
  - `setAutoCommit(boolean)` //tranzactii
  - `getAutoCommit()` :`boolean`
  - `commit()`

# Clasa Statement

- Se folosește pentru executarea unei instrucțiuni SQL și pentru transmiterea rezultatului.
- Metode:
  - `execute(sql:String, ...):boolean` //pentru orice instructiune SQL
    - `getResultSet():ResultSet`
    - `getUpdateCount():int`
  - `executeQuery(sql:String, ...):ResultSet` //pentru SELECT
  - `executeUpdate(sql:String, ...):int` //INSERT,UPDATE,DELETE
    - `cancel()`
    - `close()`

# Exemplu Statement – Structura bazei de date



	Field Name	Data Type
PK	ID	AutoNumber
	title	Text
	authors	Text
	isbn	Text
	year	Number

# Statement exemplu

```
//Coneectarea la o baza de date SQLite
//Class.forName("org.sqlite.JDBC");
Connection conn=DriverManager.getConnection("jdbc:sqlite:/Users/teste/database/
    books.db");
//select
try(Statement stmt=conn.createStatement()){
    try(ResultSet rs=stmt.executeQuery("select * from books")){
        ...
    }
} catch(SQLException ex) {
    System.err.println(ex.getSQLState());
    System.err.println(ex.getErrorCode())
    System.err.println(ex.getMessage());
}

//update
String upString="update books set isbn='tj234' where isbn='tj237' "
try(Statement stmt=conn.createStatement()){
    stmt.executeUpdate(upString);
} catch(SQLException ex) {...}
```

# Statement exemplu

```
//insert
String insert="insert into books (title, authors, isbn, year) values
    ('Nuvele', 'Mihai Eminescu', '4567567', 2008)";
try(Statement stmt=conn.createStatement()) {
    stmt.executeUpdate(insert);
} catch (SQLException e) {
    System.out.println("Insert error "+e);
}

//delete
String delString="delete from books where isbn='tj234'"
try(Statement stmt=conn.createStatement()) {
    stmt.executeUpdate(delString);
} catch (SQLException ex) {
    //...
}
```

# ResultSet

- Conține o tabelă ce reprezintă rezultatul unei instrucțiuni SELECT.
- Un obiect de tip ResultSet conține un cursor care indică linia curentă din tabela.
- La început cursorul este poziționat înaintea primei linii din tabela.
- Metoda **next** mută cursorul pe următoarea linie din tabela. Rezultatul returnat este **false**, dacă nu mai există linii neparcuse în obiectul ResultSet.
- Metoda **next** se folosește pentru a parurge toate liniile din tabela.
- Se pot configura anumite proprietăți (daca tabela poate fi modificată, modul de parcursere, etc.).
- Configurarea se face în momentul apelului metodei de tip **createStatement(...)** :

```
Statement stmt = con.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE);

ResultSet rs = stmt.executeQuery("SELECT name, address FROM users");
// rs poate fi iterat, nu va fi notificat de modificari facute
// de alti utilizatori ai BD, si poate fi actualizat.
```

# ResultSet

- Metode:
  - `absolute(row:int)`
  - `relative(n:int)`
  - `afterLast()` , `beforeFirst()` , `first()` , `last()` , `next():boolean`
  - `getRow():int`
  - `getInt(columnIndex|columnLabel):int`
  - `getFloat(...)` , `getString(...)` , `getObject(...)` , etc
  - `updateInt(columnIndex|columnLabel, newValue)`
  - `updateFloat(...)` , `updateString(...)` , etc
  - `updateRow()`
  - `refreshRow()`
  - `rowDeleted()` , `rowInserted()` , `rowUpdated()`

# ResultSet

- Implicit un obiect de tip **ResultSet** este unidirecțional, cu parcurgerea înainte și nu poate fi modificat (actualizat).

```
try(Statement stmt=conn.createStatement()) {  
    try(ResultSet rs=stmt.executeQuery("select * from books")) {  
        while(rs.next()) {  
            System.out.println("Book "+rs.getString("title")  
                +' '+rs.getString("author")+' '+rs.getInt("year"));  
        }  
    }  
} catch(SQLException ex) {  
    // ...  
}
```

# Clasa PreparedStatement

- Unui obiect de tip **PreparedStatement** î se transmite instrucțiunea SQL în momentul creării.
- Instrucțiunea SQL este transmisa sistemului de gestiune a bazei de date (SGBD), unde este compilată.
- Când se execută instrucțiunea asociată unui **PreparedStatement**, SGBD execută direct instrucțiunea SQL fără a o reverifica în prealabil.
- Este mai eficientă decât Statement.
- Poate să aibă parametrii. Aceştia sunt marcați folosind ‘?’.

```
PreparedStatement preStmt = con.prepareStatement(  
    "select * from books WHERE year=?") ;
```

- Valoarea unui parametru este transmisa folosind metodele de tip **setxyz**, unde **xyz** reprezinta tipul parametrului.
- Pozițiile parametrilor încep de la 1.

```
preStmt.setInt(1, 2008) ;  
ResultSet rs=preStmt.executeQuery() ;
```

# Tranzacții

- Implicit, fiecare instrucțiune SQL este tratată ca și o tranzacție și este înregistrată/operată imediat după execuție.
- Comportamentul implicit poate fi modificat folosind metoda `setAutoCommit(false)` din clasa `Connection`.
- Metode:
  - `commit`
  - `rollback`
  - `setSavePoint`

# Tranzacții - exemplu

```
con.setAutoCommit(false);

PreparedStatement updateSales = con.prepareStatement(
    "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");

updateSales.setInt(1, 50);
updateSales.setString(2, "Black");
updateSales.executeUpdate();

PreparedStatement updateTotal = con.prepareStatement(
    "UPDATE COFFEES SET TOTAL = TOTAL + ? WHERE COF_NAME
    LIKE ?");

updateTotal.setInt(1, 50);
updateTotal.setString(2, "Black");
updateTotal.executeUpdate();

con.commit();

con.setAutoCommit(true);
```

# Proceduri stocate

- O procedură stocată este un grup de instrucțiuni SQL care formează o unitate logică și îndeplinește o anumită sarcină.
- Ele sunt folosite pentru a îngloba o serie de operațiuni sau interogări ce trebuie executate pe un server de baze de date.
- De exemplu, operațiunile de pe o bază de date angajat (angajarea, concedierea, promovarea, cautarea) ar putea fi codificate ca proceduri stocate executate în funcție de codul cerere.
- Procedurile stocate pot fi compilate și executate cu diferiți parametri și pot avea orice combinație de intrare, ieșire sau intrare/ieșire.

# CallableStatement

- Este folosită pentru executarea procedurilor stocate.
- Tehnologia JDBC API furnizează o sintaxă de apelare a procedurilor stocate independentă de SGBD folosit.
- Sintaxa folosită are două variante:
  - conține un parametru de tip rezultat
  - nu conține un parametru de tip rezultat.
- Dacă se folosește prima variantă, parametrul de tip rezultat trebuie să fie înregistrat ca și parametru de tip OUT. Ceilalți parametrii pot fi folosiți pentru intrare, ieșire sau ambele.
- Parametrii sunt referiți secvențial, folosind numere, primul parametru fiind pe poziția 1.

```
{?= call <procedure-name>[(<arg1>,<arg2>, ...) ] }  
{call <procedure-name>[(<arg1>,<arg2>, ...) ] }
```

```
CallableStatement cs = con.prepareStatement("{call SHOW_SUPPLIERS}");  
ResultSet rs = cs.executeQuery();
```

# Properties

- Clasa **Properties** (pachetul `java.util`) se folosește pentru a păstra perechi cheie-valoare. Perechile pot fi citite sau salvate dintr-un/într-un flux de date (ex. fișier). Cheia și valoarea sunt de tip String, cheia fiind unică.

```
//exemplu.properties
tasksFile=tasks.txt
inputDir=input
outputDir=output

//Citirea fișierului cu proprietăți
Properties props=new Properties();
try {
    props.load(new FileInputStream("exemplu.properties"));
} catch (IOException e) {
    System.out.println("Eroare: "+e);
}
```

# Properties

- Metode:
  - `getProperty(cheie:String) :String`
  - `setProperty(c:String, v:String) :Object`
  - `list(PrintWriter)`
  - `load(Reader)`
  - `store(w:Writer, comentarii:String)`

```
Properties props=new Properties();
try {
    props.load(new FileInputStream("exemplu.properties"));
} catch (IOException e) {
    System.out.println("Eroare: "+e);
}
String tasksFile=props.getProperty("tasksFile");
if (tasksFile==null) //proprietaea nu a fost gasita in fisier
    System.out.println("fisier incorrect");
...
```

# System +Properties

- Metode din clasa System:

- `setProperties(Properties)`
- `setProperty(c:String, v:String) :String`
- `getProperty(String) :String`
- ...

```
Properties serverProps=new Properties(System.getProperties());
try {
    serverProps.load(new FileReader("exemplu.properties"));
    System.setProperties(serverProps);
    System.getProperties().list(System.out);
} catch (IOException e) {
    System.out.println("Eroare "+e);
}
String tasksFile=System.getProperty("tasksFile");
```

# Exemplu configurare BD

```
//Fisierul bd.properties sau bd.config
jdbc.url=jdbc:mysql://localhost/mpp
jdbc.user=test
jdbc.pass=test

//cod
Connection getNewConnection() {
    String url=System.getProperty("jdbc.url");
    String user=System.getProperty("jdbc.user");
    String pass=System.getProperty("jdbc.pass");
    Connection con=null;
    try {
        con= DriverManager.getConnection(url,user,pass);
    } catch (SQLException e) {
        System.out.println("Eroare stabilire conexiune "+e);
    }
    return con;
}
```

# Dependențe driver JDBC - Gradle

```
//Fisierul build.gradle
dependencies {

    testImplementation group: 'org.junit.jupiter', name: 'junit-jupiter-api', version:
        '5.8.2'

    testImplementation group: 'org.junit.jupiter', name: 'junit-jupiter-engine',
        version:'5.8.2'

    //jurnalizare

    implementation group: 'org.apache.logging.log4j', name: 'log4j-core', version:
        '2.18.0'

    implementation group: 'org.apache.logging.log4j', name: 'log4j-api', version:
        '2.18.0'

    //drivere conectare la baza de date

    runtimeOnly group: 'org.xerial', name: 'sqlite-jdbc', version: '3.36.0.3'

    runtimeOnly 'org.mariadb.jdbc:mariadb-java-client:2.1.2'
    runtimeOnly 'mysql:mysql-connector-java:5.1.20'

}

//Maven repository
//https://mvnrepository.com/
```

# Medii de proiectare și programare

2022-2023

Curs 3

# Conținut curs 3

- Configurarea (Gradle)
- Ierarhia repository
- ADO.NET
  - Configurarea(C# app.config)
- Adnotări
- Java Beans

# Exemplu configurare BD

```
//Fisierul bd.properties sau bd.config
jdbc.url=jdbc:mysql://localhost/mpp
jdbc.user=test
jdbc.pass=test

//cod
Connection getConnection() {
    String url=System.getProperty("jdbc.url");
    String user=System.getProperty("jdbc.user");
    String pass=System.getProperty("jdbc.pass");
    Connection con=null;
    try {
        con= DriverManager.getConnection(url,user,pass);
    } catch (SQLException e) {
        System.out.println("Eroare stabilire conexiune "+e);
    }
    return con;
}
```

# Dependențe driver JDBC - Gradle

```
//Fisierul build.gradle
dependencies {

    testImplementation group: 'org.junit.jupiter', name: 'junit-jupiter-api', version:
        '5.8.2'

    testImplementation group: 'org.junit.jupiter', name: 'junit-jupiter-engine',
        version:'5.8.2'

    //jurnalizare

    implementation group: 'org.apache.logging.log4j', name: 'log4j-core', version:
        '2.14.0'

    implementation group: 'org.apache.logging.log4j', name: 'log4j-api', version:
        '2.14.0'

    //drivere conectare la baza de date

    runtimeOnly group: 'org.xerial', name: 'sqlite-jdbc', version: '3.36.0.3'

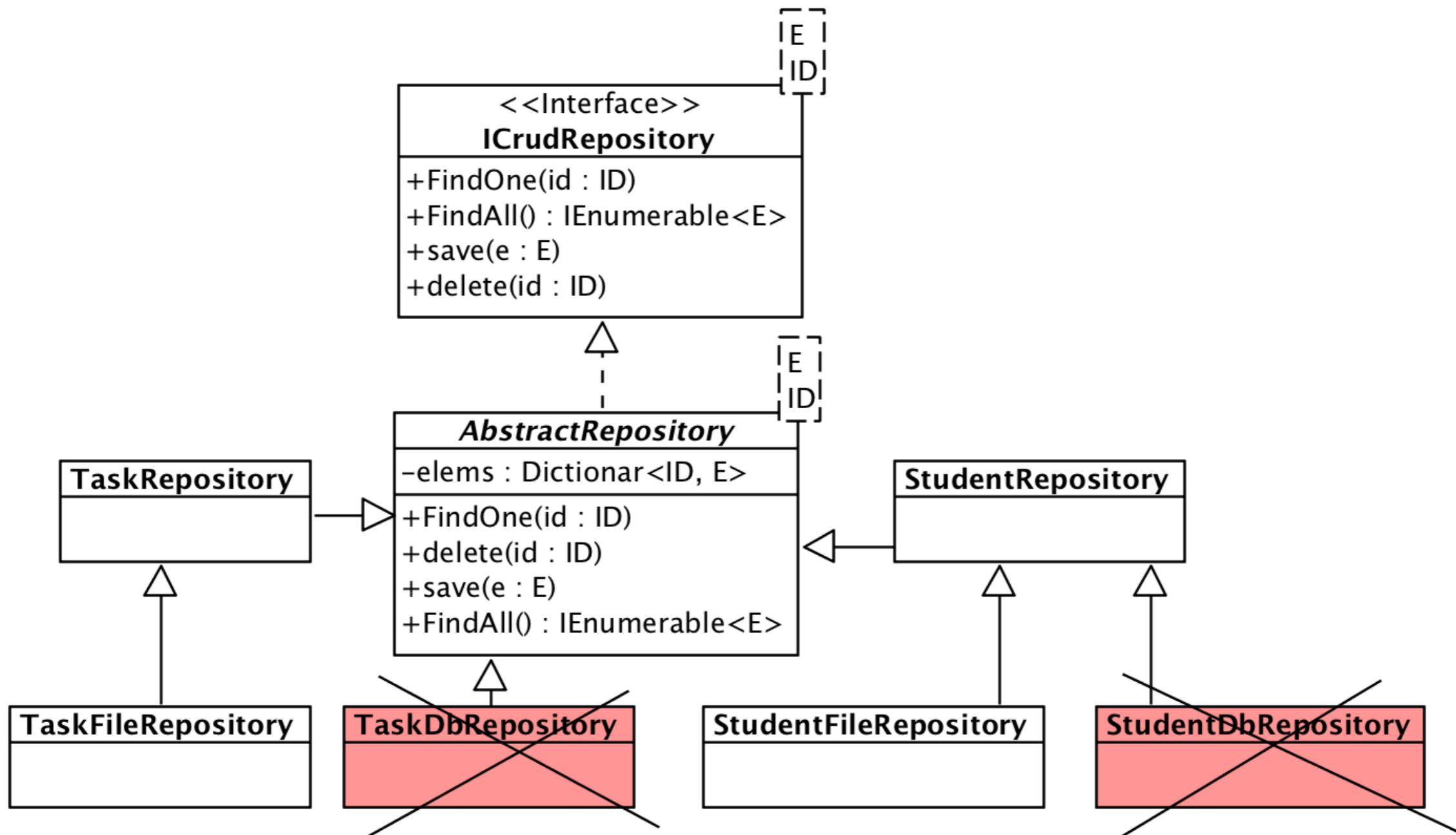
    runtimeOnly 'org.mariadb.jdbc:mariadb-java-client:2.1.2'
    runtimeOnly 'mysql:mysql-connector-java:5.1.20'

}

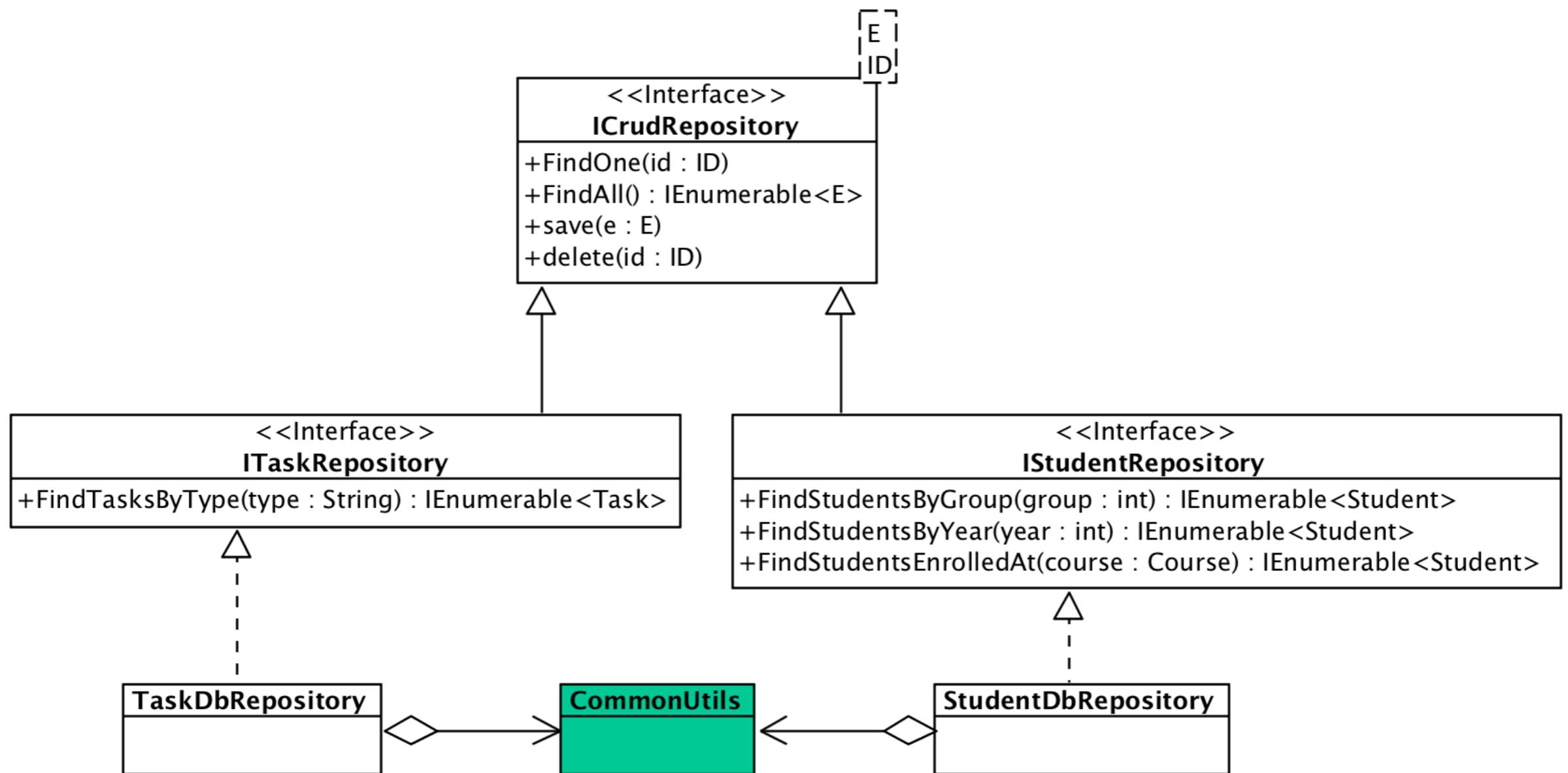
//Maven repository
//https://mvnrepository.com/
```

# Exemplu Java

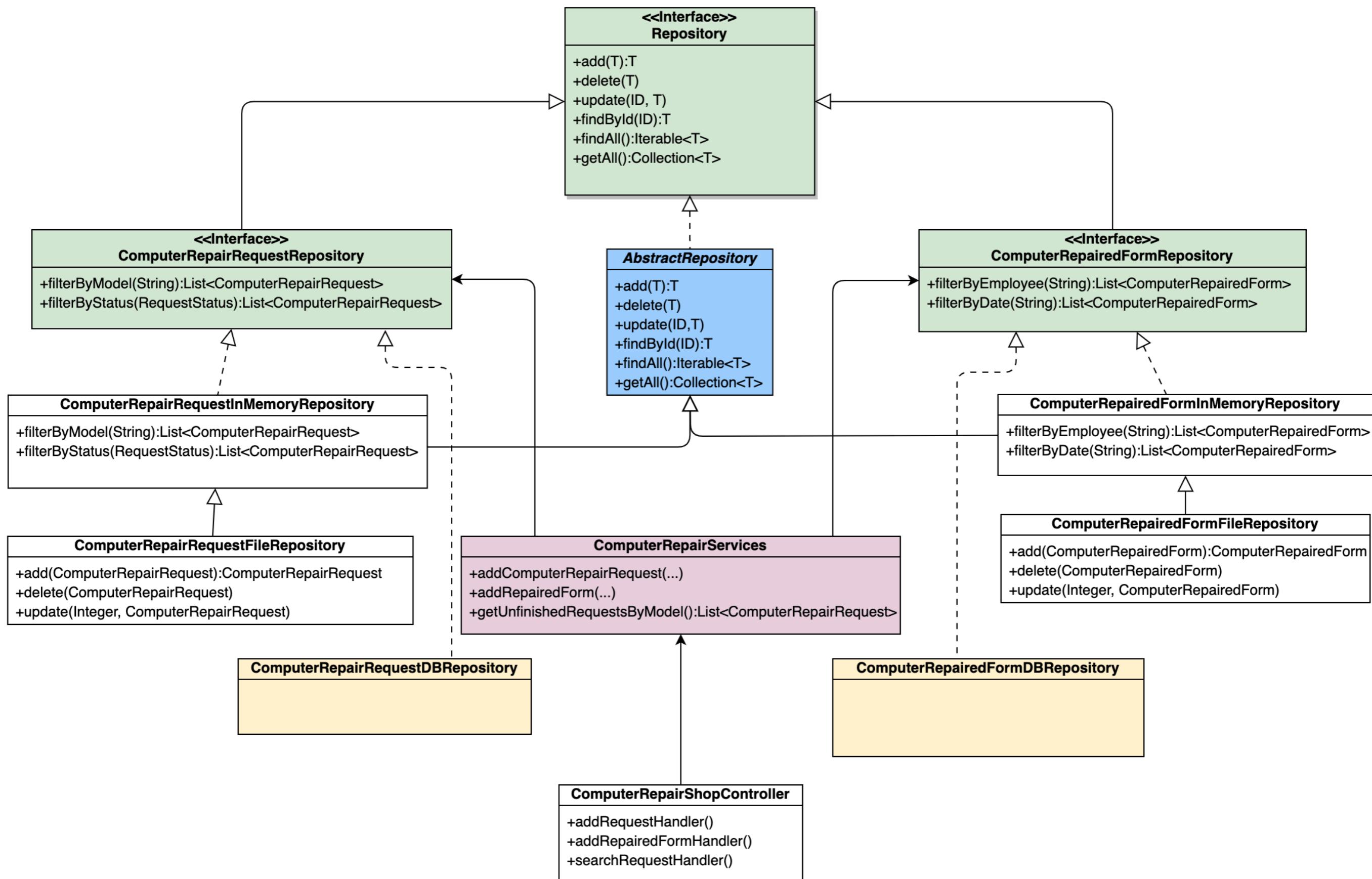
# Hierarchie repositories (1)



# Hierarchie repositories (2)



# Hierarchie repositories (3)



# ADO.NET

- ADO.NET este o bibliotecă orientată pe obiecte care permite unei aplicații să interacționeze cu diferite surse de date:
  - baze de date relaționale
  - fișiere text
  - fișiere Excel
  - fișiere XML
- Conține 4 spații de nume pentru interacțiunea cu 4 tipuri de baze de date:
  - SQL Server
  - Oracle
  - Surse ODBC
  - OLEDB.

# ADO.NET

## Spații de nume

- **System.Data**—Toate clasele generice pentru accesarea datelor.
- **System.Data.Common**—Clase comune sau redefinite de furnizori de date specifi.
- **System.Data.Odbc**—Clasele pentru ODBC
- **System.Data.OleDb**—Clasele pentru OLE DB
- **System.Data.Oracle**—Clasele pentru Oracle
- **System.Data.SqlClient**—Clasele pentru SQL Server
- **System.Data.SqlTypes**—Tipurile de date SQL Server

# System.Data

- Conține clasele și interfețele folosite indiferent de sistemul de gestiune a bazelor de date.
- **DataSet**— Clasa pentru lucru offline. Poate conține o mulțime de **DataTables** și relații între acestea.
- **DataTable**—Un container ce conține una sau mai multe coloane. Când este populat va avea una sau mai multe **DataRow**s conținând informația.
- **DataRow**—O mulțime de valori corespunzând unei linii dintr-o tabelă dintr-o bază de date relațională, sau unei linii dintr-o foaie de calcul.
- **DataColumn**—Conține definiția unei coloane dintr-o tabelă: numele și tipul.
- **DataRelation**—Reprezintă o relație între două tabele dintr-un **DataSet**. Se folosește pentru a reprezenta relația "cheie străină".
- **Constraint**—Definește constrângeri pentru una sau mai multe  **DataColumn** (ex. valori unice).

# System.Data.Common

- **DataColumnMapping**—Mapează numele unei coloane dintr-o tabelă din baza de date cu numele unei coloane dintr-un **DataTable**.
- **DataTableMapping**—Mapează numele unei tabele dintr-o bază de date cu un **DataTable** dintr-un **DataSet**.
- **DbCommandBuilder**—Genereaza automat comenzi pentru a sincroniza modificările dintr-un **DataSet** cu baza de date asociată.

# ADO.NET API

- ADO.NET conține clase specifice interacțiunii cu anumite tipuri de baze de date.
- Aceste clase implementează o mulțime de interfețe standard din spațiul de nume System.Data, permitând claselor să fie folosite într-o manieră generică, dacă este necesar.
  - **IDbConnection** folosită pentru conectarea la o baza de date.
  - **IDataAdapter** folosită pentru păstrarea instrucțiunilor *select*, *insert*, *update* și *delete* care sunt apoi folosite pentru popularea unui **DataSet** și pentru actualizarea bazei de date.
  - **IDataReader**: folosit ca și un cititor de date, forward-only.
  - **IDbCommand**: folosit ca și wrapper pentru instrucțiuni SQL sau apeluri de proceduri stocate.
  - **IDbParameter**: reprezintă un parametru pentru un obiect de tip Command.
  - **IDbTransaction**: folosit pentru reprezentarea unei tranzacții ca și un obiect.

# IDbConnection

- Reprezintă o conexiune deschisă către o sursă de date:
  - `SqlConnection`, `OleDbConnection`, `OracleConnection`, `ODBCConnection`
  - `MySqlConnection`, `SQLiteConnection (Windows)`,
  - `SqliteConnection (Mono)`
- Metode:
  - `BeginTransaction`
  - `ChangeDatabase`
  - `Open`
  - `Close`
  - `CreateCommand`
- Proprietăți:
  - `ConnectionString`, `ConnectionTimeout`, `Database`, `State`

# IDbConnection

- Conectarea la Sql Server

```
var conn = new SqlConnection(  
    "Data Source=(local);Initial Catalog=Northwind;User Id=test;  
    Password=test");
```

- Conectarea la o bază de date Access folosind OleDb

```
String connectionString="Provider=Microsoft.Jet.OLEDB.4.0;Data Source=books.mdb";  
var conn=new OleDbConnection(connectionString);
```

- Conectarea la MySql:

```
String connectionString = "Database=mpp;Data Source=localhost;User id=test;"  
+ "Password=passTest;"  
var conn= new MySqlConnection(connectionString);
```

- Conectarea la Sqlite (folosind Mono.Sqlite - specific Linux/MacOS):

```
String connectionString = "URI=file:/Users/test/database/tasks.db,Version=3";  
var conn= new SqliteConnection(connectionString);
```

# IDbCommand

- Reprezintă o instrucțiune SQL executată când există o conexiune către sursa de date.
  - `SqlCommand`, `OleDbCommand`, `OracleCommand`, `ODBCCCommand`
  - `MySqlCommand`, `SqliteCommand (Mono)`, `SQLiteCommand (Windows)`
- Metode:
  - `ExecuteReader`, `ExecuteNonQuery`, `ExecuteScalar`
  - `CreateParameter`
  - `Cancel`
- Proprietăți:
  - `CommandText`, `CommandTimeout`,  `CommandType`, `Connection`, `Parameters`, etc.
- `CommandType`:
  - `Text` (o comandă SQL), `StoredProcedure`, `TableDirect` (numele unei tabele, doar pentru furnizori OleDb).

# IDbCommand

- Text:

```
String select = "SELECT ContactName FROM Customers";
```

```
SqliteCommand cmd = new SqliteCommand(select , conn);
```

- Stored Procedure

```
MySqlCommand cmd = new MySqlCommand("CustOrderHist", conn);
```

```
cmd.CommandType = CommandType.StoredProcedure;
```

- Table Direct

```
OleDbCommand cmd = new OleDbCommand("Categories", conn);
```

```
cmd.CommandType = CommandType.TableDirect;
```

# IDbCommand

- ExecuteNonQuery:

```
string source =...;

string sqlCom = "UPDATE Customers SET ContactName = 'Bob' " +
                "WHERE ContactName = 'Bill'" ;

using(var conn = new OleDbConnection(source)) {

    conn.Open();

    var cmd = new OleDbCommand(sqlCom, conn);

    int rowsReturned = cmd.ExecuteNonQuery();

    Console.WriteLine("{0} rows affected.", rowsReturned);

}
```

# IDbCommand

- ExecuteReader:

```
string source = ...;

string select = "SELECT ContactName,CompanyName FROM Customers";

using(var conn = new MySqlConnection(source)) {

    conn.Open();

    var cmd = new MySqlCommand(select, conn);

    using(var reader = cmd.ExecuteReader()) {

        while(reader.Read()) {

            Console.WriteLine("Contact:{0} Company:{1}", reader[0] ,
                reader[1]);

        }
    }
}
```

# IDbCommand

- ExecuteScalar:

```
string source = ...;

string select = "SELECT COUNT(*) FROM Customers";

using(var conn = new SqliteConnection(source)) {

    conn.Open();

    using(var cmd = new SqlCommand(select, conn)) {

        object o = cmd.ExecuteScalar();

        Console.WriteLine ("Customers: {0}", o);

    }

}
```

# IDataReader

- Oferă posibilitatea citirii unui flux sau mai multor fluxuri secvențial (forward-only) obținute prin executarea unei comenzi asupra unei surse de date.
  - `SqlDataReader`, `OleDbDataReader`, `OracleDataReader`, `ODBCDataReader`
  - `MySqlDataReader`, `SqliteDataReader` (Mono), `SQLiteDataReader` (Windows)
- O instanță de tip `IDataReader` este obținută apelând metoda `IDbCommand.ExecuteReader`.
- Metode:
  - `Read`
  - `GetBoolean`, `GetByte`, `GetDouble`, `GetFloat`, `GetInt16`, `GetString`, etc.
  - `Close`
- Proprietăți:
  - `Item` (index sau nume), `IsClosed`

# IDataReader

```
string source = ...;

string selectCmd = "SELECT name,address FROM persons";

using(var conn = new SqliteConnection(source)) {

    conn.Open();

    using(var cmd = conn.CreateCommand()) {

        cmd.CommandText=selectCmd;

        using(var reader = cmd.ExecuteReader()) {

            while(reader.Read())

                Console.WriteLine("{0} {1}", reader["name"] , reader["address"]);

        }

    }

}
```

# IDataAdapter

- Reprezintă un set de proprietăți folosite pentru completarea unui DataSet și pentru actualizarea unei surse de date.
  - **SqlDataAdapter**, **OleDbDataAdapter**, **OracleDataAdapter**, **ODBCDataAdapter**
  - **MySqlDataAdapter**, **SqliteDataAdapter (Mono)** , **SQLiteDataAdapter (Windows)**
- Este folosit în asociere cu un DataSet.
- Un **DataSet** este un obiect în memorie care poate păstra mai multe tabele.
- **DataSets** păstrează doar informația, nu interacționează cu sursa de date.
- **IDataAdapter** gestionează conexiunile către sursa de date.
- **IDataAdapter** deschide o conexiune doar când este necesar și o închide imediat ce sarcina și-a încheiat execuția.

# IDataAdapter

- Execută următoarele când populează un DataSet cu date:
  - Deschide o conexiune la sursa de date
  - Obține și încarcă datele în **DataSet**
  - Închide conexiunea
- Execută următoarele când actualizează sursa de date cu modificările din DataSet:
  - Deschide conexiunea
  - Scrie modificările din DataSet în sursa de date.
  - Închide conexiunea
- Între populare și actualizare conexiunile către sursa de date sunt închise.
- Metode:
  - **Fill** (adaugă sau actualizează linii în DataSet potrivite cu cele din sursa de date),
  - **Update** (apeleză instrucțiunile INSERT, UPDATE, or DELETE corespunzătoare fiecărei inserări, actualizări sau ștergeri din DataSet)
- Proprietăți: **DeleteCommand**, **InsertCommand**, **SelectCommand**, **UpdateCommand**

# IDataAdapter

```
string source =...;

var Connection conn = new MySqlConnection(source);

string select = "SELECT * FROM books";

DataSet data=new DataSet();

var dataAdapter=new MySqlAdapter(select, conn);

dataAdapter.Fill(data, "Books");

DataRowCollection dra=data.Tables["Books"].Rows;

foreach(DataRow in dra)

Console.WriteLine(dr["isbn"]+dr["author"]+dr["title"]);
```

# IDataParameter

- Reprezintă parametrul unui obiect de tip **Command**.
  - **SqlParameter**, **OracleParameter**, **OleDbParameter**, **OdbcParameter**
  - **MySqlParameter**, **SqliteParameter (Mono)**, **SQLiteParameter (Windows)**
- Membrii
  - **Value**
  - **ParameterName**
  - **DbType**
- DbType:
  - **Boolean**, **Date**, **Double**, **Int32**, **String**, etc.

# IDataParameter

```
string source = ...;

string select = "SELECT * FROM Customers where city=@City";

using(var conn = new SqliteConnection(source)) {

    conn.Open();

    using(var cmd = new SqliteCommand(select, conn)) {

        var param = cmd.CreateParameter();

        param.ParameterName = "@City";

        param.Value = "ABC";

        cmd.Parameters.Add(param);

        using(var reader = cmd.ExecuteReader()) {

            while(reader.Read()) {

                Console.WriteLine("Contact:{0} Company:{1}" , reader["CompanyName"] ,
                    reader["ContactName"]);

            }
        }
    }
}
```

# app.config

- Fișier de configurare pentru aplicații .NET

```
<?xml version="1.0" encoding="utf-8"?>

<configuration>
    <connectionStrings>
        <add name="tasksDB"
            connectionString="URI=file:/Users/xyz/MPP/database/tasks.db,Version=3" />
        <!--
        <add name="tasksDB"
            connectionString="Database=mpp;Data Source=localhost;User id=test;Password=pass123;" />
        -->
    </connectionStrings>
</configuration>
```

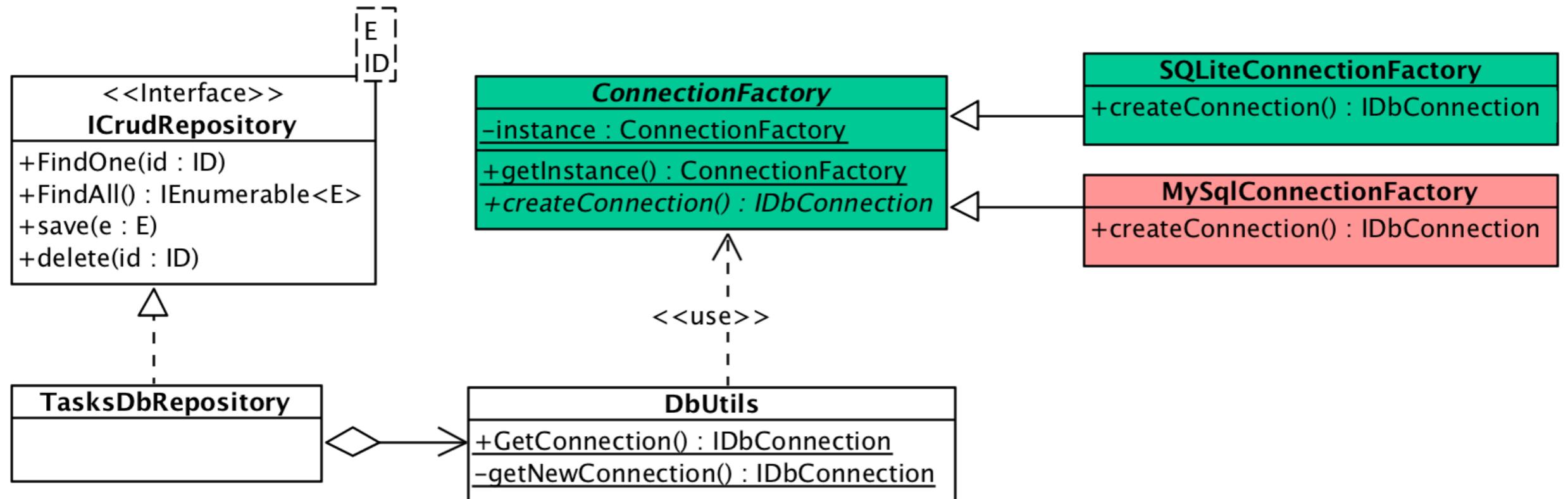
- La compilare fișierul este copiat în directul bin/debug cu numele NumeApp.exe.config (unde NumeApp este numele proiectului)

# app.config

- Obținerea datelor din app.config
  - Clasa  [ConfigurationManager](#) (spațiul de nume `System.Configuration`)

```
static string GetConnectionStringByName(string name) {  
    // Presupunem ca nu există.  
    string returnValue = null;  
  
    // Caută numele în secțiunea connectionStrings.  
    ConnectionStringSettings settings = ConfigurationManager.ConnectionStrings[name];  
  
    // Dacă este găsit, returnează valoarea asociată la connection string.  
    if (settings != null)  
        returnValue = settings.ConnectionString;  
  
    return returnValue;  
}
```

# Arhitectura C#



# Exemplu C#

# Adnotări (*Annotations*)

- Începând cu versiunea 1.5
- Adaugă informații unei părți de cod (clasă, metodă, pachet), dar **nu fac parte din program**. Adnotările **nu au nici un efect direct asupra codului pe care îl marchează**.
- Utilizări:
  - *A furniza informații suplimentare compilatorului.* Adnotările pot fi folosite de compilator pentru a detecta erori sau pentru a elimina atenționări.
  - *Procesare automata din timpul compilării sau deploymentului.* Instrumente soft specializate pot folosi adnotările pentru a genera automat cod, fișiere XML, etc.
  - *Procesare în timpul execuției.* Unele adnotări sunt disponibile pentru a fi examineate în timpul execuției codului.

# Definirea adnotărilor

```
[declaratii meta-adnotari]  
public @interface NumeAdnotare {  
    [declaratii elemente]  
}
```

Meta-adnotările (pachetul `java.lang.annotation`) (adnotări pentru adnotări) pot fi:

- `@Target(ElementType)` : specifică locul din codul sursă unde poate fi folosită adnotarea.
  - `CONSTRUCTOR`: declararea unui constructor
  - `FIELD`: declararea unui atribut (inclusiv constante enum)
  - `LOCAL_VARIABLE`: declararea unei variabile locale
  - `METHOD`: declararea unei metode
  - `PACKAGE`: declararea unui pachet
  - `PARAMETER`: declararea unui parametru
  - `TYPE`: declararea unei noi clase, interfețe, adnotări sau enum.

# Definirea adnotarilor

Meta-adnotările pot fi:

- **@Retention (RetentionPolicy)** : specifică cât timp va fi păstrată adnotarea:
  - **SOURCE**: Adnotările nu sunt salvate la compilare.
  - **CLASS**: Adnotările sunt disponibile în fișierul .class, dar pot fi eliminate de mașina virtuală.
  - **RUNTIME**: Adnotările sunt păstrate de mașina virtuală în timpul execuției și pot fi citite folosind reflecție.
- **@Documented**: adnotarea va fi inclusă în documentația Javadocs.
- **@Inherited**: Permit subclaszelor să moștenească adnotările părinților.

# Elementele unei adnotări

- Sintaxa:

```
Tip numeElement() [default valoare_implicita];
```

unde **Tip** poate fi:

- orice tip primitiv (**int**, **float**, **double**, **byte**, etc.)
- **String**
- **Class**
- Enumerări (**enum**)
- Adnotări (**annotation**)
- Tablouri de tipurile menționate mai sus.

Observații:

1. Dacă se folosește **alt tip la declararea unui element**, compilatorul va genera **eroare**.
2. Dacă o adnotare nu conține nici un element, adnotarea se numește de tip *marker*.

# Constrângeri valori implicite

- Exista **două constrângeri** pentru valoarea unui element:
  1. *Nici un element nu poate avea o valoare nespecificată* (fie se declară o valoare implicită, fie se atribuie o valoare pentru fiecare element în momentul folosirii adnotării).
  2. *Pentru elementele care nu sunt de tip primitiv, nu se acceptă valoarea **null*** (în momentul folosirii sau ca și valoare implicită).

# Adnotări - exemplu

```
import java.lang.annotation.*;

@Target(ElementType.CLASS)
@Retention(RetentionPolicy.RUNTIME)
public @interface ClassPreamble {
    String author();
    String date();
    int currentRevision() default 1;
    String lastModified() default "N/A";
    String lastModifiedBy() default "N/A";
    String[] reviewers();
}
```

# Folosirea adnotărilor

Adnotarea apare prima, de obicei pe linie proprie, și poate conține elemente.

Observații:

1. Dacă adnotarea conține un singur element numit **value**, numele acestuia poate fi omis.
2. Dacă adnotarea nu conține nici un element, parantezele pot fi omise.

```
@ClassPreamble (
    author = "Popescu Vasile",
    date = "3/17/2008",
    currentRevision = 4,
    lastModified = "4/11/2011",
    lastModifiedBy = "Ionescu Matei"
    reviewers = {"Vasilescu Ana", "Marinescu Ion", "Pop Ioana"}
)
public class A extends B{
//...
}
```

# Exemplu adnotări

Declararea:

```
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)

public @interface UseCase {

    public int id();

    public String description() default "no description";

}
```

Folosirea

```
public class A{

    @UseCase(id=3, description="abcd")

    public void f(){

}

}
```

# Adnotări standard

- JSE conține 3 adnotări standard:
  - **@Override** pentru a indica că o metoda redefineste o metodă din clasa de baza.  
Dacă numele metodei sau signatura nu sunt corecte, compilatorul va genera o eroare.

```
class A{  
    @Override  
    public String toString() { ... }  
}
```

- **@Deprecated** pentru a genera o atenționare la compilare când se folosește elementul adnotat (clasă, metodă, etc.)
- **@SuppressWarnings** spune compilatorului să nu furnizeze anumite atenționări:  
**unchecked, deprecated**

```
@SuppressWarnings("unchecked", "deprecated")  
void metodaA() { }
```

# Medii de proiectare și programare

2022-2023

Curs 4

# Continut curs 4

- Java Beans
- Introducere in Spring

# Beans

- Orice clasă Java este un **POJO** (eng. *Plain Old Java Object*).
- JavaBeans: este o clasă Java specială. Reguli:
  - *Trebuie să aibă un constructor implicit (public și fără nici un parametru)*. Alte instrumente specializate vor folosi acest constructor pentru a instanția un obiect.
  - *Atributele trebuie să poată fi accesate folosind metode de tip `getXYZ`, `setXYZ` și `isXYZ`* (pentru atrbute de tip boolean). Atributele pentru care sunt definite aceste metode se numesc **proprietăți**, numele proprietății fiind **xyz**. Când se modifică sau se dorește valoarea unei proprietăți se apelează una dintre metodele corespunzătoare.
  - *Clasa trebuie sa fie serializabilă*. Acest lucru permite instrumentelor specializate să salveze și să refacă starea unui JavaBean.
  - Exemplu: **Componentele GUI**
- Enterprise Java Beans (EJBs): pentru aplicații complexe (tranzacții, securitate, acces la baze de date)

# Exemplu Java Beans

```
public class Student implements java.io.Serializable {  
    private String nume;  
    private int grupa;  
    private boolean licentiat;  
    private int note[];  
    public Student() { }  
    public Student(String nume, int grupa, boolean licentiat){...}  
    public String getName() { return nume; }  
    public void setName(String name) { nume = name; }  
    public int getGrupa() {return grupa;}  
    public void setGrupa(int g){grupa=g;}  
    public void setLicentiat(boolean l){licentiat=l;}  
    public boolean isLicentiat(){ return licentiat; }  
    public void setNote(int[] n){ note=n; }  
    public int[] getNote(){return note;}  
}
```

# Introducere În Spring - Motivație

- Orice aplicație medie sau complexă este compusă dintr-o mulțime de obiecte care colaborează pentru atingerea unui scop. Aceste obiecte știu despre celelalte obiecte (asocierile) și comunică prin transmiterea de mesaje.
- Abordarea tradițională pentru crearea asocierilor dintre obiecte (prin instantiere sau căutare) generează cod complicat care este dificil de reutilizat și testat (folosind unit testing).

//varianta 1

```
class ConcursService{  
    private ParticipantRepositoryMock repo;  
    public ConcursService(){  
        repo=new ParticipantRepositoryMock();  
    }  
    //...  
}
```

# Introducere în Spring

//varianta 2

```
class ConcursService{  
    private ParticipantiRepositoryFile repo;  
    public ConcursService(){  
        repo=new ParticipantiRepositoryFile("Participanti.txt");  
    }  
}
```

//varianta 2a

```
public ConcursService(){  
    repo=new ParticipantiRepositoryFile("Participanti2.txt",  
                                         new ParticipantValidator());  
}
```

//varianta 3

```
class ConcursService{  
    private ParticipantiRepositoryJdbc repo;  
    public ConcursService(){  
        Properties props=...  
        repo=new ParticipantiRepositoryJdbc(props);  
    }  
}
```

# Introducere în Spring

- Spring este un framework open-source, creat inițial de Rod Johnson și descris în cartea sa, *Expert One-on-One: J2EE Design and Development*.
- Frameworkul Spring a fost creat pentru a facilita dezvoltarea aplicațiilor complexe și foarte mari.
- În *Spring se pot folosi obiecte simple Java (POJO)*, pentru a crea aplicații care anterior erau posibile doar folosind EJB.
- Un **bean Spring** este orice clasă Java (**nu respectă regulile Java Beans**).
- *Spring promovează cuplarea slabă prin “injectarea” asocierilor și folosirea interfețelor.*
- Spring folosește **principiul IoC** pentru “injectarea” asocierilor/dependențelor.

# Introducere în Spring

```
public interface ParticipantiRepository{...}
class ConcursService{
    private ParticipantiRepository repo;
    public ConcursService(ParticipantiRepository r) {
        repo=r;
    }
    ...
}
//sau
class ConcursService{
    private ParticipantiRepository repo;
    public ConcursService(){... }
    public void setParticipanti(ParticipantiRepository r){repo=r;}
}
public class ParticipantiRepositoryFile implements
    ParticipantiRepository{...}
public class ParticipantiRepositoryJdbc implements
    ParticipantiRepository{...}
```

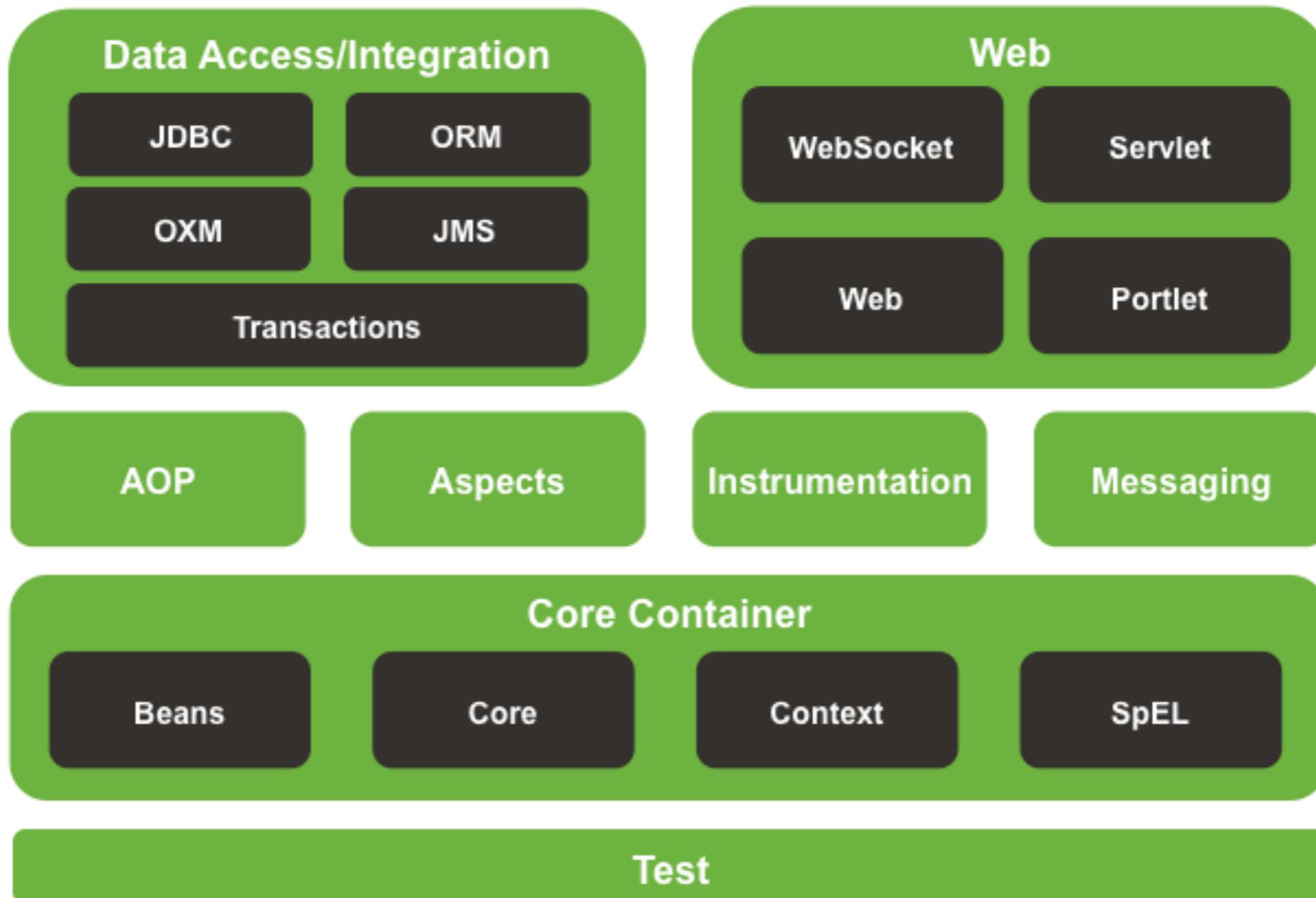
# *IoC, Dependency Injection*

- Principiul *Inversion of Control* (IoC) este cunoscut și ca *dependency injection* (DI).
- DI este procesul prin care obiectele își definesc asocierile (dependențele) fie prin parametrii constructorilor, fie prin argumentele unei metode de tip factory sau prin proprietăți de tip set, care trebuie apelate imediat după crearea obiectului.
- Un container “injectează” aceste dependențe când creează obiectul. Acest proces este invers celui tradițional, în care obiectul este responsabil de instanțierea sau localizarea dependențelor sale.
- În Spring, obiectele care formează un sistem (aplicație) soft sunt gestionate de containerul bazat pe IoC și sunt numite **bean**-uri.
- Un **bean Spring** este un obiect Java obișnuit care este instanțiat, asamblat și gestionat de containerul Spring IoC.
- Bean-urile și asocierile dintre ele sunt descrise în datele de configurare folosite de container.
- Două variante de a descrie bean-urile: folosind *fișiere de configurare* în format XML sau cod Java (fișiere de configurare sau autowire).

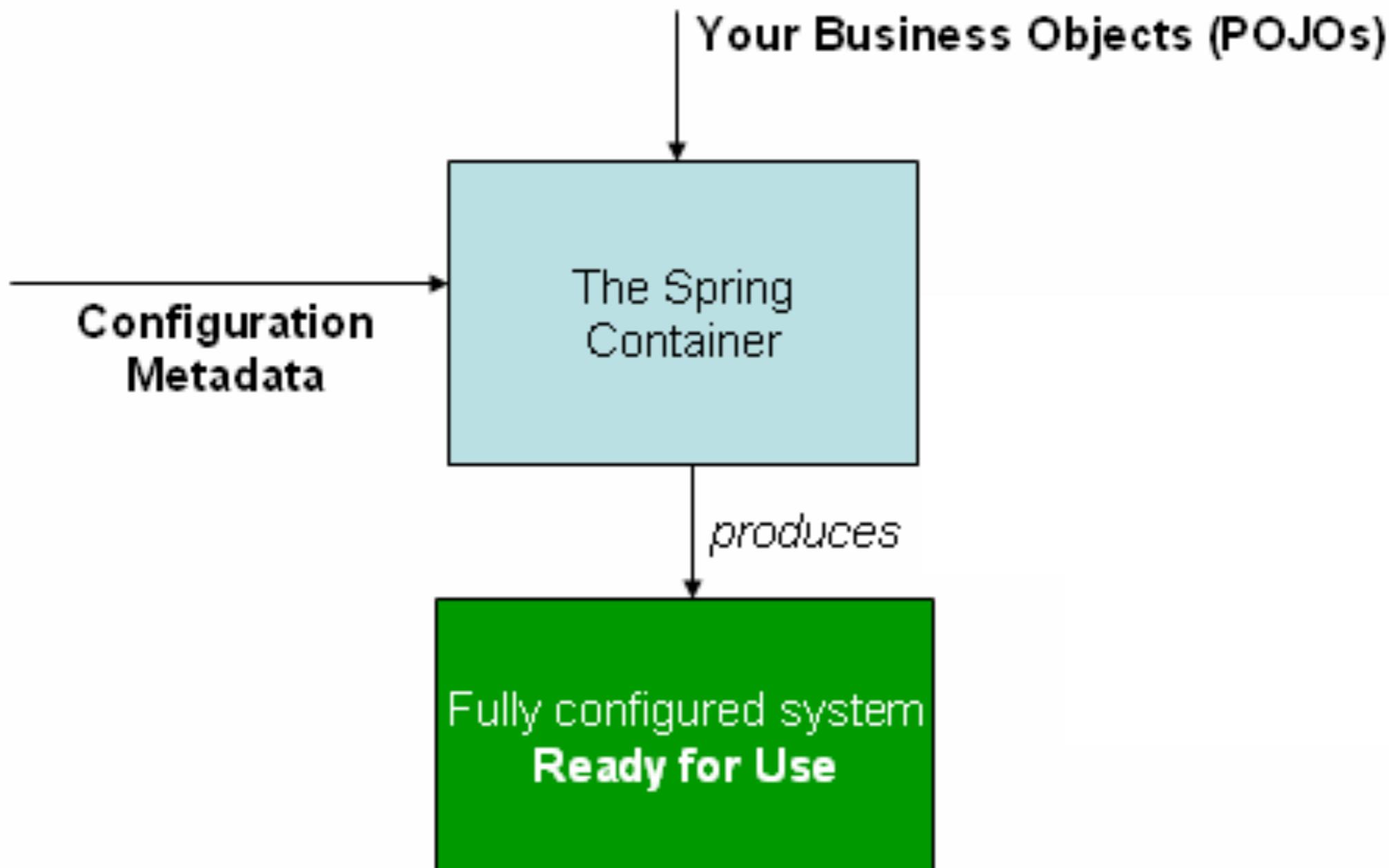
# Arhitectura framework-ului Spring



## Spring Framework Runtime



# Containerul Spring



# Crearea containerului Spring

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class StartApp{
    public static void main(String[] args) {
        ApplicationContext factory = new
        ClassPathXmlApplicationContext("classpath:spring-concurs.xml");
        //obtinerea referintei catre un bean din container
        ConcursService concurs= factory.getBean(ConcursService.class);
    }
}
```

# Fișierul de configurare XML

- Când se declară bean-urile folosind fișiere XML, elementul rădăcină a fișierului de configurare este `<beans>`.
- Un şablon simplu pentru fisierul de configurare este:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">
```

```
<!-- Declararea bean-urilor-->
```

```
</beans>
```

- În interiorul elementului `<beans>`, sunt descrise toate configurațiile specifice containerului Spring (dacă există) și toate declarațiile bean-urilor.

# Declararea unui bean simplu

```
package pizzax.validation;
import pizzax.model.Pizza;
public class DefaultPizzaValidator implements Validator<Pizza> {
    public void validate(Pizza pizza) {
        //...
    }
}
```

```
//spring-pizza.xml
<beans ...>
    <bean id="pizzaValidator"
          class="pizzax.validation.DefaultPizzaValidator"/>
</beans>
```

- Elementul **<bean>** este elementul de bază dintr-un fișier de configurare XML. El spune containerului Spring să creeze un obiect.
- Atributul **id** specifică numele prin care obiectul va fi referit în container.
- Când containerul Spring va încărca bean-urile, el va instanția bean-ul **"pizzaValidator"** folosind constructorul implicit.

# DI - Constructori

```
package pizzax.repository.file;  
import pizzax.repository;  
public class PizzaRepositoryFile implements PizzaRepository {  
    private String numefis;  
    public PizzaRepositoryFile(String numefis) {  
        ...  
    }  
    //implementarea metodelor  
    ...  
}  
//spring-pizza.xml  
...  
<bean id="pizzaRepository"  
    class="pizzax.repository.file.PizzaRepositoryFile">  
    <constructor-arg value="Pizza.txt" />  
</bean>
```

# DI - Constructori (2)

```
public class PizzaRepositoryFile implements PizzaRepository {  
    private String numefis;  
    private Validator<Pizza> valid;  
    public PizzaRepositoryFile(String numefis, Validator<Pizza> valid) { ... }  
    ...  
}  
//spring-pizza.xml  
...  
<bean id="pizzaValidator" class="pizzax.validation.DefaultPizzaValidator"/>  
<bean id="pizzaRepository"  
    class="pizzax.repository.file.PizzaRepositoryFile">  
    <constructor-arg value="Pizza.txt" />  
    <constructor-arg ref="pizzaValidator" />  
</bean>  
  
Validator<Pizza> pizzaValidator=new DefaultPizzaValidator();  
PizzaRepository pizzaRepository=new PizzaRepositoryFile("Pizza.txt",  
    pizzaValidator);
```

# DI - Constructori (3)

```
public class Produs {  
    private String denumire="";  
    private double pret=0;  
    public Produs(String denumire, double pret) {  
        this.pret = pret;  
        this.denumire = denumire;  
    }  
    //...  
}  
//spring-exemplu.xml  
<bean id="mere" class="Produs">  
    <constructor-arg index="0" value="Mere" />  
    <constructor-arg index="1" value="3.14"/>  
</bean>  
<!--sau -->  
<bean id="mere" class="Produs">  
    <constructor-arg type="java.lang.String" value="Mere" />  
    <constructor-arg type="double" value="3.14"/>  
</bean>
```

# DI - Metode factory

```
public class A {  
    private static A instance;  
    private A(){...};  
    public static A getInstance() { ... }  
    ...  
}  
  
//spring-exemplu.xml  
...  
<bean id="instanta" class="A" factory-method="getInstance"/>  
  
//echivalent cu  
A objA=A.getInstance();
```

# Scopul

Implicit toate bean-urile sunt **singleton** (se creează o singură instanță, indiferent de câte ori un bean este folosit la configurare, sau folosind metoda **getBean()** din clasa **ApplicationContext**).

Pentru a schimba scopul implicit, se folosește atributul “**scope**” al tag-ului **<bean>**

```
<bean id="bilet" class="xyz.Bilet" scope="prototype"/>
```

- *Valorile posibile* pentru atributul “**scope**” sunt:
  - **singleton**: o singură instanță pentru un container Spring.
  - **prototype**: pentru fiecare utilizare se creează un nou bean.
  - **request, session, global-session**: se utilizează pentru aplicații Web.

```
<bean id="beanA" class="test.B" scope="prototype"/>
```

```
<bean id="beanB" class="test.B"/>
```

```
<bean id="beanC" class="test.C">
```

```
  <constructor-arg ref="beanA"/> <!--se creeaza o noua instanta -->
```

```
</bean>
```

```
<bean id="beanD" class="test.D">
```

```
  <constructor-arg ref="beanA"/> <!--se creeaza o noua instanta -->
```

```
</bean>
```

# DI folosind proprietăți

```
package pizzax.repository.file;
import pizzax.repository;
public class PizzaRepositoryFile implements PizzaRepository {
    private String numefis;
    public PizzaRepositoryFile() { ... }
    public void setNumefisier(String numefis){...}
    //implementarea metodelor
    ...
}

//spring-pizza.xml
...
<bean id="pizzaRepository"
      class="pizzax.repository.file.PizzaRepositoryFile">
    <property name="numefisier" value="Pizza.txt"/>
</bean>
```

# DI folosind proprietăți

```
package pizzax.repository.file;
import pizzax.repository;
public class PizzaRepositoryMock implements PizzaRepository {
    private Validator<Pizza> valid;
    public PizzaRepositoryMock() { ... }
    public void setValidator(Validator<Pizza> v) {valid=v; }
    //implementarea metodelor
    ...
}

//spring-pizza.xml
...
<bean id="pizzaRepository"
      class="pizzax.repository.file.PizzaRepositoryMock">
    <property name="validator" ref="pizzaValidator"/>
</bean>
```

# DI Constructor + proprietăți

```
package pizzax.repository.file;
import pizzax.repository;
public class PizzaRepositoryFile implements PizzaRepository {
    private Validator<Pizza> valid;
    private String numefis;
    public PizzaRepositoryFile(String numefis) { ... }
    public void setValidator(Validator<Pizza> v){valid=v; }
    //implementarea metodelor
    ...
}

//spring-pizza.xml
...
<bean id="pizzaRepository"
      class="pizzax.repository.file.PizzaRepositoryMock">
    <constructor-arg value="Pizza.txt"/>
    <property name="validator" ref="pizzaValidator"/>
</bean>
```

# Bean-uri inner

```
package pizzax.repository.file;
import pizzax.repository;
public class PizzaRepositoryMock implements PizzaRepository {
    private Validator<Pizza> valid;
    public PizzaRepositoryMock() { ... }
    public void setValidator(Validator<Pizza> v) {valid=v; }
    //implementarea metodelor
    ...
}

//spring-pizza.xml
...
<bean id="pizzaRepository"
      class="pizzax.repository.file.PizzaRepositoryMock">
    <property name="validator">
        <bean class="pizzax.validation.DefaultPizzaValidator"/>
    </property>
</bean>
```

# Bean-uri inner

Observații:

- Bean-urile inner **nu** necesită specificarea atributului **id**. Se poate declara o valoare pentru **id**, dar nu este folosită de container.
- Aceste bean-uri nu pot fi reutilizate. Sunt folosite pentru “injectare” o singură dată și nu pot fi referite de alte bean-uri.

# Proprietăți de tip colecție

- Există situații când o proprietate/parametru-constructor este de tip **container** (*colecție, mulțime, dicționar, tablou, etc...*).
- Pentru a inițializa acest tip de proprietăți Spring a definit 4 elemente de configurare:
  - **<list>**: o listă de valori ce poate conține duplicate
  - **<set>**: o lista de valori ce nu conține duplicate
  - **<map>**: o mulțime de perechi cheie-valoare (dicționar)
  - **<props>**: o mulțime de perechi cheie-valoare, unde atât cheia cât și valoarea sunt de tip **String** (clasa `java.util.Properties`)

# Proprietăți de tip colecție

- Liste, mulțimi, tablouri:

```
class Produs{  
    private String denumire;  
    private double pret;  
    public Produs() {...}  
    public void setDenumire(String d) {...}  
    public void setPret(double d) {...}
```

```
//metode get si set
```

```
}
```

```
class Depozit{  
//...  
    public void setProduse(java.util.List<Produs> lp) {...}  
    //sau  
    public void setProduse(java.util.Collection<Produs> lp) {...}  
    //sau  
    public void setProduse(Produs[] lp) {...}  
}
```

# Proprietăți de tip colecție

- Liste, tablouri:

```
//spring-exemplu.xml
<bean id="mere" class="Produs">
    <property name="denumire" value="Mere"/>
    <property name="pret" value="2.3"/>
</bean>
<bean id="pere" class="Produs"> ...</bean>
<bean id="prune" class="Produs"> ...</bean>
<bean id="depozit" class="Depozit">
    <property name="produse">
        <list>
            <ref bean="mere"/>
            <ref bean="pere"/>
            <ref bean="prune"/>
        </list>
    </property>
</bean>
```

# Proprietăți de tip colecție

- Multimi:

```
//spring-exemplu.xml
<bean id="mere" class="Produs">
    <property name="denumire" value="Mere"/>
    <property name="pret" value="2.3"/>
</bean>
<bean id="pere" class="Produs"> ...</bean>
<bean id="prune" class="Produs"> ...</bean>
<bean id="depozit" class="Depozit">
    <property name="produse">
        <set>
            <ref bean="mere"/>
            <ref bean="pere"/>
            <ref bean="prune"/>
            <ref bean="prune"/>
        </set>
    </property>
</bean>
```

# Proprietăți de tip colecție

- Dicționare:

```
class Depozit{  
    //...  
    public void setProduse(java.util.Map<String, Produs> lp) {...}  
}
```

```
//spring-exemplu.xml  
<bean id="mere" class="Produs">...</bean>  
<bean id="pere" class="Produs"> ...</bean>  
<bean id="prune" class="Produs"> ...</bean>  
<bean id="depozit" class="Depozit">  
    <property name="produse">  
        <map>  
            <entry key="pMere" value-ref="mere"/>  
            <entry key="pPere" value-ref="pere"/>  
            <entry key="pPrune" value-ref="prune"/>  
        </map>  
    </property>  
</bean>
```

# Proprietăți de tip colecție

- Dicționare: elementul `<entry>` are următoarele attribute:
  - **key**: specifică *cheia* ca și *string*;
  - **key-ref**: specifică *cheia* ca și *referință la alt bean din container*;
  - **value**: specifică *valoarea* ca și *string*;
  - **value-ref**: specifică *valoarea* ca și *referință la un alt bean* din container.
- Proprietăți: elementele **props** și **prop**

```
class Depozit{
    public void setProprietati(Properties p) {...}
}
```

```
<bean id="depozit" class="Depozit">
<property name="proprietati">
    <props>
        <prop key="prop1"> A1 </prop>
        <prop key="prop2"> B C2 </prop>
    </props>
</property>
</bean>
```

# Bean-uri tip container

- Există situații când trebuie creat un bean de tip container (colecție, multime, dicționar, tablou, etc...).
- Pentru a crea un bean de tip container:

- ```
<util:list>, <util:set>, <util:map>, <util:props>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/util
                           http://www.springframework.org/schema/util/spring-util.xsd"> ... </beans>
```

```
<util:properties id="jdbcProps">
    <prop key="tasks.jdbc.driver">org.sqlite.JDBC</prop>
    <prop key="tasks.jdbc.url">jdbc:sqlite:database.db</prop>
</util:properties>
sau
<util:properties id="jdbcProps" location="classpath:bd.config"/>
```

# Valori null

- E posibil ca valoarea unei proprietăți să fie setată la null. Pentru aceasta se folosește elementul <null/>

```
<property name="numeProprietate"> <null/></property>
```

- **SpEL (Spring Expression Language)**

- a fost introdus începând cu versiunea 3.0
- permite calcularea/ determinarea valorilor unor proprietăți în timpul execuției:

```
<property name="numarAleator" value="#{T(java.lang.Math).random() }"/>
```

```
<property name="song" value="#{songSelector.selectSong().toUpperCase()}"/>
```

```
<property name="fullName"
```

```
  value="#{person.firstName + ' ' + person.lastName}"/>
```

# DI constructor vs. DI proprietăți

## **Recomandări:**

- *DI constructor* - pentru dependențe *obligatorii*
- DI proprietăți - pentru dependențe *optionale*
- Situații speciale:
  - constructori cu prea mulți parametrii
- Verificarea dependențelor optionale că sunt nenule!

# Configurare folosind Gradle

- Fișierul **build.gradle**

```
dependencies {  
  
    implementation 'org.springframework:spring-context:5.3.26'  
  
    runtimeOnly group: 'org.xerial', name: 'sqlite-jdbc', version: '3.16.1'  
  
    testImplementation group: 'junit', name: 'junit', version: '4.11'  
}
```

# Configurare Spring folosind XML

- Exemplu
  - TaskSpringXMLConfig

# Configurare folosind JavaConfig

- Specificarea bean-urilor care trebuie create se face într-o altă clasă, adnotată cu `@Configuration`:

```
@Configuration  
public class PizzerieConfig {  
    //...  
}
```

- Adnotarea marchează clasa ca și o clasă de configurare
- Containerul Spring consideră că această clasă conține detalii despre bean-urile ce trebuie create și cum trebuie create.

# Configurare folosind JavaConfig

- Declararea unui bean se face cu adnotarea `@Bean`:

```
import pizzax.validation.DefaultPizzaValidator;  
  
@Configuration  
public class PizzerieConfig {  
  
    @Bean  
    public Validator<Pizza> validator(){  
        return new DefaultPizzaValidator();  
    }  
}
```

- Id-ul implicit al bean-ului este numele metodei (ex. `validator`)
- Setarea unui id explicit se face folosind elementul “`name`”.

```
@Bean (name="pizzaVal")  
public Validator<Pizza> validator(){  
    return new DefaultPizzaValidator();  
}
```

# Configurare folosind JavaConfig

- Injectarea dependențelor:

```
@Configuration  
public class PizzerieConfig {  
  
    @Bean  
    public Validator<Pizza> validator() {  
        return new DefaultPizzaValidator();  
    }  
  
    //varianta a - apelul metodei care creeaza bean-ul  
    @Bean  
    public PizzaRepositoryMock pizzaRepo() {  
        return new PizzaRepositoryMock(validator());  
    }  
  
    @Bean  
    public PizzaRepositoryFile pizzaFileRepo() {  
        return new PizzaRepositoryFile(validator(),"Pizza.txt");  
    }  
}
```

- Un singur bean de tip Validator<Pizza> este creat!

# Configurare folosind JavaConfig

- Injectarea dependențelor:

```
@Configuration  
public class PizzerieConfig {  
  
    @Bean  
    public Validator<Pizza> validator() {  
        return new DefaultPizzaValidator();  
    }  
  
    //varianta b - transmiterea bean-ului ca si parametru  
    @Bean  
    public PizzaRepositoryMock pizzaRepo(Validator<Pizza> val) {  
        return new PizzaRepositoryMock(val);  
    }  
  
    @Bean  
    public PizzaRepositoryFile pizzaFileRepo(Validator<Pizza> val) {  
        return new PizzaRepositoryFile(val, "Pizza.txt");  
    }  
}
```

# Configurare folosind JavaConfig

- În interiorul unei metode anotate cu @Bean se poate folosi orice cod Java necesar creării bean-ului:

```
@Configuration
```

```
public class PizzerieConfig {  
  
    @Bean  
    public PizzaRepositoryFile pizzaFileRepo(Validator<Pizza> val) {  
        PizzaRepositoryFile repo=new new PizzaRepositoryFile("Pizza.txt");  
        repo.setValidator(val);  
        return repo;  
    }  
  
    @Bean  
    public PizzaRepositoryJdbc pizzaJdbcRepo() {  
        Properties jdbcProps=new Properties();  
        try {  
            jdbcProps.load(new FileReader("bd.config"));  
        } catch (IOException e) {  
            System.out.println("No properties were set. Cannot find bd.config "+e);  
        }  
        return new PizzaRepositoryJdbc(jdbcProps);  
    }  
}
```

# Crearea containerului Spring JavaConfig

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.AnnotationConfigApplicationContext;

public class StartApp{
    public static void main(String[] args) {
        ApplicationContext context=new
            AnnotationConfigApplicationContext(PizzerieConfig.class);
        //obtinerea referintei catre un bean din container
        PizzaService repo= factory.getBean(PizzaService.class);
    }
}
```

# Configurare Spring folosind JavaConfig

- Exemplu
  - TaskSpringJavaConfig

# Configurare automată folosind Java

- Configurarea automată (implicită) este o metodă prin care **containerul Spring descoperă automat bean**-rile care trebuie create, dependențele dintre acestea și încearcă să creeze aceste bean-uri.
- **@ComponentScan** specifică containerului opțiunea de configurare automată

```
package pizzerie.config;  
@Configuration  
@ComponentScan  
public class PizzerieAutowireConfig {  
}
```

- Implicit, containerul va încerca să descopere bean-urile începând cu pachetul clasei de configurare.

```
@ComponentScan("pizzerie")  
@ComponentScan(basePackages={"pizzerie","cofetarie"})  
@ComponentScan(basePackageClasses={C.class, D.class})
```

# Configurare automată folosind Java

- Declararea bean-urilor: **@Component**

**@Component**

```
public class DefaultPizzaValidator implements Validator<Pizza> {  
    public void validate(Pizza pizza) {  
        //...  
    }  
}
```

- Implicit, **id**-ul bean-ului este numele clasei cu prima literă transformată în literă mică.
- Bean cu id explicit:

**@Component("pizzaVal")**

```
@Component("pizzaVal")  
public class DefaultPizzaValidator implements Validator<Pizza> {  
    public void validate(Pizza pizza) {  
        //...  
    }  
}
```

# Configurare automată folosind Java

- Marcarea dependențelor: **@Autowired**
- Constructori, atribute, metode (set, etc.)

**@Component**

```
public class PizzaInMemoryRepository implements PizzaRepository {  
    private Validator<Pizza> valid;  
  
    @Autowired  
    public PizzaInMemoryRepository(Validator<Pizza> valid) { ... }  
  
    ...  
}
```

**@Component**

```
public class PizzaInMemoryRepository implements PizzaRepository {  
    private Validator<Pizza> valid;  
  
    public PizzaInMemoryRepository() { ... }  
  
    @Autowired  
    public void setValidator(Validator<Pizza> val) {...}  
}
```

# Configurare automată folosind Java

- Marcarea dependențelor: @Autowired

**@Component**

```
public class PizzaInMemoryRepository implements PizzaRepository {  
    private Validator<Pizza> validator;  
    public PizzaInMemoryRepository() { ... }  
    @Autowired(required=false)  
    public void setValidator(Validator<Pizza> val) {...}  
}
```

- Dacă nu există nici un bean care să satisfacă dependență, proprietatea va rămâne neinitializată.

```
public void save(Pizza p) {  
    if (validator!=null) {  
        //...  
    }  
}
```

# Configurare automată folosind Java

- `@Component`, `@Autowired`: adnotări specifice frameworkului Spring
- Dependența codului de frameworkul Spring
- `@Named`, `@Inject` : adnotări din specificația *Java Dependency Injection*
- Pachetul `javax.inject`

```
import javax.inject.Inject;  
import javax.inject.Named;
```

`@Named`

```
public class PizzaRepositoryMock implements PizzaRepository {  
    private Validator<Pizza> valid;  
    @Inject  
    public PizzaRepositoryMock(Validator<Pizza> val) { ... }  
    //...  
}
```

- În majoritatea cazurilor sunt interschimbabile.

# Configurare automată folosind Java

- **@Scope**: specificarea scopului (*implicit singleton*, prototype, request, session)
- **@Component, @Bean**

**@Component**

```
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)  
public class ABean { ... }
```

**@Bean**

```
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)  
public ABean abean() {  
    return new ABean();  
}
```

# Crearea containerului Spring Java Autowire

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.AnnotationConfigApplicationContext;

public class StartApp{
    public static void main(String[] args) {
        ApplicationContext context=new
        AnnotationConfigApplicationContext(PizzerieAutowireConfig.class);
        //obtinerea referintei catre un bean din container
        PizzaService repo= factory.getBean(PizzaService.class);
    }
}
```

# Configurare automată folosind Java

- Ambiguități.

`@Autowired`

```
public void setDessert(Dessert dessert) {  
    this.dessert = dessert;  
}
```

`@Component`

```
public class Cake implements Dessert { ... }
```

`@Component`

```
public class Cookies implements Dessert { ... }
```

`@Component`

```
public class IceCream implements Dessert { ... }
```

- **Care bean satisface dependență?**

```
NoUniqueBeanDefinitionException: nested exception is  
    org.springframework.beans.factory.NoUniqueBeanDefinitionException:  
No qualifying bean of type [com.desserteater.Dessert] is defined:  
    expected single matching bean but found 3: cake,cookies,iceCream
```

# Configurare automată folosind Java

- Ambiguități - Soluția 1 - `@Primary`.

`@Autowired`

```
public void setDessert(Dessert dessert) {  
    this.dessert = dessert;  
}
```

`@Component`

`@Primary`

```
public class Cake implements Dessert { ... }
```

`@Component`

```
public class Cookies implements Dessert { ... }
```

`@Component`

```
public class IceCream implements Dessert { ... }
```

- Soluția 2 - `@Qualifier`
- Soluția 3 - Adnotare proprie

# Configurare Spring folosind Autowire

- Exemplu
  - TaskSpringAutowire

# Configurare XML vs. JavaConfig vs Autowire

XML	JavaConfig	Autowire
<ul style="list-style-type: none"><li>• Nu necesită modificarea codului sursă</li><li>• Nu necesită recompilarea când apar modificări</li><li>• Necesită învățarea unui nou limbaj (XML)</li><li>• Se poate folosi când nu avem acces la tot codul sursă al aplicației</li><li>• Nu se pot verifica tipurile bean-urilor și dependențele la compilare</li><li>• Nu apar ambiguități</li></ul>	<ul style="list-style-type: none"><li>• Necesită recompilare când apar modificări</li><li>• Nu necesită învățarea unui nou limbaj</li><li>• Se verifică static tipurile</li><li>• Se poate folosi când nu avem acces la tot codul sursă al aplicației</li></ul>	<ul style="list-style-type: none"><li>• Necesită recompilare când apar modificări</li><li>• Nu necesită învățarea unui nou limbaj</li><li>• Se verifică static tipurile</li><li>• NU se poate folosi când nu avem acces la tot codul sursă al aplicației</li><li>• Dependența codului sursă de Spring</li></ul>

# Referințe Spring

- Documentația frameworkului Spring  
<http://docs.spring.io/spring-framework/docs/current/spring-framework-reference/html/index.html>
- Craig Walls, *Spring in Action*, Fourth Edition, Ed. Manning, 2015 (sau versiuni mai noi: fifth edition, sixth edition)
- Alte tutoriale ...

# Medii de proiectare și programare

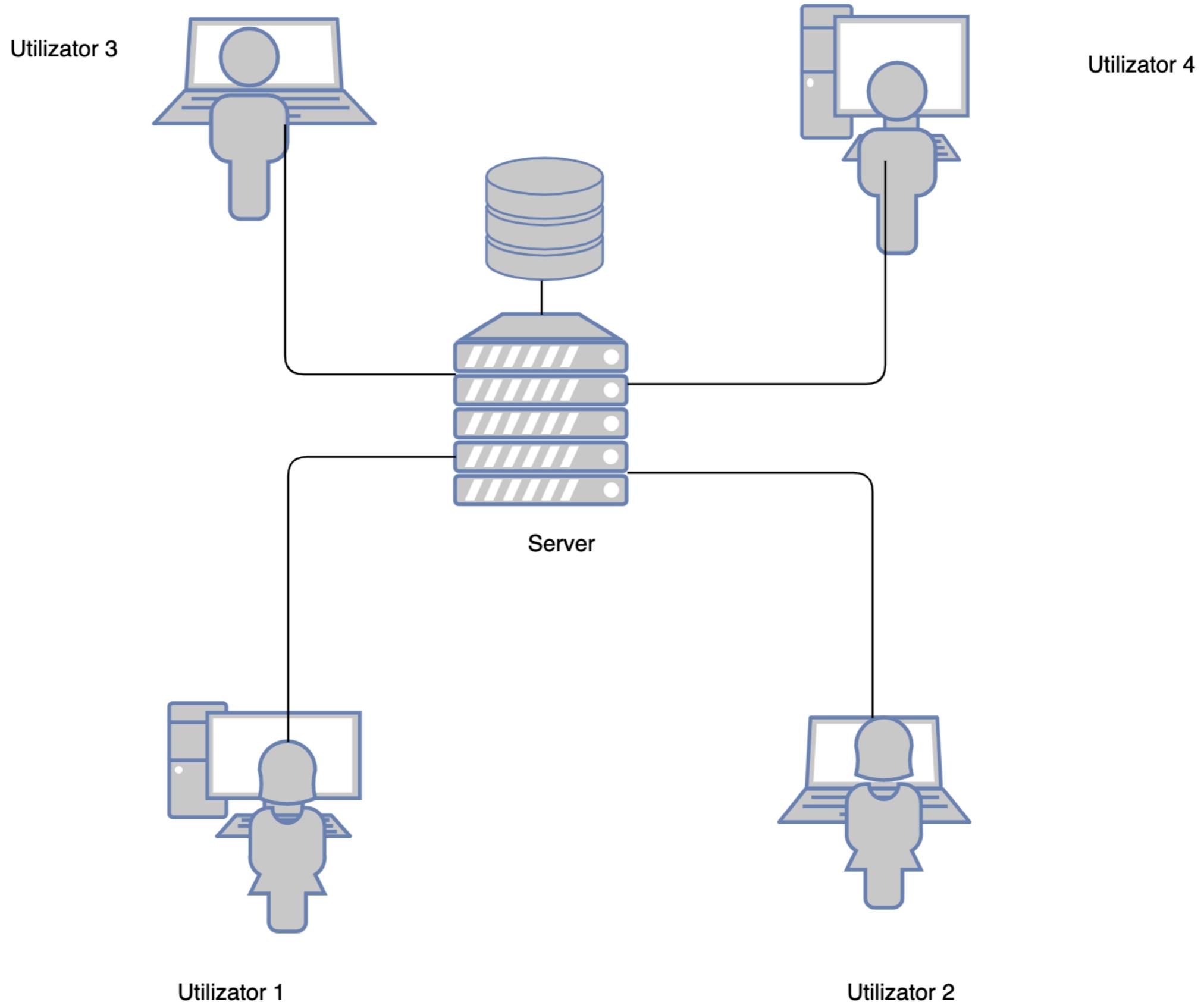
2022-2023

Curs 5

# Continut curs 5

- Aplicații client-server
  - Șablonul Proxy
  - Data transfer object
- Networking și threading în Java
  - Exemplu Mini-Chat
- Networking și threading în C# (curs 6)

# Aplicații client-server



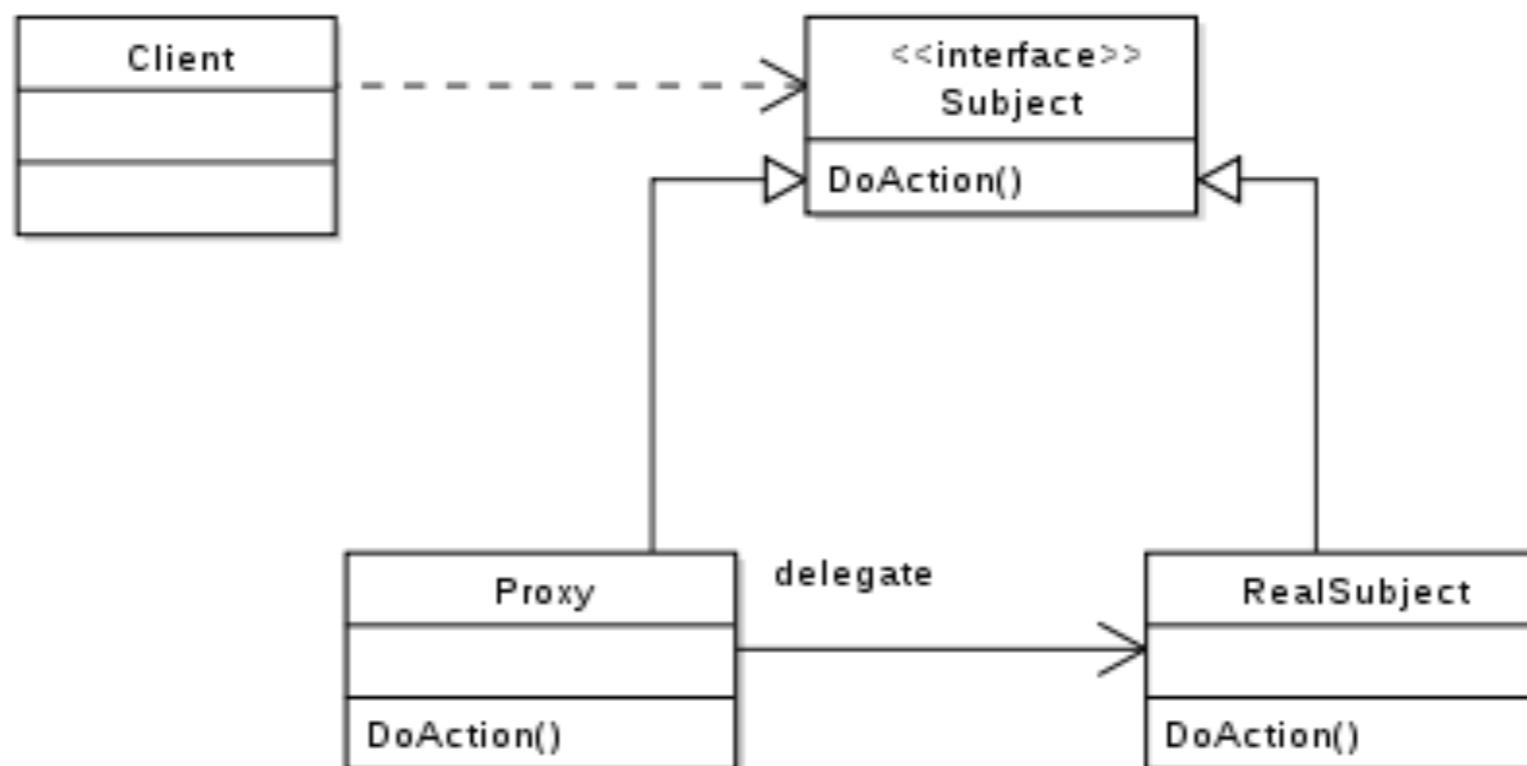
# Mini-chat

- Proiectați și implementați o aplicație client-server pentru un mini-chat având următoarele funcționalități:
  - *Login.* După autentificarea cu succes, o nouă fereastră se deschide în care sunt afișați toți prietenii *online* ai utilizatorului și o listă cu mesajele trimise/primită de utilizator. De asemenea, toți prietenii online văd în lista lor că utilizatorul este *online*.
  - *Trimiterea unui mesaj.* Un utilizator poate trimite un mesaj text unui prieten care este online. După trimiterea mesajului, prietenul vede automat mesajul în fereastra lui.
  - *Logout.* Toți prietenii online ai utilizatorului văd în lista lor că utilizatorul nu mai este *online*.

# Exemplu Java

# Şablonul Proxy

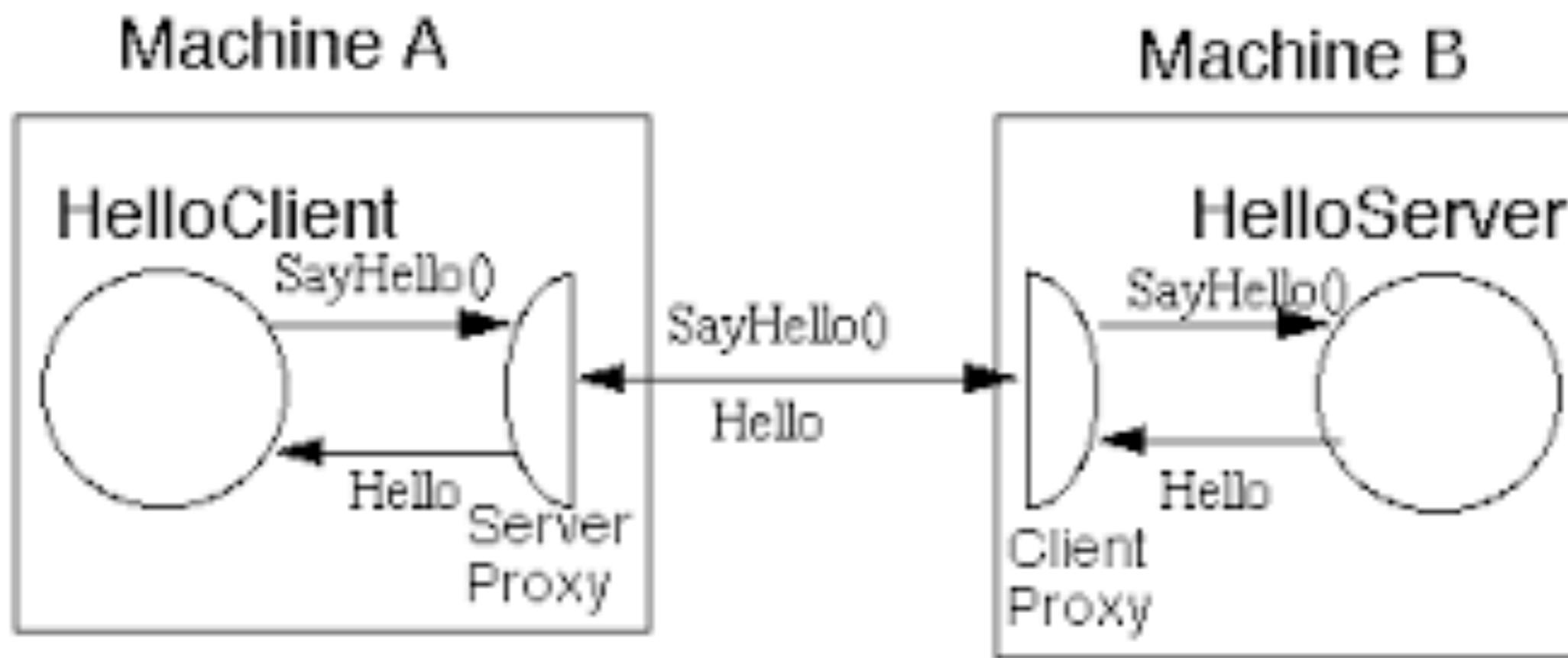
- Asigură pentru un obiect existent, un surogat sau un înlocuitor în scopul controlării accesului la acesta.
- Înlocuitorul poate fi:
  - *Proxy la distanță* (eng. *remote proxy*) - obiect în alt spațiu de adresă,
  - *Proxy virtual* (eng. *virtual proxy*) - un obiect mare din memorie,
  - *Proxy de protecție* - controlează accesul la obiectul original,
  - etc.



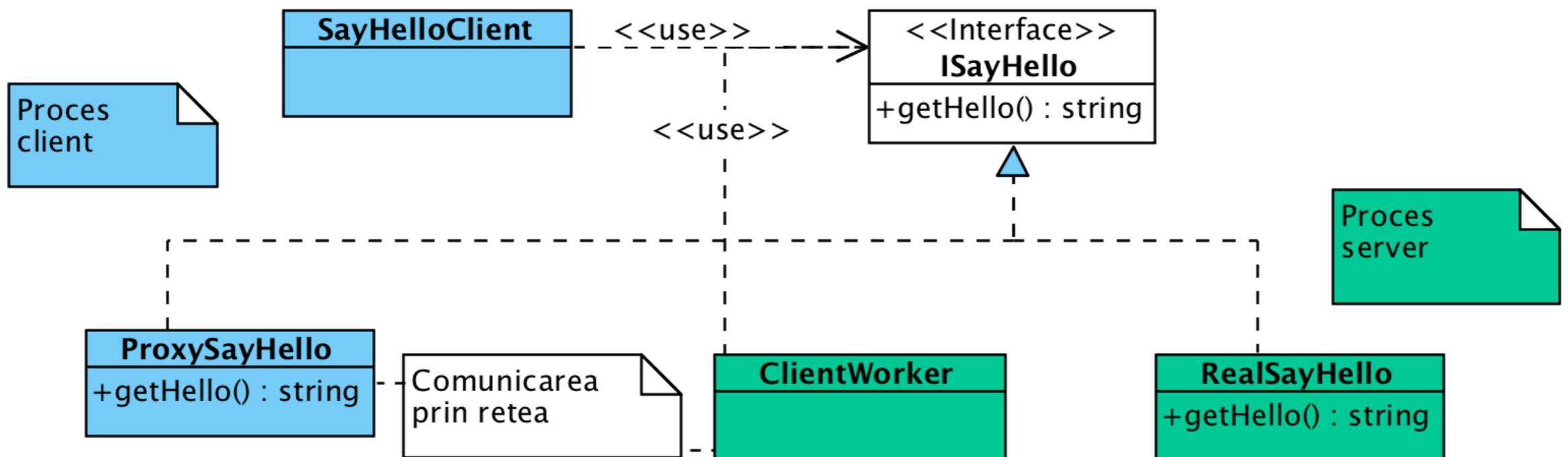
# Şablonul *Remote Proxy*

- **Remote Proxy** oferă un înlocuitor local pentru un obiect aflat în alt spațiu de adresă/memorie.
- Înlocuitorul este responsabil cu codificarea unei cereri, a parametrilor și trimiterea lor către obiectul real aflat într-un spațiu de adresă diferit.
- *Clientul* cererii crede că comunică cu obiectul real, dar este un proxy între ei.
- Proxy-ul transformă cererile clientului în cereri la distanță, obține rezultatul cererii și îl transmite clientului.

# Şablonul Remote Proxy



# Şablonul Remote Proxy

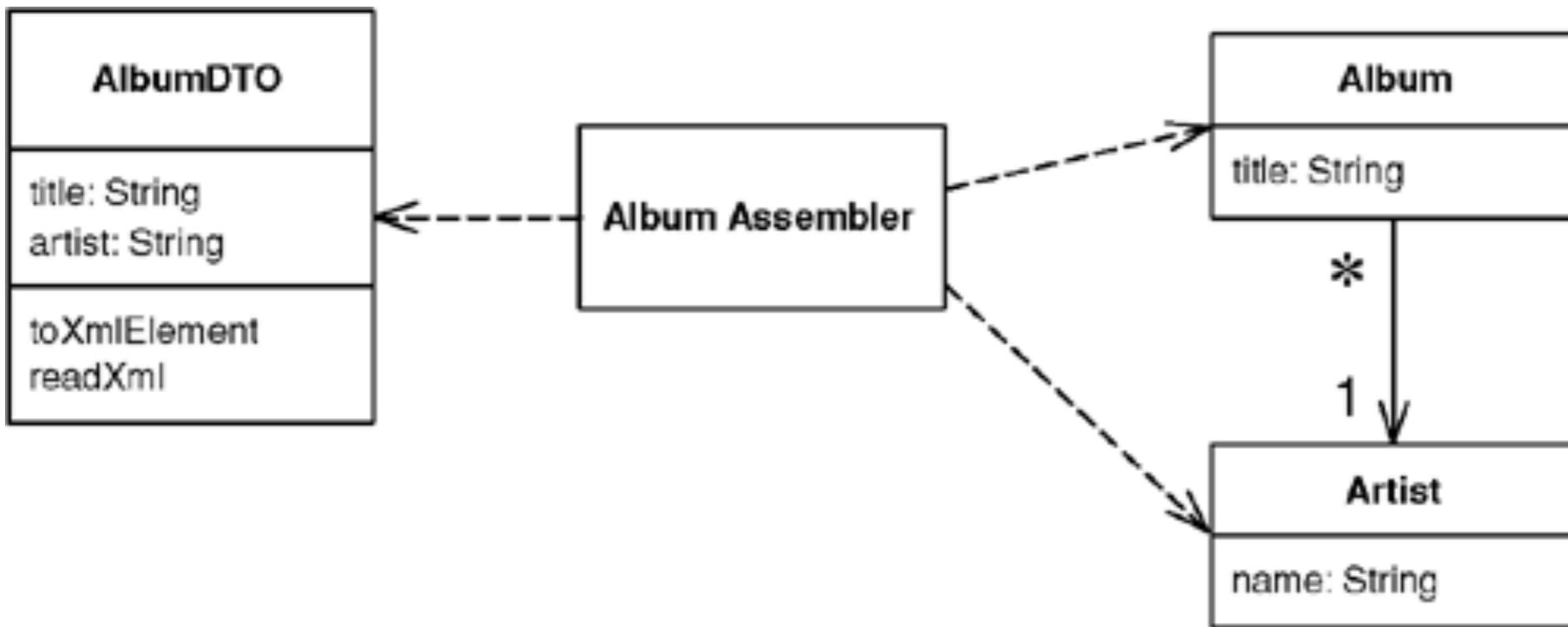


# Şablonul *Data Transfer Object* (DTO)

- Un **DTO** este un obiect care conține informația ce trebuie transmisă între unul sau mai multe procese pentru a reduce numărul de apeluri (dintre procese).
- Fiecare apel de metodă remote este costisitor, de aceea numărul de apeluri ar trebui redus și mai multă informație ar trebui transmisă la un apel.
- O soluție posibilă este de a folosi mai mulți parametri:
  - dificil de programat
  - în unele cazuri nu este posibil (ex., în Java o metodă poate returna o singură valoare).
- Soluția: crearea unui DTO (*Data Transfer Object*) care păstrează toată informația necesară unui apel. De obicei obiectul este serializabil (binar, XML, etc.) pentru a putea fi transmis prin rețea.
- Un alt obiect este responsabil cu conversia datelor din model într-un DTO și invers.

# Şablonul Data Transfer Object

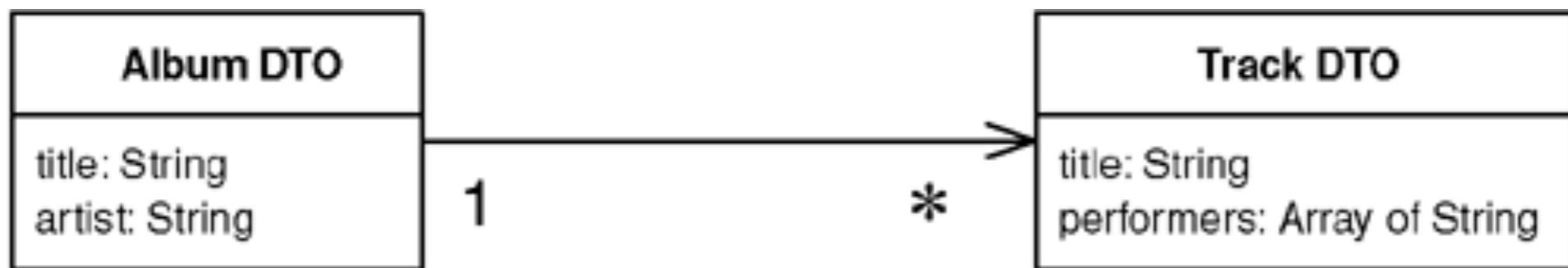
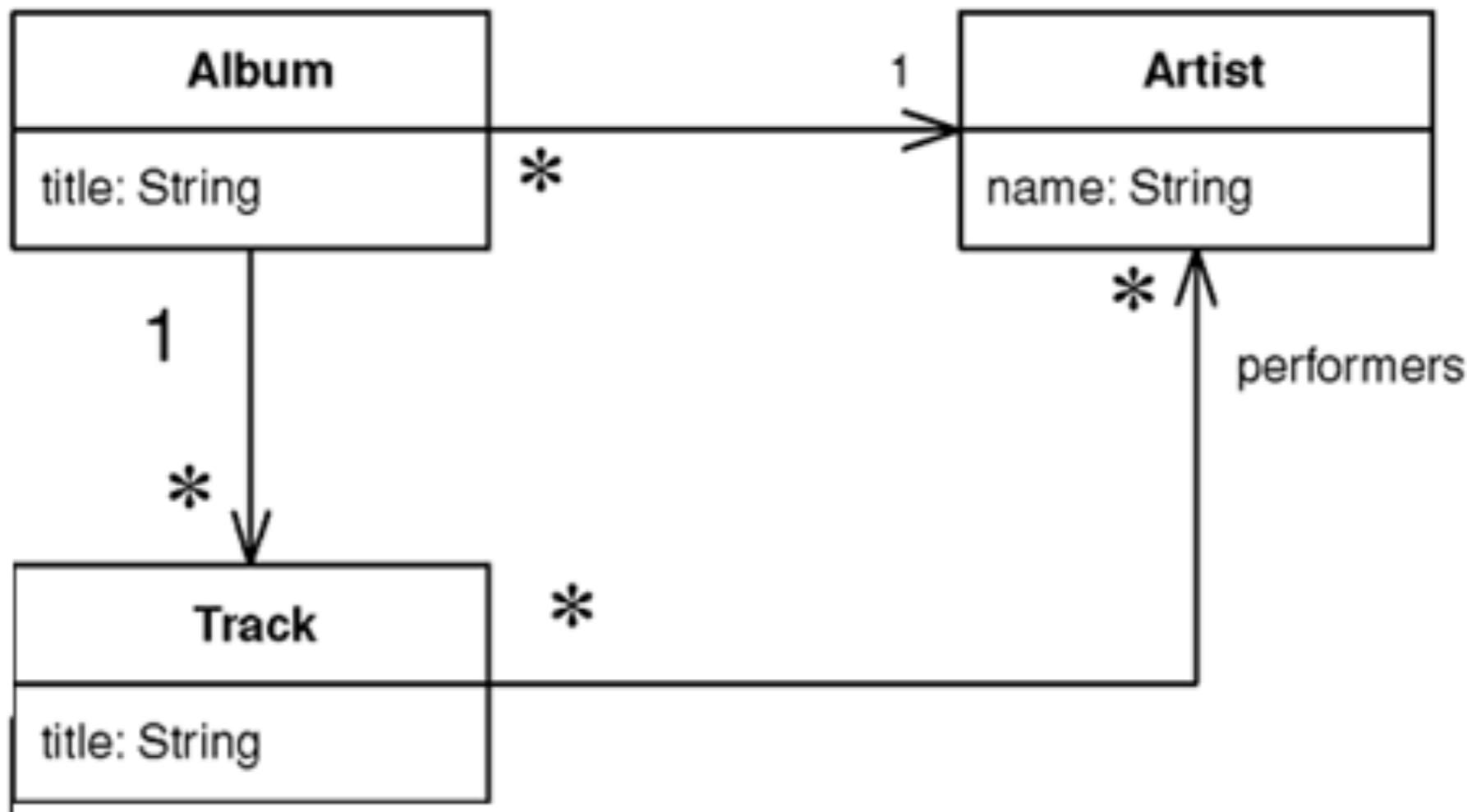
- Un DTO conține, de obicei, *multe atribute și metode de tip get/set pentru acestea.*
- Când un obiect remote are nevoie de date, cere DTO-ul corespunzător. DTO poate să conțină mai multă informație decât este necesară la acel apel, dar ar trebui să conțină toată informația de care va avea nevoie obiectul remote o perioadă.
- Un DTO conține de obicei informație provenind de la mai multe obiecte din model.



# Şablonul Data Transfer Object

- Un DTO ar trebui folosit ori de câte ori este necesară transmiterea mai multor date între două procese într-un singur apel de metodă.
- *Alternative:*
  - De a folosi metode de tip set/get cu mai mulți parametrii transmiși prin referință.
    - Multe limbaje (ex. Java) permit returnarea unei singure valori.
    - Alternativa poate fi folosită pentru actualizări (metode de tip set), dar nu poate fi folosită pentru a obține date (metode de tip get).
  - Folosirea unei reprezentări sub formă de string.
    - Totul va fi cuplat cu reprezentarea sub formă de string (poate fi costisitoare).

# Data Transfer Object - Exemplu



# Data Transfer Object - Exemplu

```
class AlbumAssembler{
    public AlbumDTO writeDTO(Album subject) {
        AlbumDTO result = new AlbumDTO();
        result.setTitle(subject.getTitle());
        result.setArtist(subject.getArtist().getName());
        writeTracks(result, subject);
        return result;
    }
    private void writeTracks(AlbumDTO result, Album subject) {
        List<TrackDTO> newTracks = new ArrayList<TrackDTO>();
        for(Track track: subject.getTracks()){
            TrackDTO newDTO = new TrackDTO();
            newDTO.setTitle(track.getTitle());
            writePerformers(newDTO, track);
            newTracks.add(newDTO);
        }
        result.setTracks(newTracks.toArray(new TrackDTO[newTracks.size()]));
    }
}
```

# Data Transfer Object - Exemplu

```
private void writePerformers(TrackDTO dto, Track subject) {  
    List<String> result = new ArrayList<String>();  
    for(Artist artist: subject.getPerformers()) {  
        result.add(artist.getName());  
    }  
    dto.setPerformers(result.toArray(new String[result.size()]));  
}
```

# Networking în Java

- `java.net` - pachetul conține clase pentru comunicarea TCP/UDP prin rețea.
- TCP: `Socket` și `ServerSocket`.
- UDP: `DatagramPacket`, `DatagramSocket` și `MulticastSocket`.
- Clasa `InetAddress` reprezintă o adresă IP:
  - `Inet4Address`: pentru adrese IPv4 (32 bits).
  - `Inet6Address`: pentru adrese IPv6 (128 bits).

```
InetAddress localHost=InetAddress.getLocalHost();  
InetAddress googAddr=InetAddress.getByName("www.google.com");
```

- `InetSocketAddress` asociere între o adresă IP și un port:

```
InetSocketAddress(InetAddress addr, int port);  
InetSocketAddress(String hostname, int port);
```

# Networking in Java

- **ServerSocket** reprezintă clasa corespunzătoare serverului care așteaptă conexiuni TCP.
- Constructori/Metode:

```
public ServerSocket(int port) throws BindException, IOException  
  
public ServerSocket( ) throws IOException //not bind yet, since Java 1.4  
  
//binds a server to a port  
public void bind(SocketAddress endpoint) throws IOException  
  
//blocks and waits for clients  
public Socket accept( ) throws IOException  
  
//closes the server  
public void close() throws IOException
```

# Networking in Java

```
ServerSocket server=null;  
try{  
    server=new ServerSocket(5555);  
    while(keepProcessing){  
        Socket client=server.accept();  
        //processing code  
    }  
}catch(IOException ex){  
    //...  
}finally{  
    if(server!=null){  
        try{  
            server.close();  
        }catch(IOException ex){...}  
    }  
}
```

# Networking în Java

- `java.net.Socket` deschide o conexiune TCP din partea clientului.

```
public Socket(String host, int port) throws UnknownHostException,  
           IOException
```

```
public Socket(InetAddress host, int port) throws IOException
```

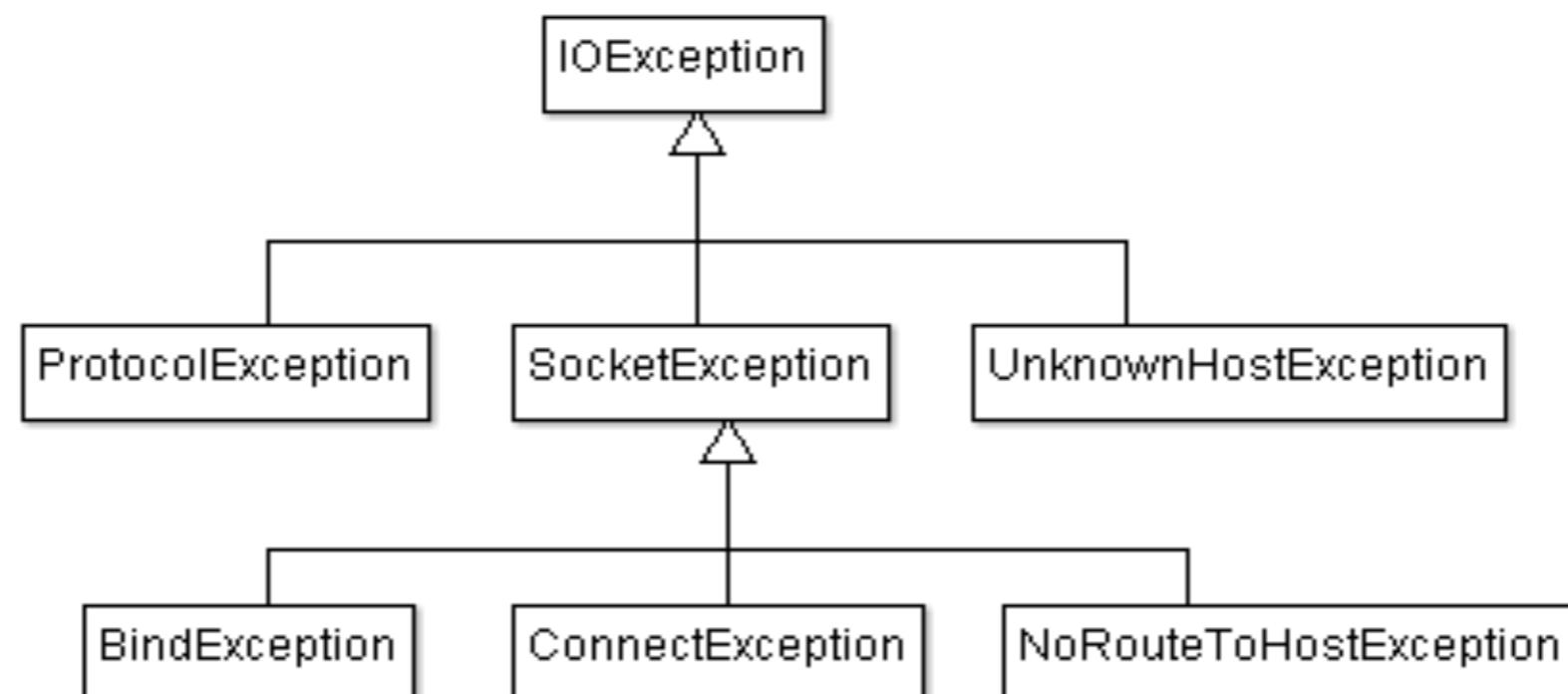
- Metode:

```
public int getPort()  
public InputStream getInputStream() throws IOException  
public OutputStream getOutputStream() throws IOException  
public void close() throws IOException
```

# Networking in Java

```
try (Socket connection=new Socket("172.30.106.5", 5555)) {  
    //processing code  
  
} catch (UnknownHostException e) {  
    //...  
} catch (IOException e) {  
    //...  
}
```

# Excepții



# Threading în Java

- Două modalități de *definire a unui thread*:
  - Extinderea clasei **Thread** și redefinirea metodei **run**.
  - Implementarea interfeței **Runnable** și definirea metodei **run**.
- Crearea unui thread se face prin intermediul clasei **Thread**

```
public Thread()  
public Thread(Runnable target)
```
- Pornirea execuției unui thread se face prin apelul metodei **start** din clasa **Thread**:

```
public void start()
```

# Sincronizarea threadurilor

- Instrucțiunea **synchronized**

```
synchronized(locker_obj) {
    //code to execute
}
```

- Sincronizarea unei metode:

```
public synchronized void methodA();
```

- Yielding: *un thread renunță la CPU alocat și permite execuția altui thread:*

```
public static void yield( );
```

```
public void run( ) {
    while (true) {
        // Time and CPU consuming thread's work... ,
        Thread.yield( );
    }
}
```

# Utilități Java Concurrency

- Java 5 a introdus **utilități pentru concurență** - un framework extensibil care permite **crearea containerelor de thread-uri și cozi sincronizate** (eng. *blocking queues*):
  - **java.util.concurrent**: Tipuri utile în programarea concurență ( ex. executors)
  - **java.util.concurrent.atomic**: Programare concurență avansată
  - **java.util.concurrent.locks**: Mecanisme de blocare avansate, mai performante decât notify/wait.

# Taskuri Java

- Un obiect *task Java* este un obiect a cărui clasa implementează interfața `java.lang.Runnable` (*taskuri runnable*) sau interfața `java.util.concurrent.Callable` (*taskuri callable*).

```
public interface Runnable{  
    void run()  
}  
  
public interface Callable<V>{  
    V call() throws Exception  
}
```

- Metoda `call()` poate returna o valoare și poate arunca excepții (checked).

# Execuția taskurilor Java

- Interfața Executor - execuția taskurilor runnable:

```
public interface Executor{  
    void execute(Runnable command)  
}
```

- ScheduledThreadPoolExecutor, ThreadPoolExecutor

- *Dezavantaje:*

- Se axeaza doar pe **Runnable**. Metoda **run()** nu returnează nici o valoare. Este dificilă returnarea unei valori ca și rezultat al execuției taskului.
- Nu oferă posibilitatea monitorizării progresului execuției unui task runnable care se execută (se execută încă?, anulat? execuția s-a încheiat?)
- Nu poate executa mai multe taskuri.
- Nu ofera posibilitatea opririi unui executor.

# ExecutorService

- `java.util.concurrent.ExecutorService` soluția pentru problemele apărute la interfața `Executor`.
- Este implementat folosind un container de threaduri (eng. *thread pool*).

```
public interface ExecutorService extends Executor {  
  
    void shutdown();  
  
    List<Runnable> shutdownNow();  
  
    <T> Future<T> submit(Callable<T> task);  
  
    <T> Future<T> submit(Runnable task, T result);  
  
    <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks);  
  
    <T> T invokeAny(Collection<? extends Callable<T>> tasks);  
  
    //alte metode  
}
```

- `ScheduledThreadPoolExecutor`, `ThreadPoolExecutor`

***Executorul trebuie oprit după terminarea execuției, altfel aplicația nu își va încheia execuția.***

# Interfața Future

- Un obiect de tip **Future** reprezintă rezultatul unui calcul asincron.
- Rezultatul este numit *future* pentru că de obicei nu va fi disponibil decât la un moment în viitor.
- Are metode pentru: anularea execuției unui task, obținerea rezultatului execuției, determinarea dacă un task și-a încheiat execuția.

```
public interface Future<V>{

    boolean isCancelled();

    boolean isDone();

    boolean cancel(boolean mayInterruptIfRunning)

    V get() throws InterruptedException, ExecutionException;

    //alte metode ...

}
```

# Clasa Executors

- Clasa **Executors** conține metode statice care returnează obiecte de tip **ExecutorService**:
  - `newFixedThreadPool(int nThreads) : ExecutorService`
  - `newSingleThreadExecutor() : ExecutorService`
  - `newCachedThreadPool() : ExecutorService`
  - `newWorkStealingPool() : ExecutorService`

# Colecții concurente

- Colecții folosite în programarea concurrentă.
- Începând cu versiunea 1.5
- Interfața **BlockingQueue**:
  - **Coadă** - conține metode care așteaptă ca coadă să devină nevidă la scoaterea unui element, respectiv așteaptă eliberarea spațiului la adăugarea unui element.
  - Implementările BlockingQueue au fost proiectate și implementate pentru a fi folosite în situații de tip producător-consumator.
  - **ArrayBlockingQueue**, **LinkedBlockingQueue**, **PriorityBlockingQueue**, etc.
- Interfața **BlockingDeque**:
  - Extinde **BlockingQueue** și oferă suport pentru operații de tip FIFO și LIFO.
  - **LinkedBlockingDeque**
- Interfața **ConcurrentMap**:
  - Subinterfață a **java.util.Map**
  - **ConcurrentHashMap**, **ConcurrentSkipListMap**.

# Exemplu BlockingQueue

- Producător-Consumator simplu cu BlockingQueue

//ambele threaduri au referință la obiectul *messages*

```
//initializarea
private BlockingQueue<String> messages=new LinkedBlockingQueue<String>();

//Producator
try {
    messages.put(message);
} catch (InterruptedException e) {
    e.printStackTrace();
}

//Consumator
String message = messages.take();
```

# Actualizare GUI

- Interfețele grafice (JavaFX, Swing) folosesc obiecte de tip Component.
- Aceste obiecte ***pot fi modificate (actualizate, șterse, etc) doar de threadul care le-a creat.***
- Nerespectarea acestei reguli are rezultate neașteptate sau aruncă excepții.

```
Platform.runLater(new Runnable() { //JavaFX
    @Override
    public void run() {
        //codul care modifica informatia de pe interfata grafica
        label.setText("New text ...");
    }
});
```

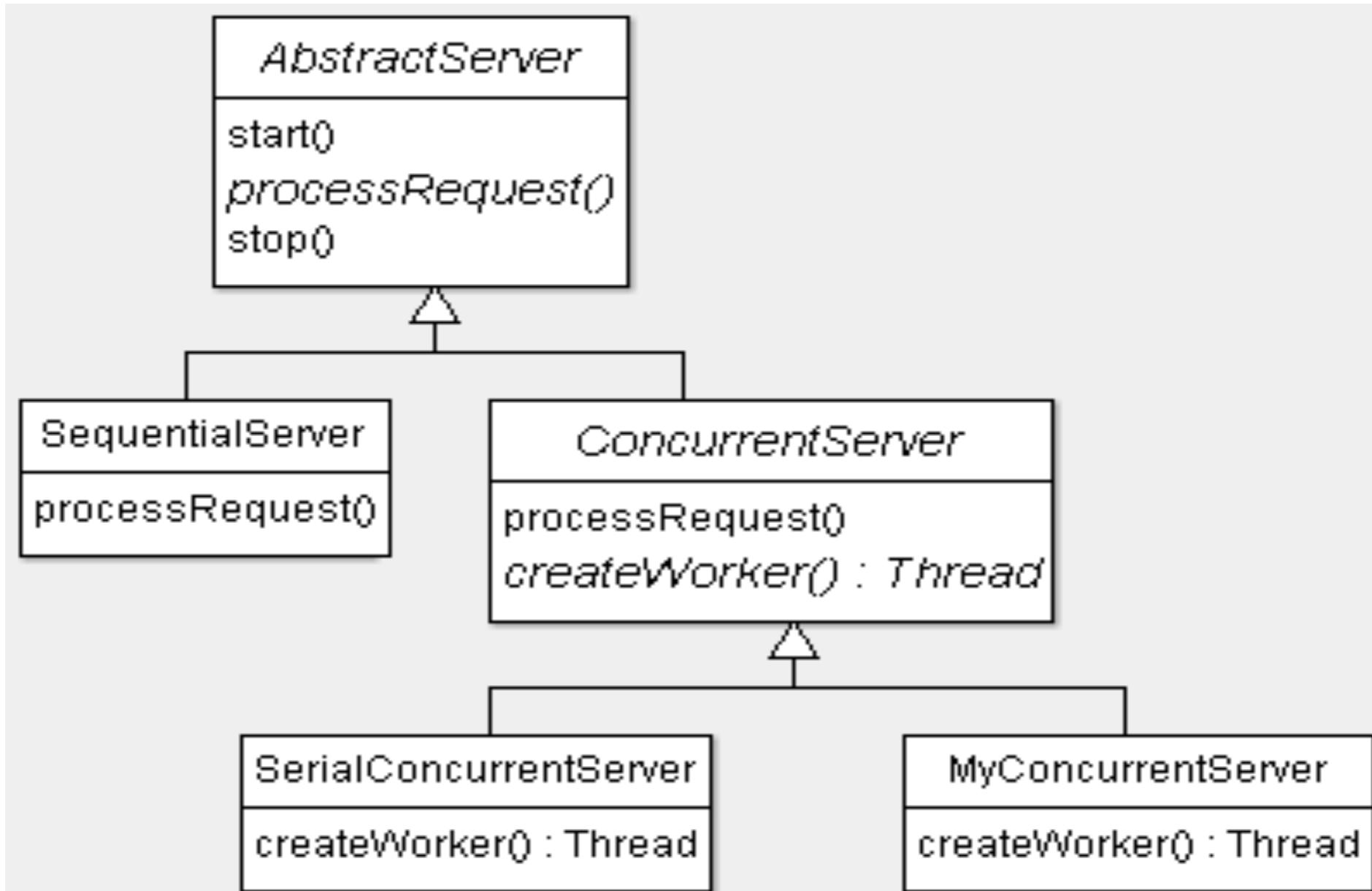
//sau, folosind funcții lambda

```
Platform.runLater(() -> { //JavaFX
    //codul care modifica informatia de pe interfata grafica
    label.setText("New text ...");
});
```

# Exemplu Java

- O aplicație simplă client/server:
  - Serverul așteaptă conexiuni.
  - Clientul se conectează la server și îi trimit un text.
  - Serverul returnează textul scris cu litere mari, la care adaugă data și ora la care a fost primit textul.

# Server Template



# Mini-Chat

- Proiectare (diagrame)
- Implementare Java

# Medii de proiectare și programare

2022-2023

Curs 6

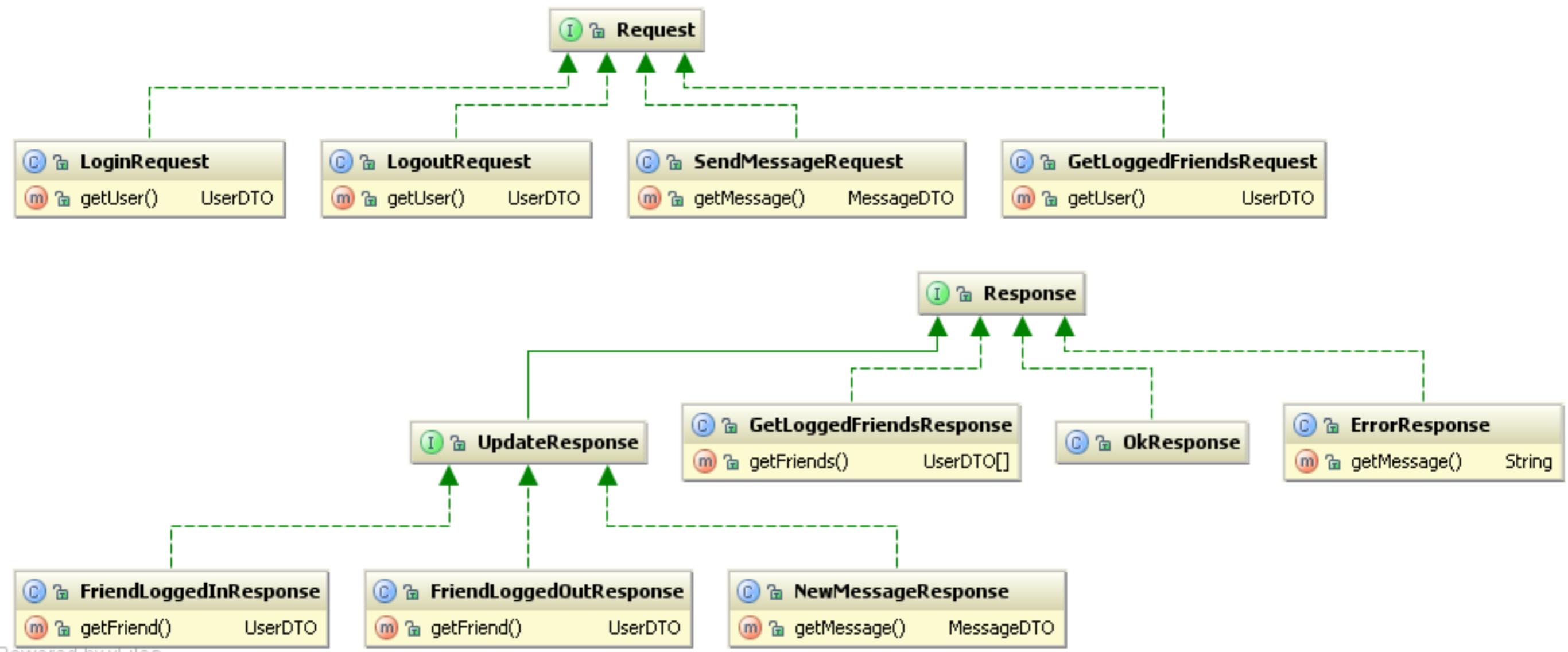
# Continut curs 6

- Exemplu Mini-Chat (networking - Java)
- Networking si threading in C#
- Exemplu Mini-Chat (networking - C#)

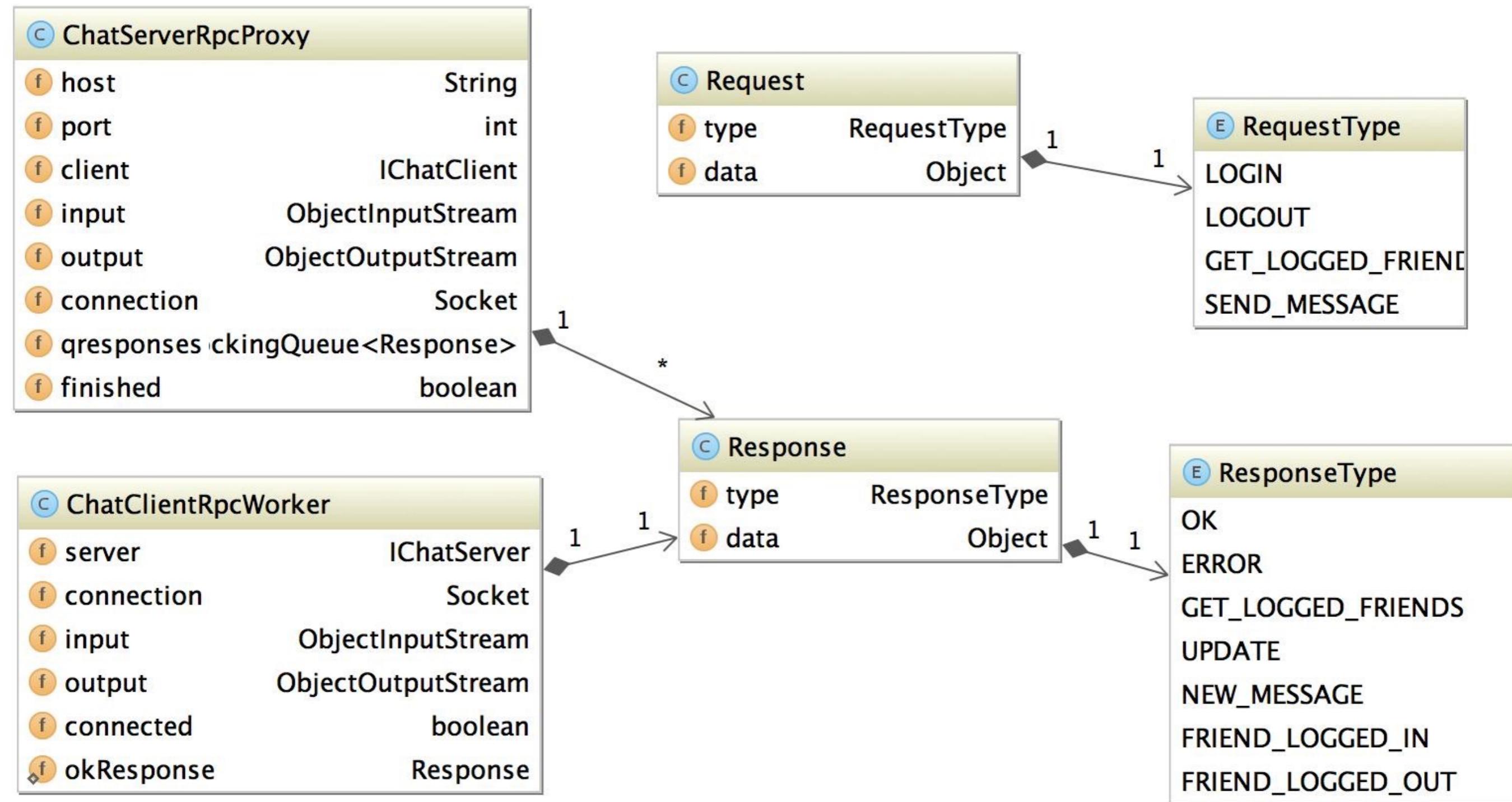
# Mini-Chat

- Proiectare (diagrame)
- Implementare Java

# Mini-chat Object Protocol



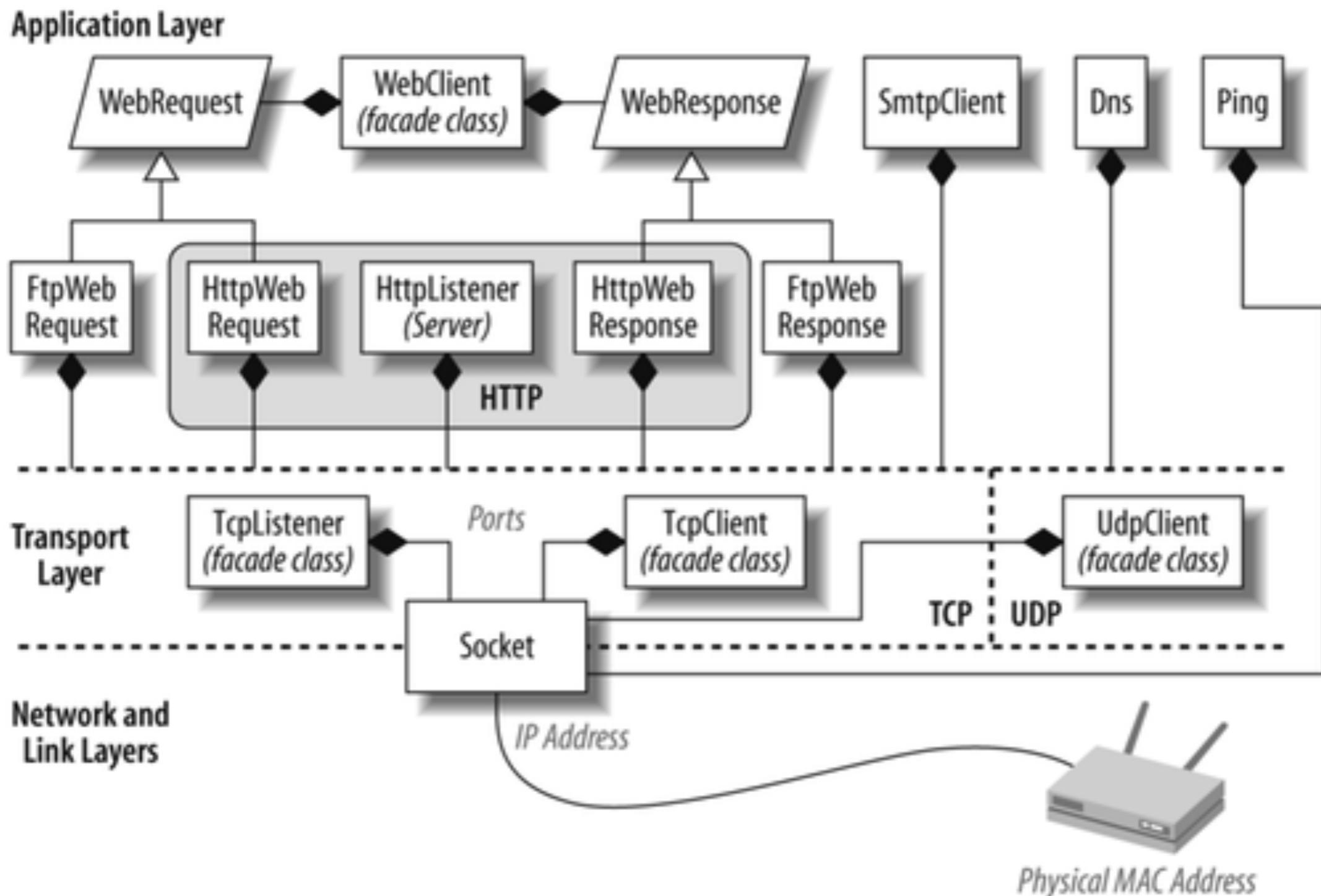
# Mini-chat Rpc Protocol



# Networking în C#

- .NET conține clase pentru comunicarea prin rețea folosind protocoale standard cum ar fi HTTP, TCP/IP și FTP.
- Spațiul de nume **System.Net.\***:
  - **WebClient** fațade pentru operații simple de download/upload folosind HTTP sau FTP.
  - **WebRequest** și **WebResponse** pentru operații HTTP și FTP complexe.
  - **HttpListener** pentru implementarea unui HTTP server.
  - **SmtpClient** pentru construirea și trimitera mesajelor folosind SMTP.
  - **TcpClient**, **UdpClient**, **TcpListener** și **Socket** pentru acces direct la nivelul rețea.

# Networking in C#



# Networking în C#

- Clasa `IPAddress` din `System.Net` reprezintă o adresă IPv4 (32 bits) sau IPv6 (128 bits).

```
IPAddress a1 = new IPAddress (new byte[] { 172, 30, 106, 5 }) ;  
IPAddress a2 = IPAddress.Parse ("172.30.106.5") ;  
IPAddress a3 = IPAddress.Parse  
(":[3EA0:FFFF:198A:E4A3:4FF2:54fA:41BC:8D31]"); //IPv6
```

- O asociere între o adresă IP și un port este reprezentată folosind clasa `IPEndPoint`:

```
IPAddress a = IPAddress.Parse ("172.30.106.5") ;  
IPEndPoint ep = new IPPEndPoint (a, 55555); // Port 55555  
Console.WriteLine (ep.ToString()); // 172.30.106.5:55555
```

- Porturile: 1 – 65535.
- Porturile dintre 49152 și 65535 nu sunt rezervate oficial.

# System.Net.Sockets Namespace

- Clasele **TcpClient**, **TcpListener** și **UdpClient** încapsulează detaliile creării conexiunilor de tip TCP și UDP.
- **Socket** implementează interfața Berkeley socket.
- **SocketException** excepția aruncată când apare o eroare la comunicarea prin socket.
- **NetworkStream** streamul folosit pentru comunicarea prin rețea.

# TcpListener

- TCP server:

```
TcpListener listener = new TcpListener (<ip address>, port);  
listener.Start();  
while (keepProcessingRequests)  
    using (TcpClient c = listener.AcceptTcpClient( ))  
        using (NetworkStream n = c.GetStream( )) {  
            // Read and write to the network stream...  
        }  
listener.Stop();
```

- **TcpListener** necesită adresa IP la care va aștepta conexiunile clientilor (dacă calculatorul are două sau mai multe plăci de rețea).
  - **IPAddress.Any** ascultă pe toate adresele IP locale (sau singura).
  - **AcceptTcpClient** blochează execuția până când se conectează un client.

# TcpClient

- Client Tcp:

```
using (TcpClient client = new TcpClient (<address>, port))  
  
    using (NetworkStream n = client.GetStream( ))  
  
    {  
  
        // Read and write to the network stream...  
  
    }
```

- **TcpClient** încearcă crearea conexiunii în momentul creării obiectului folosind adresa IP și portul specificate.
- Constructorul blochează execuția pâna la stabilirea conexiunii.

# NetworkStream

- **NetworkStream** comunicare ***bidirectională*** pentru transmiterea și recepționarea datelor după stabilirea unei conexiuni.
- Methods:
  - **Read**
  - **Close**
  - **Write**
  - **Seek**
  - **Flush**
- Properties:
  - **CanRead**, **CanWrite**
  - **Socket**
  - **DataAvailable**
  - **Length**

# Threading în C#

- Spatiul de nume **System.Threading** clase și interfețe pentru programarea concurentă:
  - Clasa **Thread**.
  - Delegate: **ThreadStart**, **ParameterizedThreadStart**.
  - Sincronizare: **lock**, **Monitor**, **Mutex**, **Semaphore**, **EventHandles**.
- **Delegates**: reprezintă metoda executată de un thread.

```
public delegate void ThreadStart();
```

```
public delegate void ParameterizedThreadStart(Object obj);
```

- Clasa **Thread**: crearea unui thread, setarea priorității, obținerea informațiilor despre statusul unui thread.

```
public Thread(ThreadStart start);
```

```
public Thread(ParameterizedThreadStart start);
```

# Threading in C#

```
class Program  {
    static void Main(string[] args)  {
        Worker worker=new Worker();
        Thread t1=new Thread(new ParameterizedThreadStart(static_run));
        Thread t2=new Thread(new ThreadStart(worker.run));
        t1.Start("a");
        t2.Start();
    }
    static void static_run(Object data)  {
        for(int i=0;i<26;i++)  { Console.WriteLine("{0} ",data);  }
    }
    class Worker {
        public void run()  {
            for(int i=0;i<26;i++)      Console.WriteLine("{0} ",i);
        }
    }
//a a a a a a a a a a 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
  19 20 21 22 23 24 25 a
```

# Sincronizarea threadurilor

- Diferite tipuri:
  - *Blocarea exclusivă*: doar un singur thread poate executa o porțiune de cod la un moment dat.
    - `lock`, `Mutex`, and `SpinLock`.
  - *Blocarea nonexclusivă*: limitarea concurenței.
    - `Semaphore` and `ReaderWriterLock`.
  - *Semnalizarea*: un thread poate bloca execuția până la primirea unei notificării de la unul sau mai multe threaduri.
    - `ManualResetEvent`, `AutoResetEvent`, `CountdownEvent` și `Barrier`.

# Sincronizarea threadurilor – Blocarea

- Instrucțunea **lock**:

```
lock(locker_obj) {  
    //code to execute  
}
```

- **locker\_obj** - tip referință.
- Doar un singur thread poate obține accesul la un moment dat. Dacă mai multe threaduri încearcă să obțină accesul, ele sunt puse într-o coadă și primesc accesul pe baza regulii “primul venit-primul servit”.
- Dacă un alt thread a obținut deja accesul, threadul curent nu își continuă execuția până nu obține accesul.

# Sincronizarea threadurilor - Signaling

- *Event wait handles* - construcții simple pentru semnalizare:
  - **EventWaitHandle**- reprezintă un eveniment pentru sincronizarea threadurilor. Unul sau mai multe threaduri blochează execuția folosind un **EventWaitHandle** până când un alt thread apelează metoda **Set** permitând execuția unuia sau mai multor threaduri aflate în așteptare.
  - **AutoResetEvent**, **ManualResetEvent**
  - **CountdownEvent** (Framework 4.0)
- **AutoResetEvent** notifică un thread aflat în așteptare de apariția unui eveniment (doar un singur thread).
  - **Set()** - eliberează un thread aflat în așteptare
  - **WaitOne()** - threadul așteaptă apariția unui eveniment

Observații:

1. Dacă **Set** este apelată când nici un thread nu se află în așteptare, handle -ul așteaptă până când un thread apelează metoda **WaitOne**.
2. Apelarea metodei **Set** de mai multe ori când nici un thread nu este în așteptare nu va permite mai multor threaduri obținerea accesului când apelează metoda **WaitOne**.

# Sincronizarea threadurilor - Signaling

- **ManualResetEvent** (asemănător **AutoResetEvent**) – notifică toate threadurile aflate în aşteptare la apariția unui eveniment.

- Crearea unui event wait handle:

- Constructori:

```
AutoResetEvent waitA=new AutoResetEvent(false);  
AutoResetEvent waitA=new AutoResetEvent(true); //calls Set
```

```
ManualResetEvent waitM=new ManualResetEvent(false);
```

- Clasa **EventWaitHandle**

```
var auto = new EventWaitHandle (false, EventResetMode.AutoReset);  
var manual = new EventWaitHandle (false, EventResetMode.ManualReset);
```

- Distrugerea unui wait handle:

- Apelul metodei **Close** pentru eliberarea resurselor sistemului de operare.
  - Ștergerea referințelor pentru a permite garbage collector-ului distrugerea obiectului.

# Exemplu Signaling

```
class WaitHandleExample
{
    static EventWaitHandle waitHandle = new AutoResetEvent (false);
    static void Main()
    {
        new Thread (Worker).Start();
        Thread.Sleep (1000);      // Pause for a second...
        waitHandle.Set();        // Wake up the Worker.
    }
    static void Worker()
    {
        Console.WriteLine ("Waiting...");
        waitHandle.WaitOne();    // Wait for notification
        Console.WriteLine ("Notified");
    }
}
```

# Task-uri C#

- ***Limitările threadurilor:***
  - Se pot transmite ușor date unui thread, dar **nu se poate obține la fel de ușor rezultatul execuției unui thread.**
  - Dacă **execuția threadului aruncă o excepție**, tratarea excepției și retransmiterea ei este **mai dificil de implementat**.
  - Nu se poate seta ca un thread să execute altceva când și-a încheiat execuția.
- Un **Task C#** reprezintă **o operație concurentă care poate fi (sau nu) executată folosind threaduri**.
  - Taskurile pot fi compuse.
  - Pot folosi un container de threaduri pentru a reduce timpul necesar pornirii execuției.
- Tipul **Task** a fost introdus începând cu Framework 4.0 ca facând parte din biblioteca pentru programare paralelă.
- **System.Threading.Tasks** namespace.

# Pornirea execuției unui Task

- Framework 4.5 - Metoda statică **Task.Run** (pornirea execuției unui task folosind threaduri) - parametru de tip **Action** delegate:

```
Task.Run (() => Console.WriteLine ("Ana"));
```

- Framework 4.0 - Metoda statică **Task.Factory.StartNew**:

```
Task.Factory.StartNew () =>Console.WriteLine ("Ana") );
```

- Implicit, taskurile folosesc threaduri din containere deja create.
- Folosirea metodei **Task.Run** - similară cu execuția explicită folosind threaduri:

```
new Thread (() => Console.WriteLine ("Ana")) .Start();
```

- **Task.Run** returnează un obiect **Task** care poate fi folosit pentru monitorizarea progresului.
- Nu este necesară apelarea metodei **start**.

# Obținerea rezultatului execuției

- **Task<TResult>** subclasă a clasei **Task** care permite returnarea rezultatului execuției.
- **Task<TResult>** poate fi obținut apelând **Task.Run** folosind un delegate de tip **Func<TResult>** (sau o expresie lambda compatibilă).
- Rezultatul poate fi obținut folosind proprietatea **Result**.
- Dacă taskul nu și-a încheiat execuția, apelul proprietății **Result** va bloca execuția threadului curent până la terminarea execuției taskului:

```
Task<int> task = Task.Run(()=>{int x=2; return 2*x; });

int result = task.Result; // Blocks if not already finished

Console.WriteLine(result); // 4
```

# Taskuri și excepții

- Taskurile propagă excepțiile (threadurile nu).
- Dacă un task aruncă o excepție, excepția este rearuncată către codul care apelează metoda `Wait()` a clasei Task sau care apelează proprietatea `Result` a clasei `Task<TResult>`:
- Excepția va fi inclusă într-o excepție de tip `AggregateException` de către CLR:

```
// Start a Task that throws a NullReferenceException:  
  
Task task = Task.Run (() => { throw null; });  
  
try{  
  
    task.Wait();  
  
}catch (AggregateException aex){  
  
    if (aex.InnerException is NullReferenceException)  
  
        Console.WriteLine ("Null!");  
  
    else throw;  
  
}
```

# Actualizare GUI

- Interfețele grafice de tip Windows Forms folosesc obiecte de tip **Control**.
- Aceste obiecte pot fi modificate (actualizate, șterse, etc) doar de threadul care le-a creat.
- Nerespectarea acestei reguli are rezultate neașteptate sau aruncă excepții.
- Dacă se dorește apelarea unui membru (metoda, atribut, proprietate) a obiectului X creat într-un thread Y, cererea trebuie transmisă threadului Y (folosind metoda **Invoke** sau **BeginInvoke** a obiectului X).

# Actualizare GUI

- **Invoke** și **BeginInvoke** au un parametru de tip delegate care referă metoda corespunzătoare obiectului de tip Control care se dorește a se executa.
  - **Invoke** execuție sincronă: execuția apelantului este blocată până la actualizarea controlului.
  - **BeginInvoke** execuție asincronă: execuția apelantului continuă, iar cererea este pusă într-o coadă (corespunzătoare evenimentelor de la tastatură, mouse, etc) și se va executa ulterior.

# Exemplu

```
//1. definirea unei metode pentru actualizarea unui ListBox
private void updateListBox(ListBox listBox, IList<String> newData) {
    listBox.DataSource = null;
    listBox.DataSource = newData;
}

//2. definirea unui delegate care va fi apelat de GUI Thread
public delegate void UpdateListBoxCallback(ListBox list, IList<String> data);

//3. în celălalt thread se transmite metoda care actualizează ListBox:
list.Invoke(new UpdateListBoxCallback(this.updateListBox), new Object[] {list, data});
or
list.BeginInvoke(new UpdateListBoxCallback(this.updateListBox), new Object[] {list, data});
```

# Exemplu C#

- O aplicație simplă client/server:
  - Serverul așteaptă conexiuni.
  - Clientul se conectează la server și îi trimite un text.
  - Serverul returnează textul scris cu litere mari, la care adaugă data și ora la care a fost primit textul.

# Exemplu C#

- Mini-chat C#

# Medii de proiectare și programare

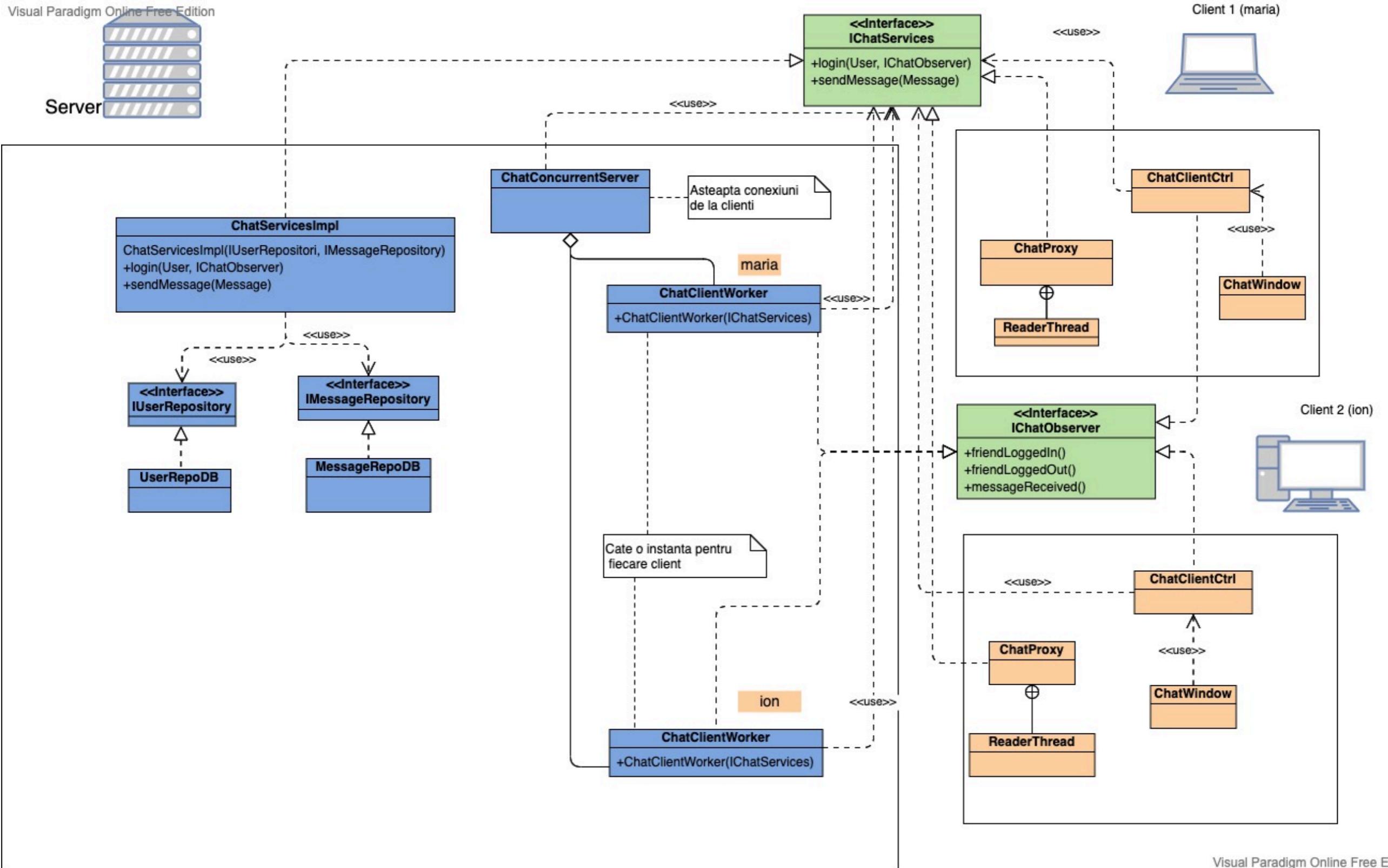
2022-2023

Curs 7

# Continut curs 7

- Remote Procedure Call
- Aplicații distribuite cross-platform

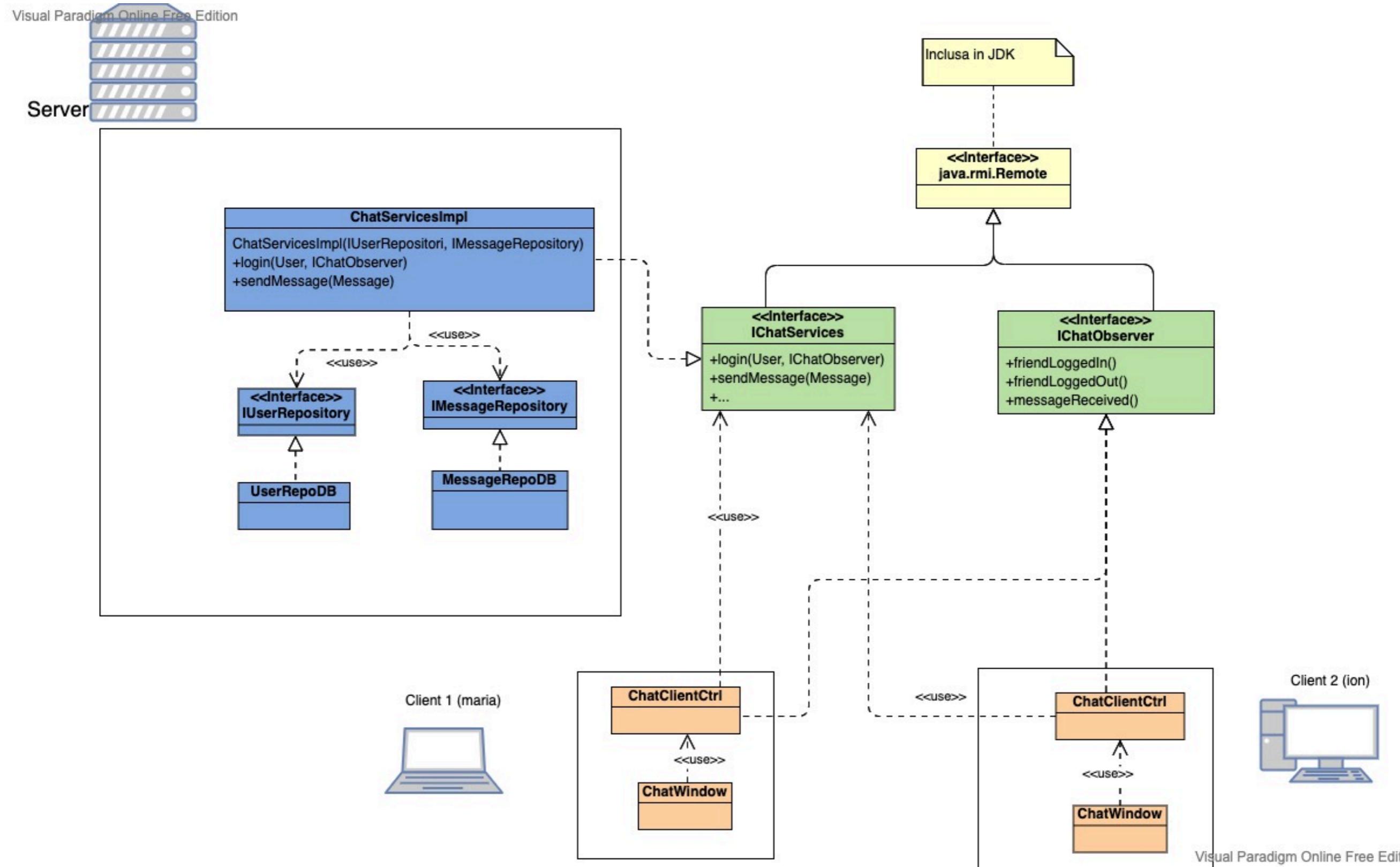
# Networking



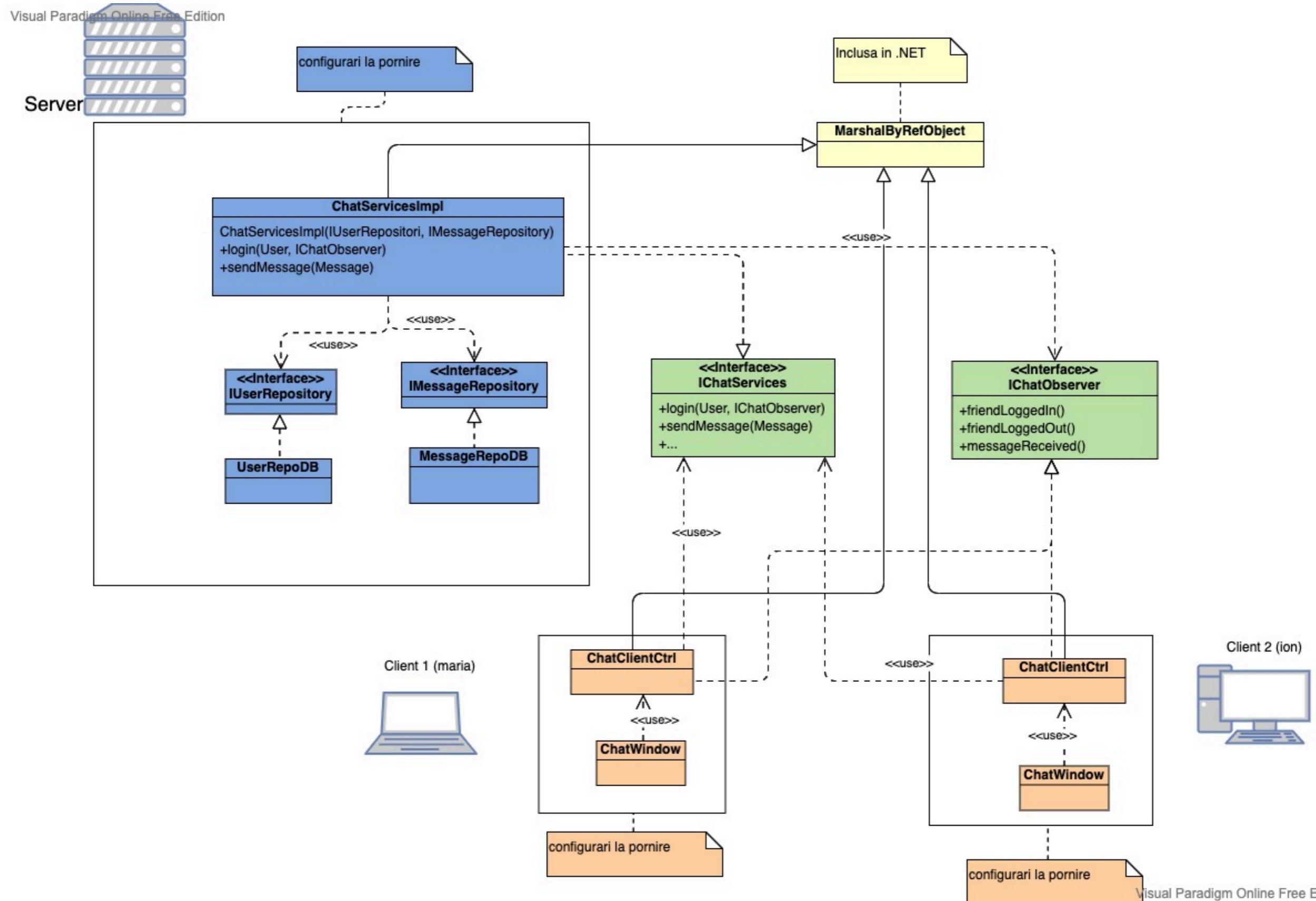
# Remote Procedure Call

- **Apelul procedurilor la distanță** (eng. *Remote procedure call - RPC*) este o **tehnologie de comunicare între procese** care permite unei aplicații să inițieze execuția unei subrutine sau a unei proceduri în alt spațiu de adrese, **fără ca programatorul să scrie explicit codul corespunzător interacțiunii dintre procese**.
- Programatorul scrie aproximativ același cod indiferent dacă apelează o rutină locală (din același proces) sau una la distanță (din alt proces, în alt spațiu de adrese).
- Când se folosește **paradigma orientată pe obiecte** RPC - apeluri la distanță sau **apelul metodelor la distanță**.
- RPC este adesea folosit pentru implementarea aplicațiilor client-server.
- Un **apel la distanță** este inițiat de **client** prin trimitera unei cereri către un server la distanță pentru execuția unei proceduri cu parametrii dați. **Serverul trimitе un răspuns clientului**, iar aplicația își continuă execuția.
- Cât timp **serverul procesează cererea clientului**, **execuția clientului este blocată** (așteaptă până când serverul a terminat procesarea cererii).

# Java RMI



# .NET Remoting



# Remote Procedure Call

## Java RMI, .NET Remoting

- **Avantaje**

- Dezvoltarea ușoară a aplicațiilor client-server
- Ascunderea detaliilor de implementare pentru comunicarea client-server.

- **Dezavantaje**

- Aplicațiile se pot dezvolta într-un singur limbaj (Java/C#/etc)
- Dacă apar versiuni noi pentru interfețe (servicii), trebuie modificați toți clienții.

# Aplicații distribuite cross-platform

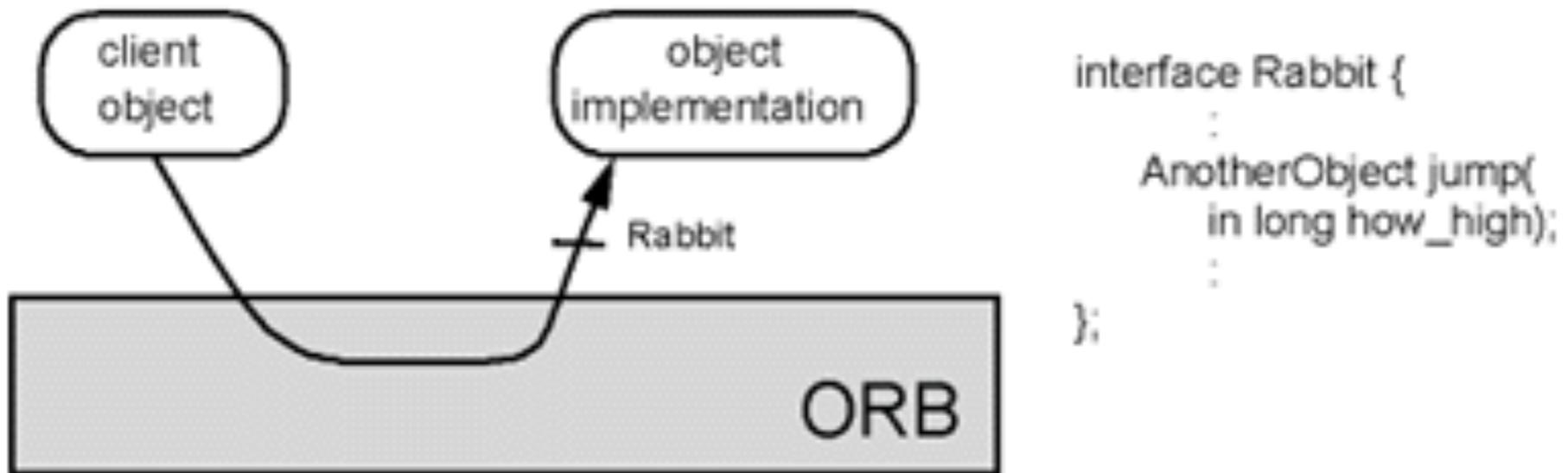
- CORBA
- Protobuf
- gRPC
- Thrift

# CORBA

- A fost propusă de OMG (Object Management Group) înainte de anul 2000.
- CORBA (Common Object Request Broker Architecture) este o arhitectură standard pentru sisteme distribuite orientate pe obiect.
- Produsele CORBA oferă un framework pentru dezvoltarea și execuția aplicațiilor distribuite.
- Permite unei colecții de obiecte distribuite **heterogene** să interacționeze.
- CORBA definește arhitectura obiectelor distribuite.
- Paradigma de bază CORBA este de cerere a unor servicii oferite de un obiect distribuit (remote).
- Serviciile oferite de un obiect sunt specificate de interfața acestuia.
- **Interfețele** sunt definite folosind limbajul Interface Definition Language (IDL) definit de OMG.
- Obiectele distribuite (remote) sunt identificate prin referințe, de tipul interfețelor IDL.

# Arhitectura CORBA

- Un client păstrează o referință către un obiect distribuit (remote).
- Un proces numit Object Request Broker (ORB), trimite cererea obiectului și returnează rezultatul primit clientului.



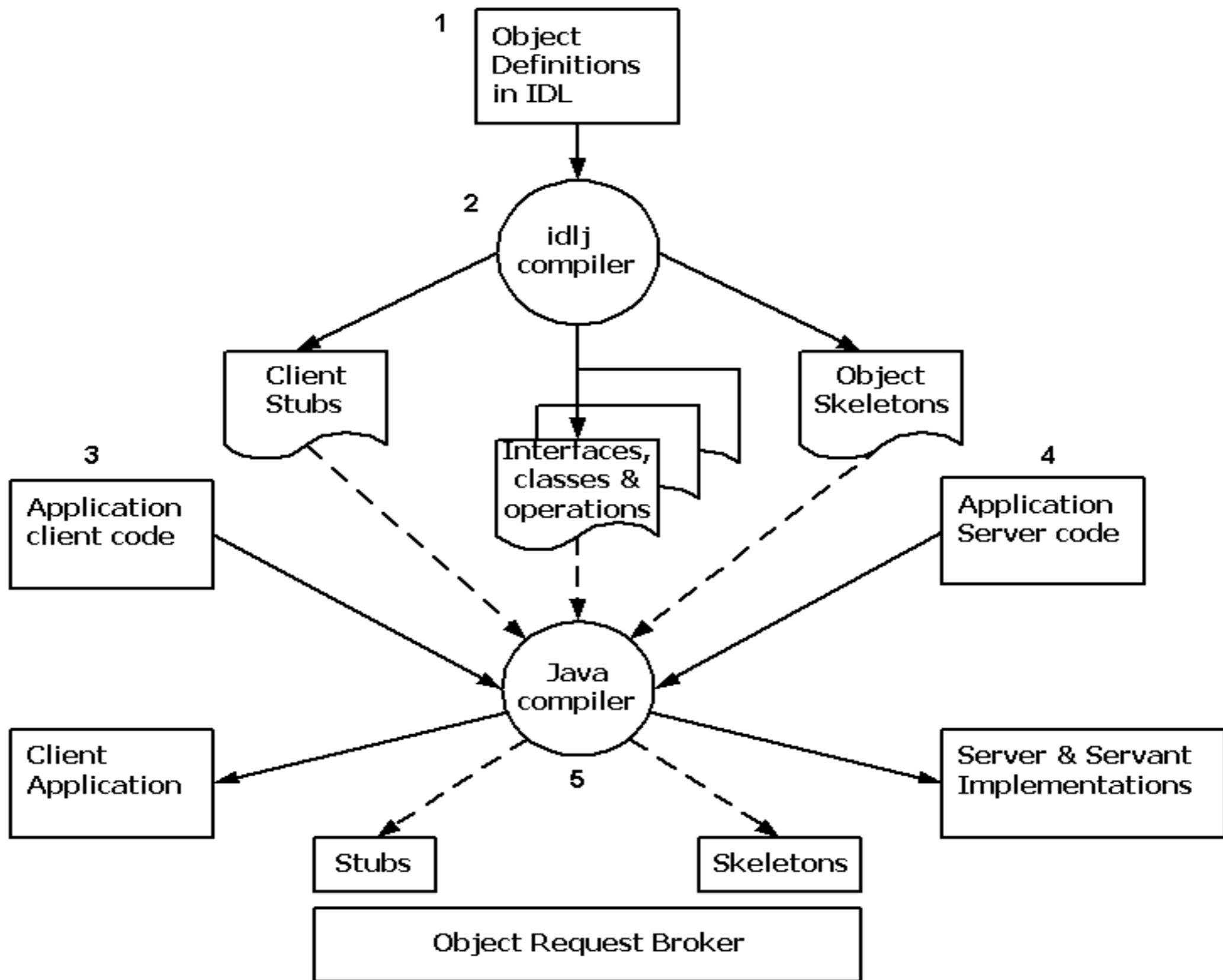
# Interfețe IDL

- Limbajul *Interface Definition Language (IDL)* propus de OMG permite specificarea interfețelor obiectelor remote.
- **Interfață** unui obiect remote **indică metodele** ce pot fi apelate la distanță, dar nu cum sunt implementate aceste metode.
- În IDL nu se poate declara starea unui obiect sau algoritmi.
- Implementarea unui obiect CORBA este furnizată într-un limbaj de programare (Java, C++, etc). O interfață specifică contractul dintre codul care folosește obiectul și codul care implementează obiectul. Clientii depind doar de interfață.
- Interfețele IDL nu depind de un anumit limbaj de programare. IDL definește transformări (eng. *language bindings*) pentru diferite limbi de programare.
- Se permite astfel ca pentru obiectul remote să fie ales cel mai potrivit limbaj de programare, și, de asemenea, permite clientilor să aleagă cel mai potrivit limbaj (care poate dифeри de cel folosit pentru implementarea obiectului remote).
- Transformări pentru C, C++, Java, Ada, Smalltalk, etc.

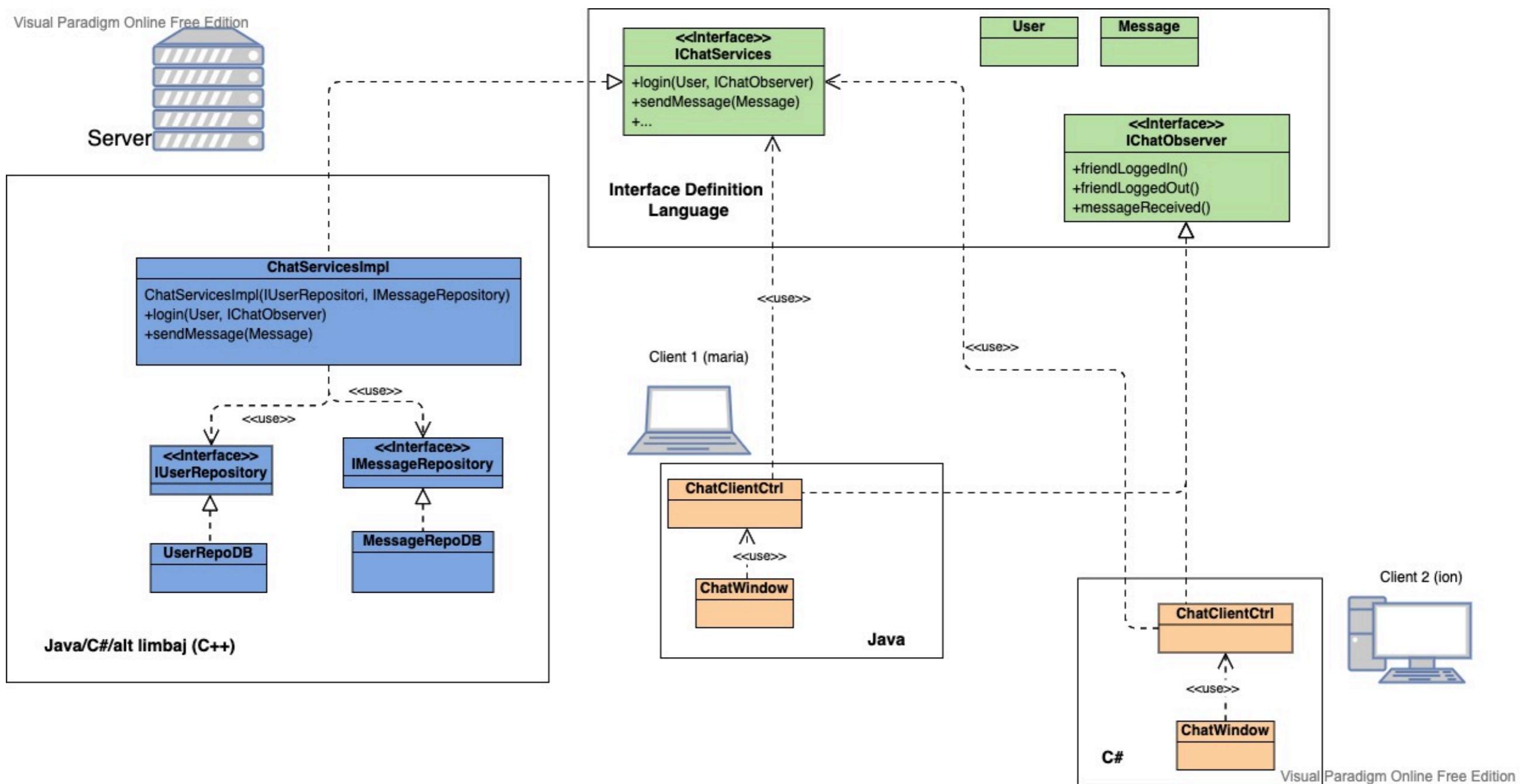
# Exemplu IDL

```
module BookShop {  
    struct Book {  
        double price;  
        String title;  
        String author  
    };  
    exception Unknown{};  
    interface Library {  
        Book getBook(in string title) raises(Unknown);  
        readonly attribute string name;  
    };  
    interface BookFactory {  
        Book create_book( in string title, in string author, in double  
        price);  
    };  
};
```

# CORBA și Java



# CORBA și Java/C#/C++



# Protocol Buffers (Protobuf)

- Reprezintă o modalitate independentă de limbaj și platformă de a serializa structuri de date folosite în protocoale de comunicație, stocarea datelor, etc.
- Sunt flexible, eficiente și au un mecanism automat de serializare a datelor structurate (similar cu XML), dar mecanismul este mai rapid, mai simplu și mai ușor de folosit.
- Se definește o singura dată modul de structurare a datelor (folosind un IDL), iar apoi se folosește un compilator special (*protoc*) care generează cod ce permite scrierea/citirea structurilor de date în/din o varietate de streamuri de date și folosind diferite limbi de programare.
- Este permisă modificarea structurii datelor fără a provoca apariția erorilor în programele dezvoltate folosind vechea structură de date.
- Specificarea structurii datelor se definește folosind tipuri de mesaje protocol buffers și sunt salvate în fișiere **.proto**.
- Fiecare mesaj protocol buffer este o înregistrare logică mică de informații, conținând o serie de perechi cheie-valoare.

# Protocol Buffers vs XML

- Protocol buffers au avantaje asupra XML în privința serializării datelor:
  - sunt mai simple
  - sunt de 3 până la de 10 ori mai mici ca și dimensiune
  - sunt de 20 până la de 100 de ori mai rapide
  - sunt mai puțin neclare (ambigue)
  - clasele generate automat pentru accesarea datelor sunt mai ușor de folosit în limbajul de programare
- Protocol buffers nu sunt totdeauna o soluție mai bună decât XML:
  - Protocol buffers nu sunt potrivite pentru modelarea unui text ce folosește markup (ex. HTML), deoarece nu permite ușor imbricarea structurii datelor cu text.
  - XML este ușor de citit de oameni (eng. *human-readable*) și de editat (eng. *human-editable*);
  - Protocol buffers nu pot fi citite de oameni și editate
  - XML se descrie pe sine.
  - Un protocol buffer are sens doar dacă avem acces și la definiția mesajului (fișierul .proto).

# Protocol Buffers vs XML

//XML

```
<person>  
  <name>John Doe</name>  
  <email>jdoe@example.com</email>  
</person>
```

//Reprezentarea textuală a unui Protobuf (nu octetii serializați)

```
person {  
  name: "John Doe"  
  email: "jdoe@example.com"  
}
```

# Protocol Buffers vs XML

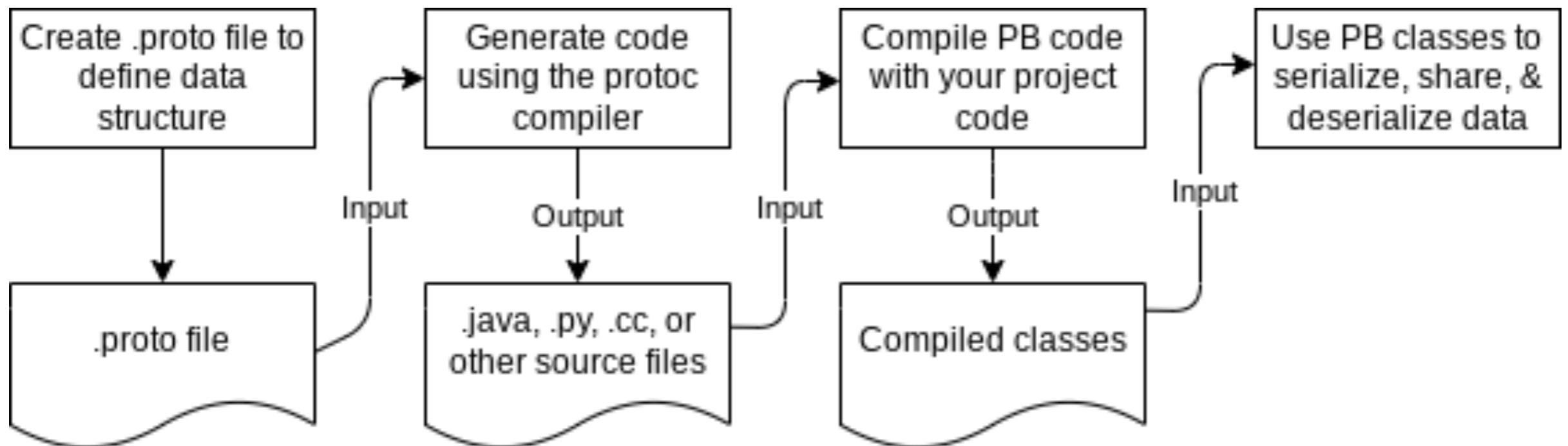
//Parsarea unui XML

```
<person>
    <name>John Doe</name>
    <email>jdoe@example.com</email>
</person>
//Java
person.getElementsByTagName("name").item(0).getNodeValue()
person.getElementsByTagName("email").item(0).getNodeValue();
```

//Parsarea unui mesaj Protobuf

```
person {
    name: "John Doe"
    email: "jdoe@example.com"
}
System.out.println("Name: " + person.getName());
System.out.println("E-mail: "+ person.getEmail());
```

# Protocol Buffers Workflow\*



\*<https://developers.google.com/protocol-buffers/docs/overview>

# Protocol Buffers vers. 2 – Exemplu

```
syntax="proto2"

message Person {
    required string name = 1;
    required int32 id = 2;
    optional string email = 3;
    enum PhoneType {
        MOBILE = 0;
        HOME = 1;
        WORK = 2;
    }
    message PhoneNumber {
        required string number = 1;
        optional PhoneType type = 2 [default = HOME];
    }
    repeated PhoneNumber phone = 4;
}
```

# Definirea unui nou tip de mesaj - vers. 3

- Tipurile mesajelor sunt salvate într-un fisier **.proto**

```
syntax="proto3";  
  
message SearchRequest {  
    string query = 1;  
    int32 page_number = 2;  
    int32 result_per_page = 3;  
}
```

- Fiecare câmp are un nume, tip și un tag asociat.
- **Tipul** poate fi:
  - scalar (double, float, int32, int64, bool, string, bytes, etc).
  - tip compus (enumerare)
  - tipul unui alt mesaj

# Definirea unei enumerări - vers. 3

- **Enumerare**: un câmp de tip enumerare poate avea ca și valoare doar una dintre constantele specificate.
- Fiecare enumerare trebuie să definească o constantă care are asociată valoarea zero ca primă constantă din enumerare.
- Aceasta valoare va fi folosită ca și valoarea implicită.
- Două constante pot avea aceeași valoare (dacă se setează opțiunea allow\_alias).

```
enum Corpus {  
    option allow_alias = true;  
    UNIVERSAL = 0;  
    WEB = 1;  
    IMAGES = 2;  
    LOCAL = 3;  
    NEWS = 4;  
    PRODUCTS = 5;  
    VIDEO = 5;  
}  
Corpus corpus = 4;
```

# Reguli pentru specificarea câmpurilor - vers. 3

- Un câmp poate fi:
  - *singular*: într-un mesaj bine format acest câmp poate să apară cel mult o dată (zero sau unu).
  - *repeated*: acest câmp poate să apară de ori câte ori (inclusiv zero) într-un mesaj bine format. Ordinea valorilor care se repetă se va păstra.
- Dacă un mesaj nu conține nici o valoare pentru un câmp singular, la parsare câmpul va primi valoarea implicită corespunzătoare tipului său.

```
message Result {  
    string url = 1;  
    string title = 2;  
    repeated string snippets = 3;  
}
```

# Atribuirea tagurilor

- Fiecare câmp din definiția unui mesaj are un tag numeric unic asociat.
- Aceste taguri sunt folosite pentru identificarea câmpurilor în formatul binar și nu ar trebui schimbate după începerea folosirii lui.
- Tagurile cu valori între 1 și 15 ocupă un octet (incluzând numărul de identificare și tipul câmpului).
- Tagurile cu valori între 16 - 2047 ocupă 2 octeți.
- Tagurile cu valori între 1 și 15 ar trebui folosite pentru câmpurile folosite cele mai des din mesaj.
- Tagurile pot avea valori între 1 și 536,870,911.
- Numerele între 19000 și 19999 sunt rezervate pentru implementarea Protocol Buffers - compilatorul protoc va genera o eroare dacă se încearcă folosirea lor.

# Tipuri multiple de mesaje

- Mai multe tipuri de mesaje pot fi definite într-un singur fișier .proto.

```
message SearchRequest {  
    string query = 1;  
    int32 page_number = 2;  
    int32 result_per_page = 3;  
}
```

```
message SearchResponse {  
    ...  
}
```

# Câmpuri rezervate

- Dacă definiția tipului unui mesaj se modifică ulterior prin ștergerea sau comentarea unui câmp, utilizatorii pot refolosi tag-ul asociat câmpului respectiv.
- Pot apărea erori când se folosesc mesaje salvate cu versiuni mai vechi ale aceluiasi tip de mesaj.
- Se poate specifica că anumite taguri sau nume de câmpuri sunt rezervate și nu mai pot fi refolosite ulterior, folosind instrucțiunea **reserved**.
- În aceeași instrucțiune **reserved** nu pot fi folosite taguri și nume de câmpuri.

```
message Foo {  
    int32 foo = 2; // câmpul foo  
  
    reserved 2, 15, 9 to 11;  
    reserved "foo", "bar";  
}
```

# Fisiere .proto multiple

- Într-un fișier .proto se pot folosi definițiile din alte fișiere .proto prin importarea lor.
- Importarea definițiilor din alt fișier .proto se face cu instrucțiunea import la începutul fișierului (după instrucțiunea syntax):

```
import "myproject/other_protos.proto";
```

- Compilatorul caută fișierele importate în lista de directoare specificată la linia de comandă ca și parametru al compilatorului folosind opțiunea **-I sau --proto\_path**.
- Dacă nu a fost specificată această opțiune, compilatorul va căuta în directorul în care a fost rulat compilatorul.

# Tipuri Nested

- Se pot defini tipuri de mesaje în interiorul altor tipuri de mesaje:

```
message SearchResponse {  
    message Result {  
        string url = 1;  
        string title = 2;  
        repeated string snippets = 3;  
    }  
    repeated Result result = 1;  
}
```

- Tipul nested se poate folosi din exterior respectând formatul **Parent.Type**:

```
message SomeOtherMessage {  
    SearchResponse.Result result = 1;  
}
```

# Tipul Any

- Tipul Any este folosit pentru a defini câmpuri pentru care nu știm/avem fișierul .proto corespunzător.
- Un câmp de tip Any va conține un mesaj necunoscut serializat într-un sir de octeți și un URL folosit ca și identificator global unic care va indica către definiția tipului mesajului.
- Pentru a folosi tipul Any trebuie importat fișierul *google/protobuf/any.proto*.

```
import "google/protobuf/any.proto";
```

```
message ErrorStatus {  
    string message = 1;  
    repeated google.protobuf.Any details = 2;  
}
```

URL implicit asociat unui mesaj definit într-un fișier .proto are formatul

**type.googleapis.com/packagename.messagename**

# Opțiunea oneof

- Este folosită când valoarea unui câmp poate fi cel mult una dintr-o mulțime finită de valori de tipuri diferite. Folosirea opțiunii reduce dimensiunea mesajului.
- Câmpurile Oneof sunt câmpuri normale, exceptând faptul că partajează aceeași zonă de memorie și cel mult un câmp poate avea asociată o valoare la un moment dat.
- Setarea valorii uneia dintre câmpurile marcate *oneof* duce automat la ștergerea valorii celorlalte câmpuri.
- Se poate determina care câmp din mulțimea specificată *oneof* are asociată o valoare (dacă există un astfel de câmp). (Dependent de limbaj)
- Nu se pot folosi câmpuri repeated în mulțimea câmpurilor specificate cu *oneof*.

```
message SampleMessage {  
    oneof test_oneof {  
        string name = 4;  
        SubMessage sub_message = 9;  
    }  
}
```

# Tipul Map

- Se pot defini dicționare ca și câmp al unui nou tip de mesaj.

```
map<key_type, value_type> map_field = N;
```

- **key\_type** poate fi orice tip *integral* (orice tip scalar exceptând tipurile reale și octeți) sau tipul string. **value\_type** poate fi orice tip.

- Observații:

- Câmpurile Map nu pot fi *repeated*.
- Ordinea serializării valorilor dintr-un dicționar este nespecificată, nu ne putem baza pe o anumită ordine a elementelor din dicționar.
- Dacă la parsare/deserializare există mai multe chei având aceeași valoare se păstrează ultima valoare întâlnită.

```
map<string, Project> projects = 3;
```

# Pachete

- Optional se poate specifica pachetul corespunzător tipurilor de mesaje definite într-un fișier .proto pentru a evita coliziuni de nume.

```
package foo.bar;  
message Open { ... }
```

- Numele pachetului poate fi utilizat ulterior la definirea tipului câmpurilor:

```
message Foo {  
    ...  
    foo.bar.Open open = 1;  
    ...  
}
```

- Modul în care specificarea unui pachet afectează codul generat depinde de limbaj:

- În Java, pachetul este folosit ca și pachet Java, dacă nu se specifică explicit un alt pachet folosind opțiunea `option java_package` în fișierul .proto.
- În C#, pachetul este folosit ca și spațiu de nume după transformarea la PascalCase, dacă nu se specifică explicit un alt pachet folosind opțiunea `csharp_namespace` în fișierul .proto. (Exemplu, câmpul Open va face parte din spațiul de nume Foo.Bar).

# Definirea serviciilor RPC

- Se pot defini servicii RPC într-un fișier .proto, iar compilatorul va genera codul corespunzător interfeței și stub-urilor (proxies) în limbajul ales.

```
service SearchService {  
    rpc Search (SearchRequest) returns (SearchResponse);  
}
```

- gRPC - tehnologie open-source cross-platform RPC dezvoltată de Google care folosește protocol buffers și permite generarea codului RPC direct din fișierul .proto dacă se folosește un plugin adițional pentru compilatorul protoc.
  - Există și alte tehnologii RPC bazate pe Protocol Buffers care pot fi utilizate pentru RPC

# Alte opțiuni

- Într-un fișier .proto se pot declara opțiuni individuale care nu modifică semnificația tipurilor de mesaje definite în fișier, dar influențează modul în care este generat codul sursă corespunzător unui anumit limbaj.
- Lista completă a opțiunilor disponibile: [google/protobuf/descriptor.proto](#).
- Cele mai folosite opțiuni:
  - **java\_package** (nivel fișier): Numele pachetului care va fi folosit la generarea claselor Java. Dacă nu se specifică explicit, numele implicit al pachetului va fi cel specificat folosind opțiunea package.

```
option java_package = "com.example.foo";
```
  - **java\_outer\_classname** (nivel fișier): Numele clasei folosite ca și clasă exterioară pentru tipurile mesajelor. Dacă nu se specifică explicit opțiunea **java\_outer\_classname**, numele clasei se va construi prin transformarea numelui fișierului .proto la camel-case (**foo\_bar.proto** va deveni **FooBar.java**).

```
option java_outer_classname = "FooBar";
```

# Alte opțiuni

- **java\_multiple\_files** (nivel fișier): Determină generarea tipurilor mesajelor, a enumerărilor și a serviciilor la nivel de pachet (nu într-o clasă exterioară) **în fișiere separate**.

```
option java_multiple_files = true;
```

- Opțiuni C#

```
option (google.protobuf.csharp_file_options).namespace =
"MyCompany.MyProject";
```

```
option (google.protobuf.csharp_file_options).umbrella_classname =
"ProjectProtos";
```

# Generarea claselor Java/C#

- Generarea claselor Java, C#, Python, C++, etc. se face folosind compilatorul **protoc** asupra fișierului .proto.

```
protoc --proto_path=IMPORT_PATH --java_out=DST_DIR path/to/  
file(s).proto
```

```
protoc --proto_path=IMPORT_PATH --csharp_out=DST_DIR path/to/  
file(s).proto
```

- **IMPORT\_PATH** specifică directorul în care compilatorul va căuta fișierele .proto când încearcă să rezolve opțiunile *import*.
  - Dacă nu este specificat, se folosește directorul curent.
  - Se poate folosi opțiunea **-I=IMPORT\_PATH** ca și o formă prescurtată a opțiunii **--proto\_path**.

# Clase generate

- Pentru fiecare tip de **mesaj M** definit într-un fișier .proto, compilatorul va genera **clasa M**.
- Fiecare clasa are un Builder asociat prin intermediu căruia se vor crea instanțe ale clasei respective.
- Clasa M cât și Builder-ul asociat au metode de tip *get* generate automat de compilatorul *protoc*. Clasa Builder are și metode de tip *set* care permit setarea valorilor.
- Clasele corespunzătoare mesajelor generate de compilator sunt **immutable**.
- După ce un obiect a fost construit, el nu mai poate fi modificat.
- Pentru a construi un mesaj, întâi se construiește builder-ul, se setează valorile câmpurilor folosind builder-ul, iar apoi se apelează metoda build() pentru a obține mesajul.
- Fiecare metodă din builder returnează un builder (obiectul *this*) care permite legarea mai multor apeluri de metode set într-o singură linie de cod.

# Clase generate

- Fiecare mesaj și builder conțin metode care permit verificarea și transformarea mesajului:
  - `toByteArray()` : `byte[]` serializează mesajul și returnează un sir de octeți conținând mesajul serializat.
  - `parseFrom(byte[] data)` : `Person` parsează un mesaj dintr-un sir de octeți.
  - `writeTo(OutputStream output)` serializează mesajul și îl scrie într-un OutputStream.
  - `writeDelimitedTo(OutputStream output)` serializează mesajul și scrie *dimensiunea* lui și *mesajul* serializat într-un OutputStream.
  - `parseFrom(InputStream input)` : `Person` citește și parsează un mesaj dintr-un InputStream.
  - `parseDelimitedFrom(InputStream input)` : `Person` citește și parsează un mesaj dintr-un InputStream (mesajul a fost *scris folosind writeDelimitedTo*).

# Exemplu mini-chat (proto2)

```
syntax="proto2";
package chat.protocol;
option java_package = "chat.protocol.protobuf";
option java_outer_classname = "ChatProtobufs";
message User{
    required string id=1;
    optional string passwd=2;
}
message Message{
    required string receiverId=1;
    required string senderId=2;
    required string text=3;
}
message ChatRequest {
    enum Type { Login = 1; Logout = 2; SendMessage = 3; GetLoggedFriends=4 ; }
    // Identifies which request is filled in.
    required Type type = 1;
    // One of the following will be filled in, depending on the type.
    optional User user = 2;
    optional Message message = 3;
}
message ChatResponse{
    enum Type { Ok = 1; Error = 2; GetLoggedFriends=3; FriendLoggedIn = 4; FriendLoggedOut=5; NewMessage=6; }
    // Identifies which request is filled in.
    required Type type = 1;
    // One of the following will be filled in, depending on the type.
    optional string error = 2;
    repeated User friends=3;
    optional User user=4 ;
    optional Message message = 5;
}
```

# Exemplu mini-chat (proto3)

```
syntax="proto3";
package chat.protocol;
option java_package = "chat.network.protobufprotocol";
option java_outer_classname = "ChatProtobufs";
message User{
    string id=1;
    string passwd=2;
}
message Message{
    string receiverId=1;
    string senderId=2;
    string text=3;
}
message ChatRequest {
    enum Type {Unkown=0; Login = 1; Logout = 2; SendMessage = 3; GetLoggedFriends=4 ;}
    // Identifies which request is filled in.
    Type type = 1;
    // One of the following will be filled in, depending on the type.
    oneof payload{
        User user = 2;
        Message message = 3;
    }
}
message ChatResponse{
    enum Type { Unknown=0; Ok = 1; Error = 2; GetLoggedFriends=3; FriendLoggedIn = 4; FriendLoggedOut=5; NewMessage=6; }
    // Identifies which request is filled in.
    Type type = 1;
    // One of the following will be filled in, depending on the type.
    string error = 2;
    repeated User friends=3;
    User user=4 ;
    Message message = 5;
}
```

# Bibliografie

- Protocol Buffers

<https://developers.google.com/protocol-buffers/>

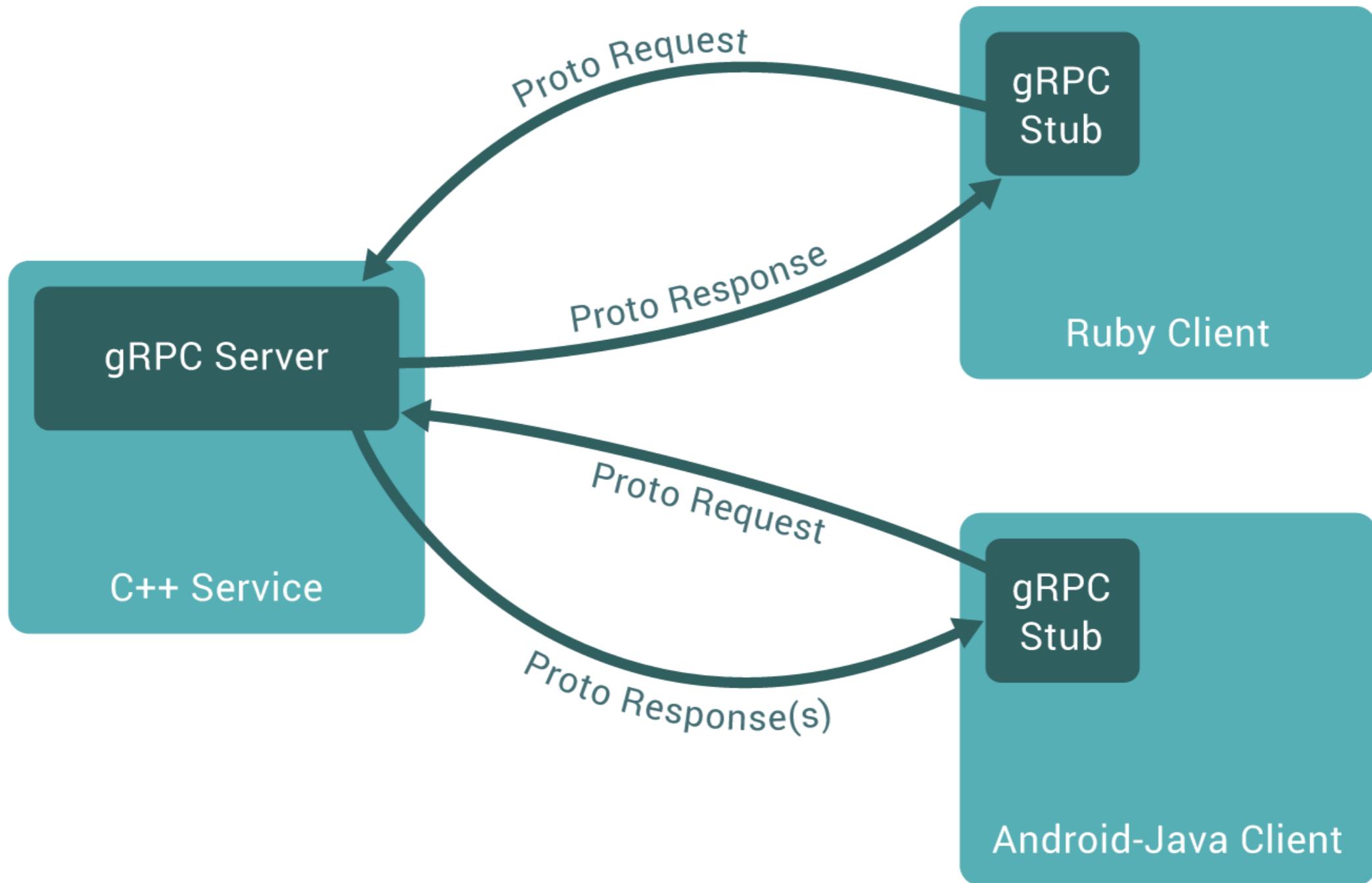
<https://github.com/google/protobuf>

- Exemplu MiniChat

# gRPC

- Cu **gRPC** o aplicație client poate apela metode la distanță ca și cum ar fi metode ale unui obiect local.
- Facilitează crearea aplicațiilor distribuite și a serviciilor.
- gRPC folosește idea definirii unui serviciu (interfețe) care specifică metodele ce pot fi apelate la distanță, împreună cu parametrii și tipul returnat al acestora.
- În aplicația **server** există un obiect remote care implementează interfața, iar serverul gRPC gestionează cererile clienților.
- Clientul (aplicația client) are un stub (proxy) care oferă aceleași metode ca și obiectul remote.
- Clienții și serverele gRPC pot fi scrise și pot rula în diferite limbaje de programare: Java, C#, C++, Python, Go, etc.

# gRPC



# gRPC - Definirea unui serviciu

- Implicit, gRPC folosește Protocol Buffers ca și limbaj de definire a tipurilor de mesaje și a serviciilor (IDL).
- Se pot folosi alte IDL-uri.

```
service HelloService {  
    rpc SayHello (HelloRequest) returns (HelloResponse);  
}  
  
message HelloRequest {  
    string greeting = 1;  
}  
  
message HelloResponse {  
    string reply = 1;  
}
```

# gRPC - Definirea unui serviciu

- gRPC permite specificarea a 4 tipuri de apeluri de metode la distanță:
  - *unare* - clientul trimită o singura cerere la server și primește un singur răspuns de la server (apeluri de funcții normale).

```
rpc SayHello(HelloRequest) returns (HelloResponse) {}
```

- *server streaming* - clientul trimită o cerere la server și primește un *stream* cu ajutorul căruia poate citi o secvență de răspunsuri de la server. Clientul citește răspunsurile până când nu mai sunt mesaje pe stream.

```
rpc LotsOfReplies(HelloRequest) returns (stream HelloResponse) {}
```

- *client streaming* - clientul scrie o secvență de mesaje și le trimită la server folosind *streamul furnizat*. După ce clientul a terminat scrierea mesajelor, așteaptă ca serverul să le citească și să trimită un singur răspuns.

```
rpc LotsOfGreetings(stream HelloRequest) returns (HelloResponse) {}
```

# gRPC - Definirea unui serviciu

- gRPC permite specificarea a 4 tipuri de apeluri de metode la distanță:
  - *Bidirectional streaming* - atât serverul cât și clientul trimit o secvență de mesaje folosind un stream bidirectional (read-write). Streamurile (read-write) funcționează independent; clienții și serverul pot citi și scrie în ordinea dorită de ei.

Exemple:

- serverul poate aștepta până primește toate mesajele de la client înainte de a trimite răspunsurile;
- serverul poate citi un mesaj, trimite răspunsul, sau altă combinație de read-write. Ordinea mesajelor din fiecare stream se păstrează.

```
rpc BidiHello(stream HelloRequest) returns (stream HelloResponse) {}
```

# gRPC - Definirea unui serviciu

- Pornind de la definiția unui serviciu dintr-un fișier .proto, gRPC furnizează **plugin-uri** pentru compilatorul protoc care permit generarea codului corespunzător clientilor și serverului.
- Clientii și serverul gRPC folosesc acest API pentru implementarea funcționalității dorite:
  - În aplicația server, **serverul implementează** metodele declarate de serviciu, și le face disponibile clientilor folosind un server gRPC. Frameworkul gRPC decodifică cererile, execută metodele remote și codifică răspunsurile.
  - În aplicația client, **clientul are un obiect local (stub, proxy)** care implementează aceleași metode ca și serviciul. Client poate apela aceste metode ca și cum ar fi apeluri locale, folosind parametrii corespunzători.

# gRPC - Timeout

- gRPC permite clienților să specifică cât timp sunt dispuși să aștepte ca un apel la distanță să își încheie execuția. Dacă durata specificată a fost depășită, apelul la distanță se va încheia cu eroarea DEADLINE\_EXCEEDED.
- Pe server se poate verifica dacă un anumit apel la distanță a depășit durata sau cât timp mai are la dispoziție.
- Atât clientul cât și serverul pot anula execuția unui apel la distanță în orice moment. Anularea duce la oprirea imediată a execuției apelului la distanță.
- Anularea nu este o operație de tip “undo”: modificările făcute înainte de anulare nu vor fi anulate și ele.

# gRPC - Terminare, canale de comunicare

- Terminarea unui apel - clientul și serverul determină independent succesul execuției unui apel la distanță, iar concluziile lor pot să difere.
  - Exemplu 1: un apel RPC se poate termina cu succes pe server ("I have sent all my responses!"), dar poate eșua pe client ("The responses arrived after my deadline!").
  - Exemplu 2: serverul poate decide terminarea înainte ca clientul să fi terminat de trimis toate cererile.
- Canal de comunicare gRPC - furnizează o conexiune la un server gRPC specificând adresa și portul și se folosește la instanțierea unui proxy (client stub). Clienții pot specifica parametrii pentru a modifica comportamentul implicit (setarea opțiunii de compresie a mesajelor, etc ).
- Un canal are asociată o stare (dacă este conectat, dacă este idle, etc.)

# gRPC - Exemplu

- O aplicație simplă client/server:
  - Obiectul remote are o metodă care transformă un text în text cu litere mari și adaugă data și ora la care a fost primit textul.
  - Clientul obține o referință la obiectul remote, apelează metoda și tipărește rezultatul primit.

# Bibliografie

- Protocol Buffers

<https://developers.google.com/protocol-buffers/>

<https://github.com/google/protobuf>

- gRPC

<http://www.grpc.io/>

# Medii de proiectare și programare

2022-2023

Curs 8

# Continut

- Remote Procedure Call

- Thrift

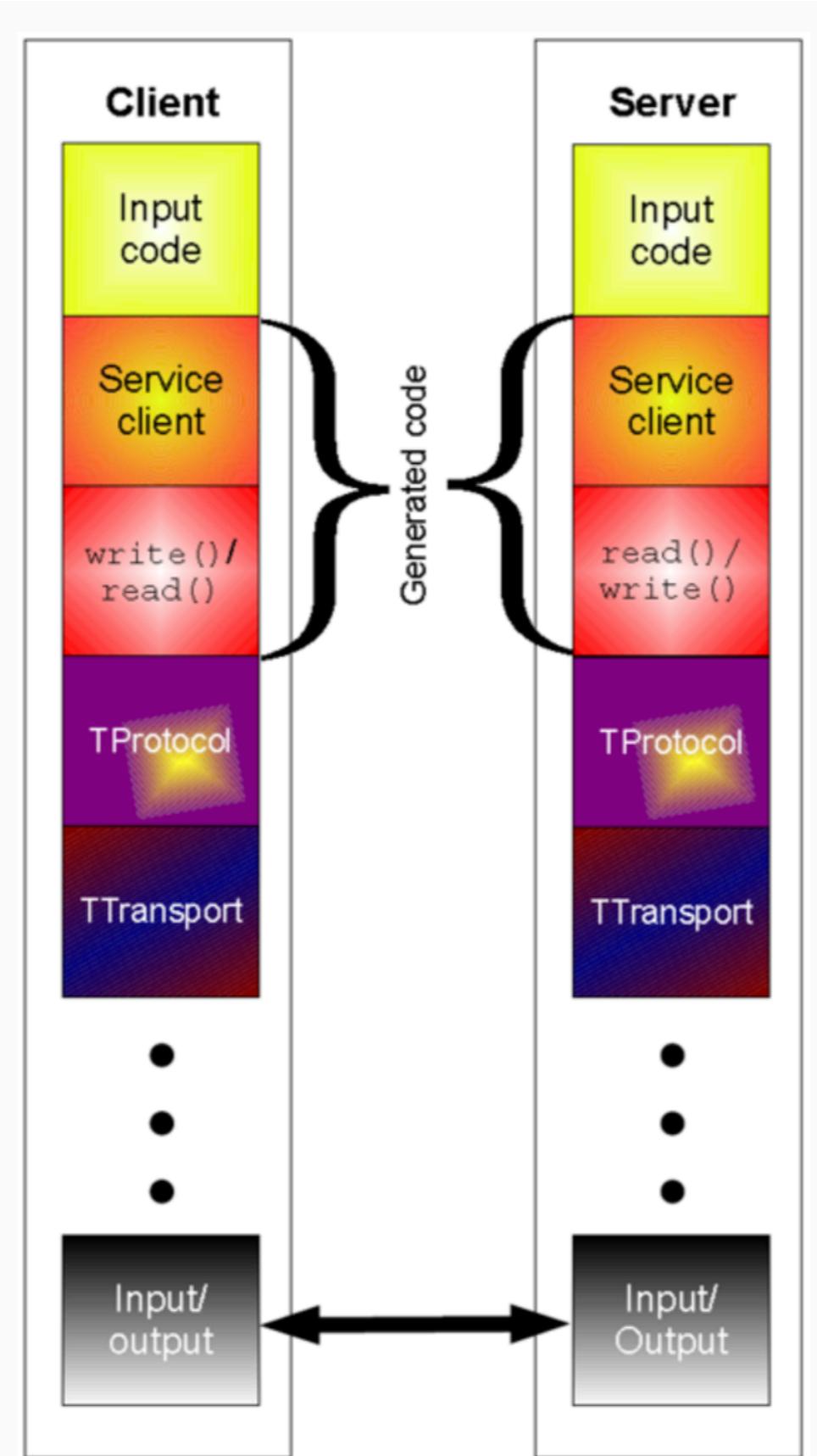
- ORM

- Hibernate

- .NET ORM

# Apache Thrift

- **Thrift** este o tehnologie cross-platform pentru RPC.
- Thrift oferă separare clară între nivelul transport, serializarea datelor și logica aplicației.
- Thrift a fost dezvoltat inițial de Facebook, acum este un proiect open-source găzduit de Apache.
- Apache Thrift conține un set de unelte de generare de cod care permit utilizatorilor să dezvolte clienți și servere RPC doar prin definirea tipurilor de date necesare și a interfețelor serviciilor într-un fișier IDL **.thrift**.
- Folosind acest fișier ca și date de intrare, se generează automat cod sursă pentru clienți și servere RPC în diferite limbaje de programare care permit aplicațiilor să comunice indiferent de limbajul folosit pentru scrierea clientului/serverului.
- Thrift suportă diferite limbaje de programare: C++, Java, C#, Python, PHP, Ruby.



*Thrift Architecture.*

# Apache Thrift

- Componente cheie:

- tipuri de date
- transport
- protocol
- versioning
- processors

- Tipuri de date Thrift

- Sistemul de tipuri de date Thrift nu introduce tipuri speciale dinamice sau obiecte wrapper.
- Programatorul nu trebuie să scrie cod pentru serializarea tipurilor de date sau pentru transportul datelor.
- Conține tipuri de bază, structuri, containere.
- Tipuri de bază:
  - *bool* - valoarea booleană: true sau false
  - *byte* - a signed byte
  - *i16* - întreg cu semn pe 16-biți
  - *i32* - întreg cu semn pe 32-biți
  - *i64* - întreg cu semn pe 64-biți
  - *double* - real pe 64-biți
  - *string*
  - *binary* - sir de octeți folosit pentru reprezentarea blob-urilor.

# Tipuri Thrift -Structuri

- O structură Thrift definește un obiect comun tuturor limbajelor ce poate fi transmis prin rețea.
- O structură este echivalentul unei clase dintr-un limbaj orientat obiect.
- Structura conține o mulțime de câmpuri care au un tip definit și un identificator unic asociat.
- Sintaxa asemănătoare structurii C.
- Câmpurile pot fi adnotate cu un identificator numeric unic (valoare întreagă) și o valoarea implicită (optional). Identificatorii trebuie să fie unici în cadrul structurii.
- Dacă identificatorii câmpurilor sunt omisi, li se vor atribui automat valori.

```
struct Example {  
    1:i32 number=10,  
    2:i64 bigNumber,  
    3:double decimals,  
    4:string name="thrifty"  
}
```

# Tipuri Thrift -Containere

- Containerele Thrift sunt containere cu tip care se vor măpa la cele mai folosite containere din limbajele de programare corespunzătoare.
- Sunt parametrizate folosind stilul Java Generics/C++ template.
- Trei tipuri de containere disponibile:
  - `list<type>` : o listă de elemente.
    - Se mapează la STL `vector`, Java `ArrayList`, sau orice alt tablou din limbajele scripting. Poate conține duplicate.
  - `set<type>` : O mulțime neordonată de elemente.
    - Se mapează la STL `set`, Java `HashSet`, `set` în Python, sau dicționare native în PHP/Ruby.
  - `map<type1, type2>` : Un dicționar cu chei unice
    - Se mapează la STL `map`, Java `HashMap`, PHP associative array, sau Python/Ruby dictionary.
- Elementele din container pot fi de orice tip valid Thrift, inclusiv alte containere sau structuri.
- În codul generat corespunzător limbajului de programare dorit, fiecare definiție va conține două metode, `read` și `write`, folosite pentru serializarea și transportul obiectelor folosind Thrift TProtocol.

# Thrift - Excepții

- Excepțiile Thrift sunt **sintactic și funcțional echivalente cu structurile Thrift** doar că sunt declarate folosind **exception** în loc de **struct**.
- Clasele generate vor moșteni din clasele de bază corespunzătoare excepțiilor în limbajul de programare folosit, pentru a se putea folosi în mod transparent cu mecanismul de tratare a excepțiilor din limbajul respectiv.

# Servicii Thrift

- Serviciile Thrift se definesc folosind tipurile Thrift.
- Definirea unui serviciu este echivalentă semantic cu definirea unei interfețe într-un limbaj de programare orientat pe obiecte.
- Compilatorul Thrift va genera stub-uri client și server complet funcționale care implementează interfața.
- Lista parametrilor și lista excepțiilor sunt implementate ca și structuri Thrift.

```
service <name> {
    <returntype> <name>(<arguments>) [throws (<exceptions>)]
    ...
}
```

```
service StringCache {
    void set(1:i32 key, 2:string value),
    string get(1:i32 key) throws (1:KeyNotFound knf),
    void delete(1:i32 key)
}
```

# Nivelul transport Thrift

- Nivelul transport este folosit de codul generat automat pentru a ușura transmiterea datelor între clienți și server.
- Thrift se folosește de obicei peste TCP/IP ca și nivel de bază pentru comunicare.
- Codul generat Thrift trebuie să știe doar cum să scrie și cum să citească datele.
- Originea sau destinația datelor sunt irelevante: poate fi socket, memorie partajată sau un fișier pe discul local.
- Interfața Thrift **TTransport** conține metodele:
  - **open** deschiderea transportului
  - **close** închiderea transportului
  - **isOpen** verifică dacă transportul este deschis
  - **read** citește date
  - **write** scrie date
  - **flush** forțează scrierea datelor păstrate în zona tampon.

# Nivelul transport Thrift

- Interfață **TServerTransport** folosită pentru crearea și acceptarea obiectelor transport:
  - **open** deschidere
  - **listen** așteaptă conexiuni de la clienți
  - **accept** returnează un nou obiect transport (când s-a conectat un client nou)
  - **close** închidere
- Interfața transport este proiectată pentru implementare ușoară în orice limbaj de programare.
- Se pot defini noi mecanisme de transport:
  - Clasa **TSocket** este implementată în toate limbajele de programare suportate. Oferă o modalitate simplă de comunicare cu un socket TCP/IP.
  - Clasa **TFileTransport** este o abstractizare a unui stream ce reprezintă un fișier de pe discul local. Poate fi folosită pentru a salva cererile clientilor într-un fișier de pe disc.

# Thrift - Protocol

- Thrift cere respectarea unei anumite structurii a mesajelor când sunt transmise prin rețea, dar **nu știe de protocolul efectiv folosit pentru serializarea/codificarea acestora.**
- Nu contează dacă datele sunt serialized folosind XML, human-readable ASCII (stringuri) sau octeți, dacă mecanismul suportă o mulțime de operații prestabilite care permit citirea și scrierea datelor.
- Interfața Thrift Protocol suportă:
  - *trimiterea mesajelor bidirectional,*
  - *codificarea tipurilor de bază, a structurilor și a containerelor.*
- Are o implementare care folosește un protocol binar.
- Scrie **toate datele** într-un format binar:
  - Tipurile întregi sunt convertite într-un format rețea independent de limbaj.
  - Stringurile au adăugate la început lungimea lor în număr de octeți.
  - Mesajele și antetul câmpurilor sunt scrise folosind construcții de serializare a datelor întregi.
  - Numele câmpurilor nu sunt serialized, identificatorii asociați sunt suficienți pentru reconstruirea datelor.

# Thrift - Protocol

```
writeMessageBegin(name, type, seq)
writeMessageEnd()
writeStructBegin(name)
writeStructEnd()
writeFieldBegin(name, type, id)
writeFieldEnd()
writeMapBegin(ktype, vtype, size)
writeMapEnd()

...
name, type, seq = readMessageBegin();
readMessageEnd()
name = readStructBegin()
readStructEnd()
name, type, id = readFieldBegin()
    readFieldEnd()
k, v, size = readMapBegin()
readMapEnd()

...
```

# Thrift - Versioning

- Thrift este **robust** la schimbări de versiuni și de definiție a datelor.
- **Versioning** este implementat folosind identificatorii asociați câmpurilor:
  - Antetul unui câmp dintr-o structură Thrift este codificat folosind identificatorul unic.
  - Combinarea (identificator câmp, tip câmp) este folosită pentru a identifica unic un câmp din structură.
- Limbajul IDL Thrift permite atribuirea automată de identificatori pentru câmpuri, dar se recomandă definirea explicită a acestora.
- Dacă la parsare/decodificare/deserializare se întâlnește un câmp necunoscut acesta este **ignorat** și **distrus**.
- Dacă un câmp care ar trebui să existe nu apare, programatorul poate fi notificat de lipsa acestuia (folosind structura **isset** definită în interiorul fiecărei obiect).
- Obiectul **isset** din interiorul unei structuri Thrift poate fi interogat pentru a determina existența unui anumit câmp. De fiecare dată când se primește o instanță a unei structuri, programatorul ar trebui să verifice existența unui câmp înainte de folosirea lui.

# Thrift - Implementare RPC

- Instanțe a clasei **TProcessor** sunt folosite pentru a trata cererile RPC:

```
interface TProcessor {  
    bool process(TProtocol in, TProtocol out) throws TException  
}
```

- Pentru fiecare serviciu dintr-un fișier .thrift se generează următoarele:

- o interfață Servicelf corespunzătoare serviciului
- clasa ServiceClient implementează Servicelf

    TProtocol in

    TProtocol out

- clasa ServiceProcessor : TProcessor

    Servicelf handler

- clasa ServiceHandler implementează Servicelf

- TServer(TProcessor processor,  
          TServerTransport transport,  
          TTransportFactory tfactory,  
          TProtocolFactory pfactory)

serve()

# Thrift - Implementare RPC

- Serverul încapsulează logica corespunzătoare conexiunii, threadurilor, etc, iar obiectul de tip TProcessor se ocupă de apelul metodelor la distanță.
- Programatorul trebuie să scrie doar codul din fișierul .thrift și implementarea corespunzătoare serviciilor (ServiceHandler)
- Există mai multe implementări posibile pentru TServer:
  - TSimpleServer: server secvențial
  - TThreadedServer: server concurrent (se creează câte un thread pentru fiecare conexiune)
  - TThreadPoolServer: server concurrent care folosește un container de threaduri.
- Exemplu: Text transformer

# Bibliografie

- Apache Thrift

<http://thrift-tutorial.readthedocs.io/en/latest/index.html>

<https://thrift.apache.org/>

# Object/Relational Mapping (ORM)

- *Object-relational mapping* ( ORM, O/RM sau O/R mapping) este o tehnică de programare pentru convertirea informațiilor/tipurilor dintr-un sistem orientat-obiect într-o bază de date relațională.
- Principiul mapării obiect-relație/înregistrare este de a delega altor instrumente managementul persistenței și de a lucra doar cu entitățile din domeniu, nu cu structurile dintr-o bază de date relațională.
- Instrumentele de mapare obiect-relație stabilesc o legătură bidirectională între o bază de date relațională și obiectele din sistem, pe baza unei configurații și execută interogări SQL la baza de date (interogări construite dinamic).

# Terminologie

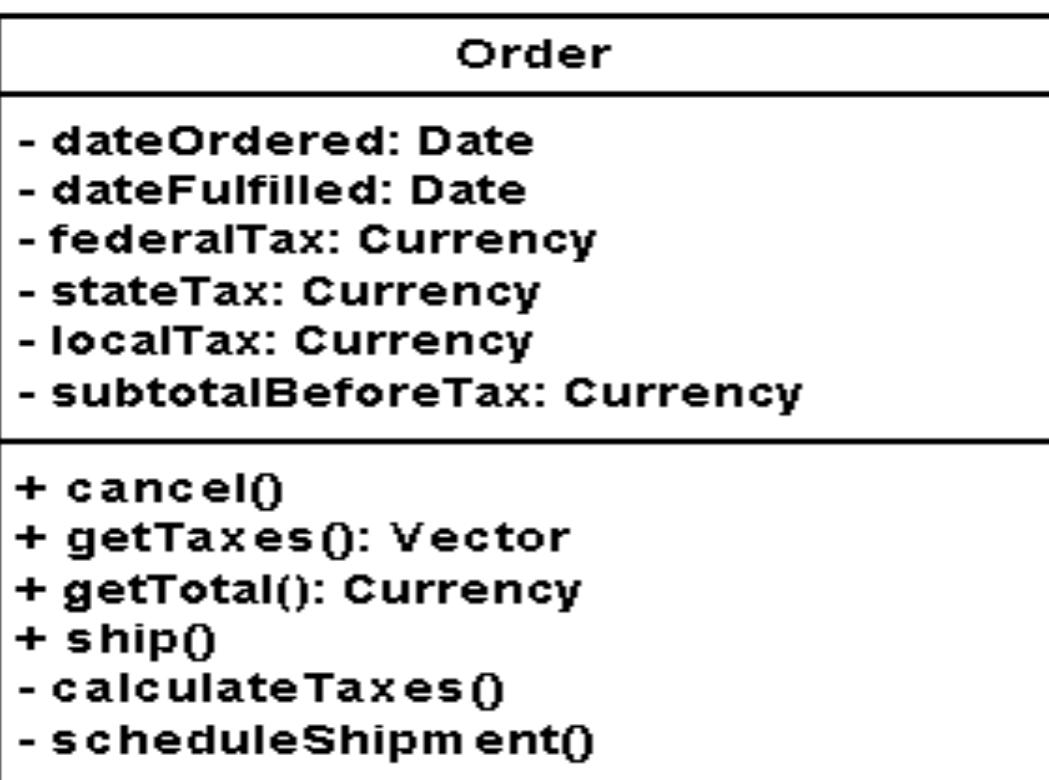
- *Mapare*. Determinarea modului în care obiectele și relațiile dintre ele vor fi păstrate într-un mediu de stocare permanent (ex. bază de date relatională, fișiere XML).
- *Proprietate*. O proprietate care poate avea asociat un atribut `firstName:string` sau o metodă prin care se determină valoarea `getTotal()`.
- *Maparea proprietății*. O mapare care descrie cum va fi stocată valoarea proprietății.
- *Maparea relațiilor*. O mapare care descrie cum vor fi persistate relațiile dintre unul sau mai multe obiecte (asociere, agregare, moștenire).

# Mapări simple

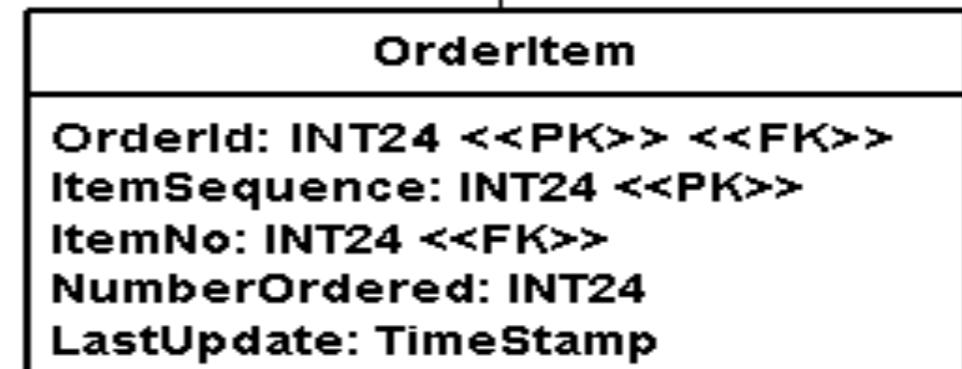
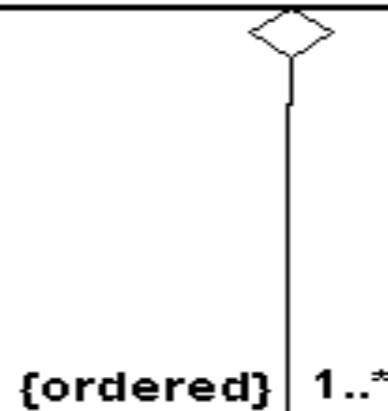
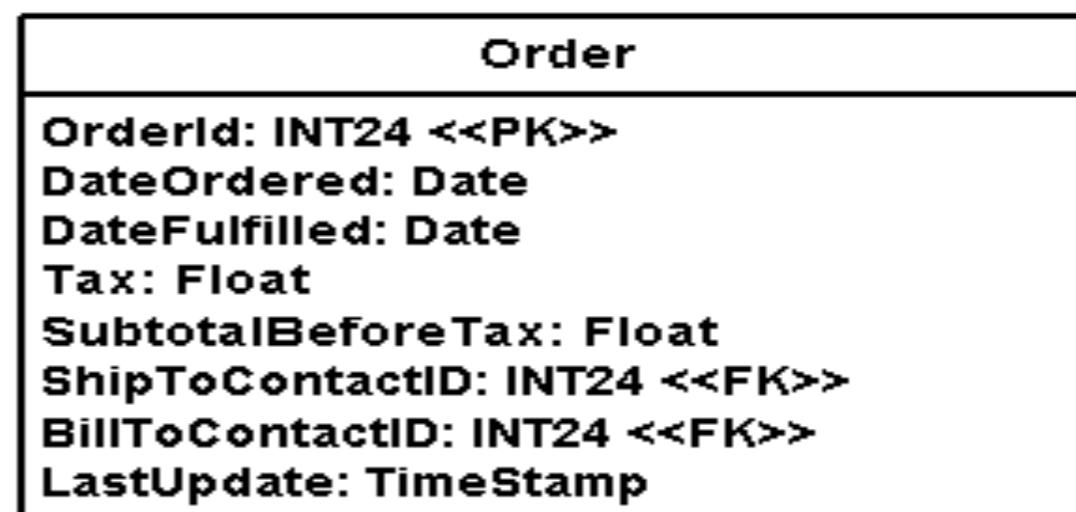
- O clasă este mapată într-o tabelă.
- Exceptând modele foarte simple, maparea unu-la-unu este rară.
- Cea mai simplă mapare este maparea unei proprietăți asociate unui singur atribut la o singură coloană din tabela corespunzătoare.
- Este și mai simplu dacă ambele (atributul și coloana) au aceleași tipuri de bază.
  - ambele sunt stringuri/date,
  - atributul este un string, coloana e de tip char(varchar),
  - atributul este un număr, iar coloana este float.

# Mapări simple

<<Class Model>>



<<Physical Data Model>>



# Diferențe

- Sunt mai multe atribute pentru **tax** în modelul orientat obiect - doar o singură coloană în schema relațională. Cele trei atribute din clasa **Order** ar trebui însumate și rezultatul păstrat în tabelă.
- În schema relațională apar chei, în modelul orientat obiect nu există chei. Înregistrările din tabela relațională sunt identificate în mod unic de cheia primară, iar relațiile dintre tabele sunt păstrate folosind chei străine.
- **Relațiile dintre obiecte sunt păstrate prin referințe, nu prin chei străine.** Pentru a putea persista obiectele și relațiilor dintre ele, obiectele trebuie să știe de valoarea cheilor păstrate în baza de date pentru a le putea identifica. Informația adițională este numită “*shadow information*”.
- Sunt folosite diferite tipuri în modelul orientat obiect și în schema relațională:
  - atributul **subTotalBeforeTax** din clasa **Order** este de tip **Currency**
  - coloana **SubTotalBeforeTax** din tabela **Order** este de tip float.
  - Pentru a implementa maparea trebuie să putem converti între cele două reprezentări fără a pierde informații.

# Shadow Information

- *Shadow information* sunt orice informații pe care obiectele trebuie să le păstreze (pe lângă informațiile normale) pentru a putea fi persistate.
- Include:
  - *Cheia primară*: în special când cheia primară este o cheie surogat care nu are altă semnificație în domeniu.
  - Informații pentru *controlul concurenței*: timestamps sau incremental counters.
  - Informații pentru păstrarea *versiunii*: *versioning numbers*.
- Exemplu: tabela **Order** are coloana **OrderID** folosită ca și cheie primară și coloana **LastUpdate** folosită pentru controlul concurenței care nu apar în clasa **Order**.

# Mapare Metadata

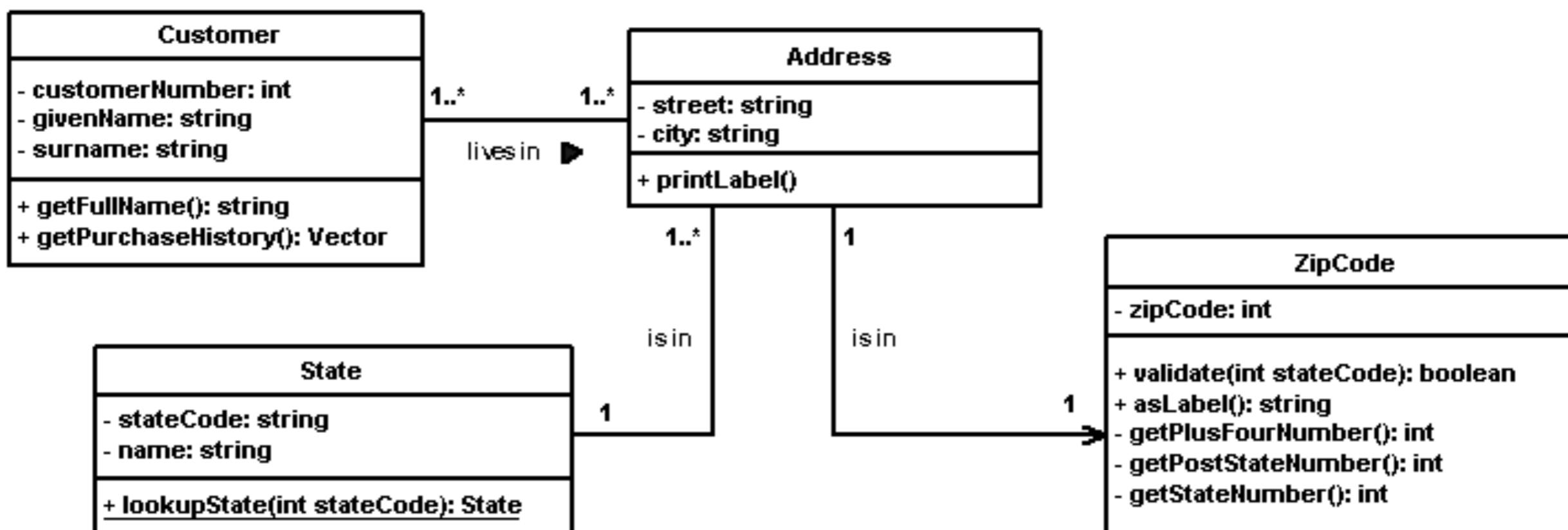
- Metadata păstrează informații despre date. Maparea metadatelor descrie modul în care metadatele reprezentând proprietățile sunt mapate la metadatele corespunzând tabelelor.

Proprietate (OO)	Coloana (BD)
Order.orderID	Order.OrderID
Order.dateOrdered	Order.DateOrdered
Order.dateFulfilled	Order.DateFulfilled
Order.getTotalTax()	Order.Tax
Order.subtotalBeforeTax	Order.SubtotalBeforeTax
Order.shipTo.personID	Order.ShipToContactID
Order.billTo.personID	Order.BillToContactID
Order.lastUpdate	Order.LastUpdate
OrderItem.ordered	OrderItem.OrderID
Order.orderItems.position(orderItem)	OrderItem.ItemSequence
OrderItem.item.number	OrderItem.ItemNo
OrderItem.numberOrdered	OrderItem.NumberOrdered

# ORM Impedance Mismatch

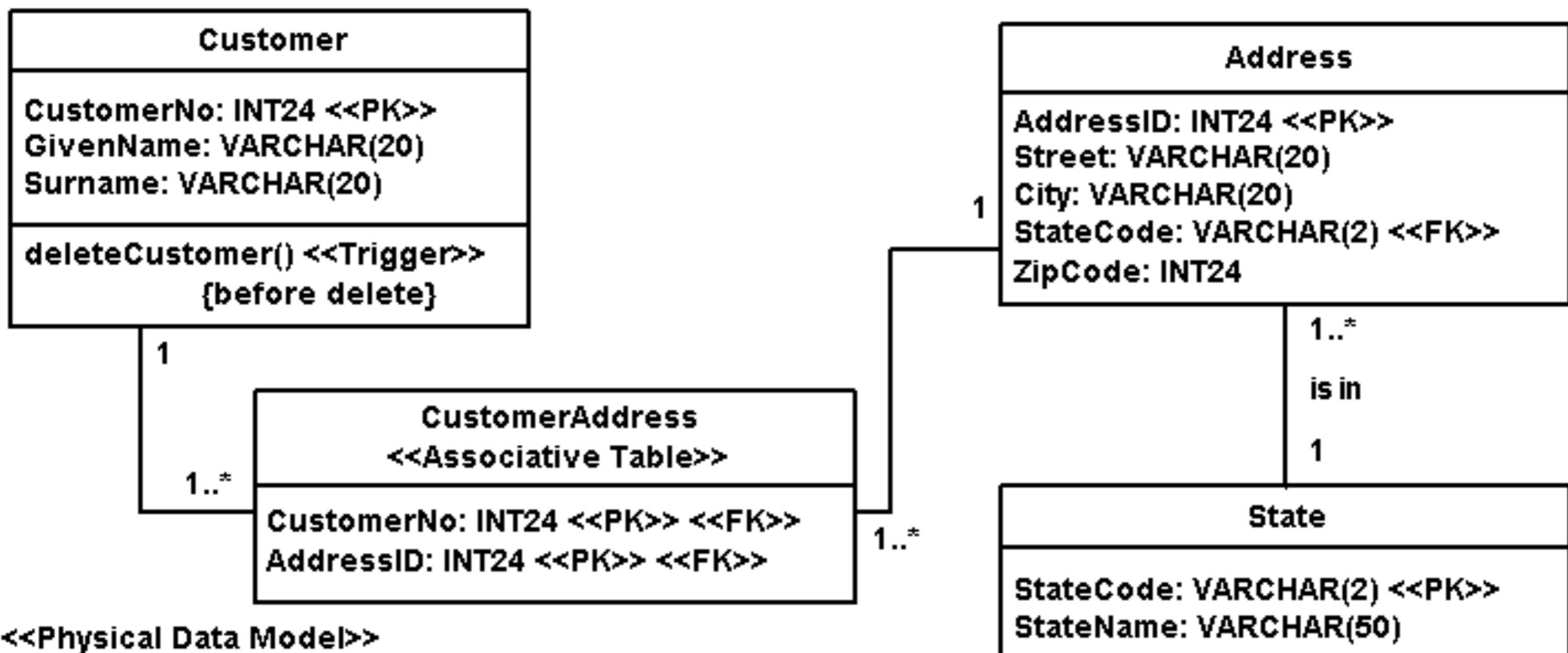
- Paradigma orientată obiect promovează dezvoltarea aplicațiilor folosind obiecte care păstrează date, dar conțin și logica aplicației.
- Bazele de date relaționale stochează datele în tabele și manipulează datele folosind proceduri stocate și interogări SQL.
- Diferențele dintre cele două abordări au fost numite: *object-relational impedance mismatch* sau doar *impedance mismatch*.
- Ex. În paradigma orientată obiect obiectele sunt traversate folosind relațiile dintre ele, în paradigma relațională se folosește operația de join.
- Tipurile de date diferite în limbajele orientate obiect și bazele de date relaționale:
  - Java: string și int - Oracle: varchar și smallint.
  - Java: colecții - Oracle: tabele
  - Java: obiecte - Oracle: blobs

# ORM Impedance Mismatch



Copyright 2002-2006 Scott W. Ambler

# ORM Impedance Mismatch



Copyright 2002-2006 Scott W. Ambler

# Strategii pentru Impedance Mismatch

- Maparea moștenirii
- Maparea relațiilor dintre obiecte
- Maparea proprietăților statice

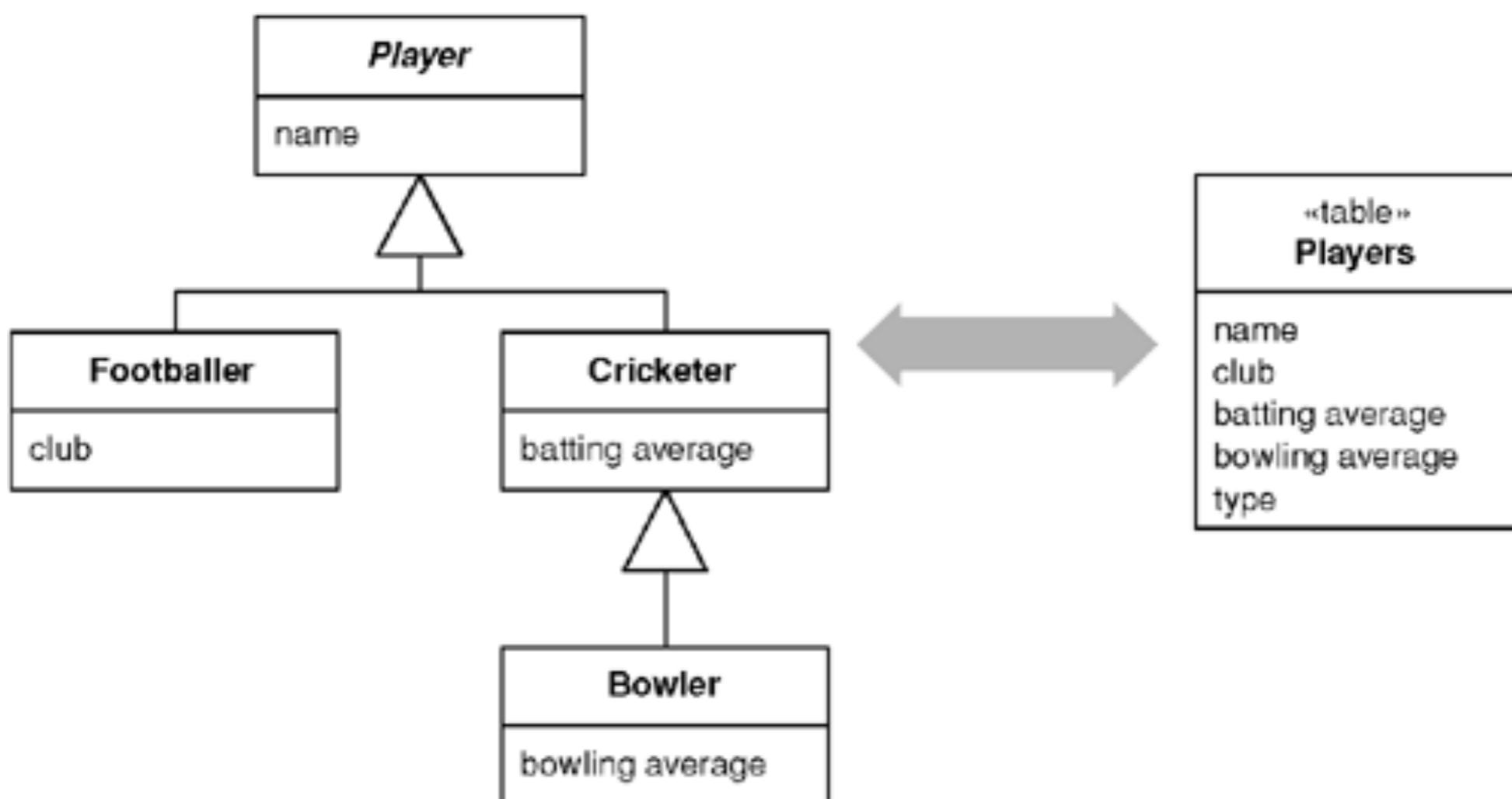
# Maparea moștenirii

- Bazele de date relaționale nu suportă moștenirea.
- Programatorul trebuie să mapeze moștenirea dintre entitățile din modelul orientat obiect într-o bază de date relatională.
- Tehnici:
  - Maparea ierarhiei de clase într-o singură tabelă.
  - Maparea fiecărei clase concrete în tabela ei.
  - Maparea fiecărei clase într-o tabelă.
  - Maparea claselor într-o structura de tabele generică.

# Moștenirea - O singură tabelă

- Reprezentarea unei ierarhii de clase (moștenire) **ca și o singură tabelă** cu coloane pentru toate atributele din toate clasele din ierarhie.
- Fiecare clasă păstrează informațiile relevante pentru ea într-o înregistrare din tabelă. Coloanele care nu sunt relevante rămân goale.
- Când se încarcă un obiect din tabelă, instrumentul ORM trebuie să știe ce clasă să instanțieze.
- În tabelă se adaugă o coloană care indică ce clasă ar trebui instanțiată (numele clasei sau un cod):
  - Codul trebuie interpretat în codul sursă pentru a putea face maparea cu clasa corespunzătoare.
  - Numele clasei poate fi folosit direct pentru instanțiere (folosind reflecție).

# Moștenirea - O singură tabelă



# Moștenirea - O singură tabelă

```
Footballer fb=new Footballer("A A", "ABC")
Cricketer cr=new Cricketer("C C", 23);
Bowler bw=new Bowler("B B", 21, 47);
Footballer fb2=new Footballer("D D", "BGD");
Bowler bw2=new Bowler("H H", 12, 23);
```

## Players

PK	Name	Club	BattlingAvg	BowlingAvg	Type
1	A A	ABC			Footballer
2	C C		23		Cricketer
3	B B		21	47	Bowler
4	D D	BGD			Footballer
5	H H		12	23	Bowler

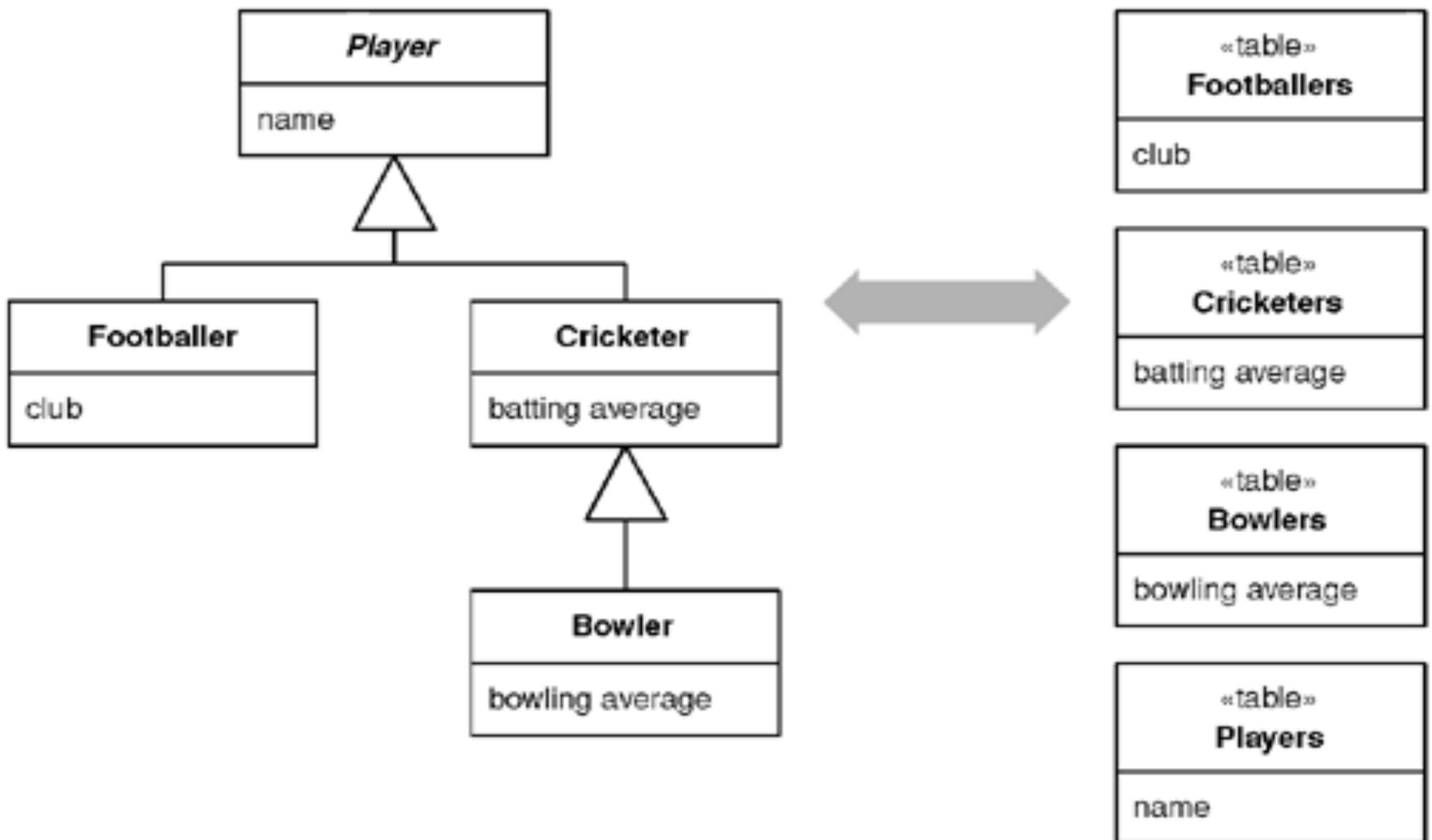
# Moștenirea - O singură tabelă

- Avantaje:
  - Există doar o singură tabelă în baza de date.
  - Nu e nevoie de operații **join** pentru regăsirea informației.
  - Orice refactorizare care mută atributele în ierarhie nu necesită modificarea bazei de date.
- Dezavantaje:
  - Nu toate câmpurile din tabelă sunt **relevante** (depinde de tipul clasei). Este confuz pentru cei care folosesc tabelele direct (fără instrumentul ORM).
  - **Coloanele folosite doar de subclase duc la spațiu nefolosit** (multe coloane goale).
  - **Tabela poate deveni prea mare**, cu mulți indecși și blocări dese ale tabelei. Poate afecta performanța.
  - **Există un singur spațiu de nume pentru câmpuri**, dezvoltatorul trebuie să se asigure că se vor folosi nume diferite în tabelă.
    - Adăugarea numelui clasei (prefix, postfix) poate ajuta. (ex. NumeClasă\_NumeProprietate)

# Tabelă pentru fiecare clasă

- Fiecare clasă din ierarhie are tabela ei.
- Atributele din clasă se mapează direct la coloanele corespunzătoare din tabelă.
- *Problemă:* Cum se leagă înregistrările din tabele?
  - *Soluția A:* folosirea cheii primare atât în tabela corespunzătoare clasei de bază cât și în clasa derivată. Deoarece clasa de bază are câte o înregistrare pentru fiecare înregistrare din clasele derivate, cheia primară va fi unică între toate tabele.
  - *Soluția B.* Fiecare tabelă să aibă cheia primară proprie, și folosirea cheii străine pentru a păstra legătura cu tabela corespunzătoare clasei de bază.
- Provocare: încărcarea/regăsirea informațiilor din mai multe tabele în mod eficient.
  - Operații de join între diferite tabele
  - Operațiile de join între mai mult de 3 sau 4 tabele sunt lente din cauza modului în care bazele de date optimizează operațiile interne.
- Interogările asupra bazei de date sunt dificile.

# Tabelă pentru fiecare clasă



# Tabelă pentru fiecare clasă

```
Footballer fb=new Footballer("A A", "ABC")
Cricketter cr=new Cricketter("C C", 23);
Bowler bw=new Bowler("B B", 21, 47);
Footballer fb2=new Footballer("D D", "BGD");
Bowler bw2=new Bowler("H H", 12, 23);
```

**Players**

PK	Name
1	A A
2	C C
3	B B
4	D D
5	H H

**Footballers**

PK	Club
1	ABC
4	BGD

**Cricketters**

PK	BattlingAvg
2	23
3	21
5	12

**Bowlers**

PK	BowlingAvg
3	47
5	23

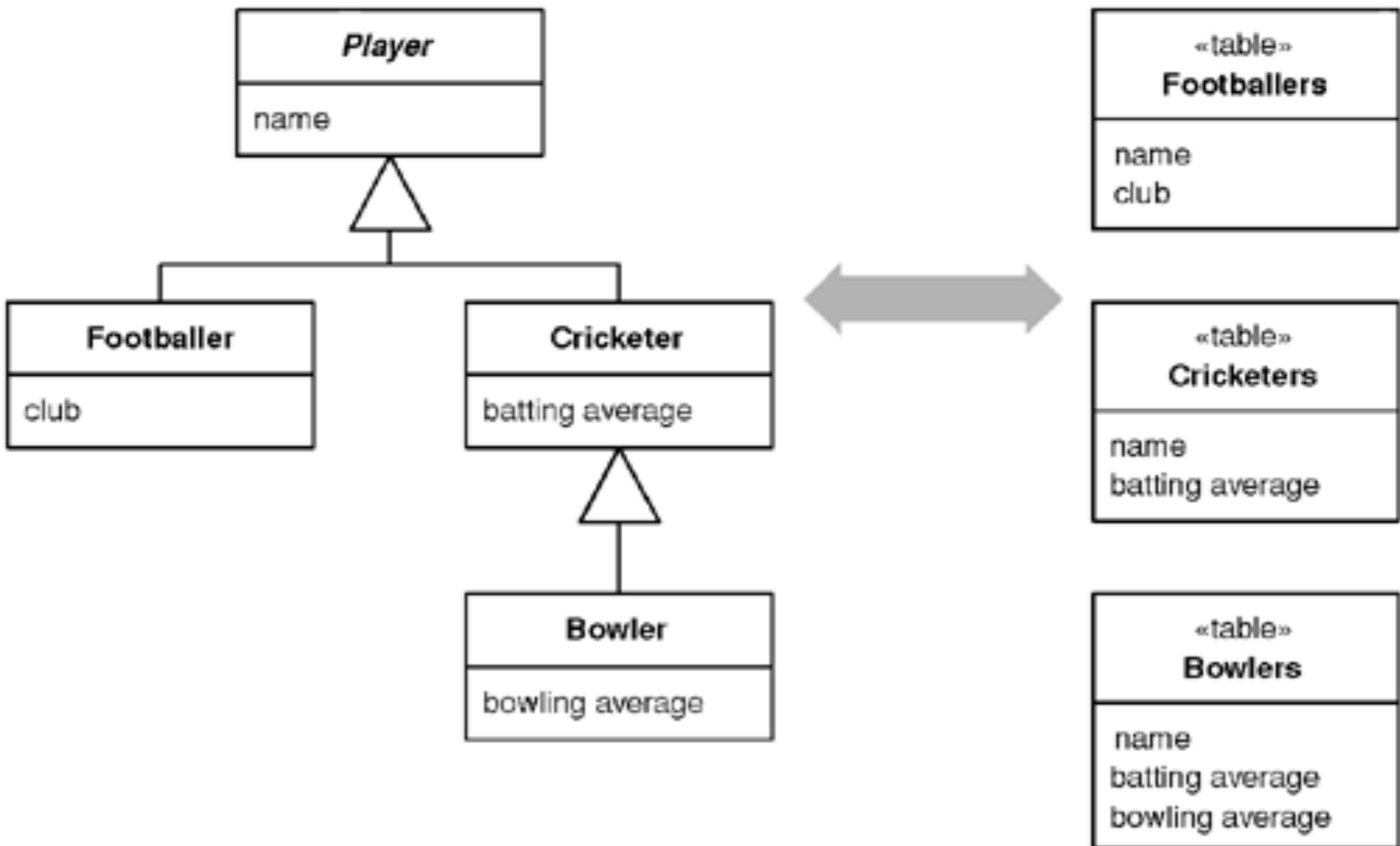
# Tabelă pentru fiecare clasă

- Avantaje:
  - Toate coloanele sunt relevante pentru fiecare înregistrare, tabelele sunt mai ușor de înțeles și nu se folosește spațiu în mod inefficient.
  - Relația dintre entitățile din modelul orientat obiect și baza de date relațională este ușor de înțeles.
- Dezavantaje:
  - Este necesară folosirea mai multor tabele pentru a încărca un obiect din mediu persistent (operație join sau mai multe interogări și folosirea memoriei).
  - Orice refactorizare (mutarea câmpurilor în ierarhia de clase) cauzează modificarea structurii bazei de date.
  - Tabelele corespunzătoare claselor de bază pot cauza probleme de performanță din cauza accesării dese.
  - Normalizarea poate duce la înțelegerea dificilă a interogărilor ad-hoc.

# Tabelă pentru fiecare clasă concretă

- Fiecare clasă concretă (non-abstract) din ierarhie are tabela ei.
- Fiecare tabelă conține coloane pentru toate proprietățile din ierarhie până la ea. **Atributele din clasa de bază sunt duplicate în tabelele corespunzătoare subclaszelor.**
- Este responsabilitatea programatorului de a se asigura că cheile sunt unice nu doar în tabela corespunzătoare clasei dar și între toate tabelele asociate ierarhiei.

# Tabelă pentru fiecare clasă concretă



# Tabelă pentru fiecare clasă concretă

```
Footballer fb=new Footballer("A A", "ABC")
Cricketter cr=new Cricketter("C C", 23);
Bowler bw=new Bowler("B B", 21, 47);
Footballer fb2=new Footballer("D D", "BGD");
Bowler bw2=new Bowler("H H", 12, 23);
```

**Footballers**

PK	Name	Club
1	A A	ABC
4	D D	BGD

**Cricketters**

PK	Name	BattlingAvg
2	C C	23

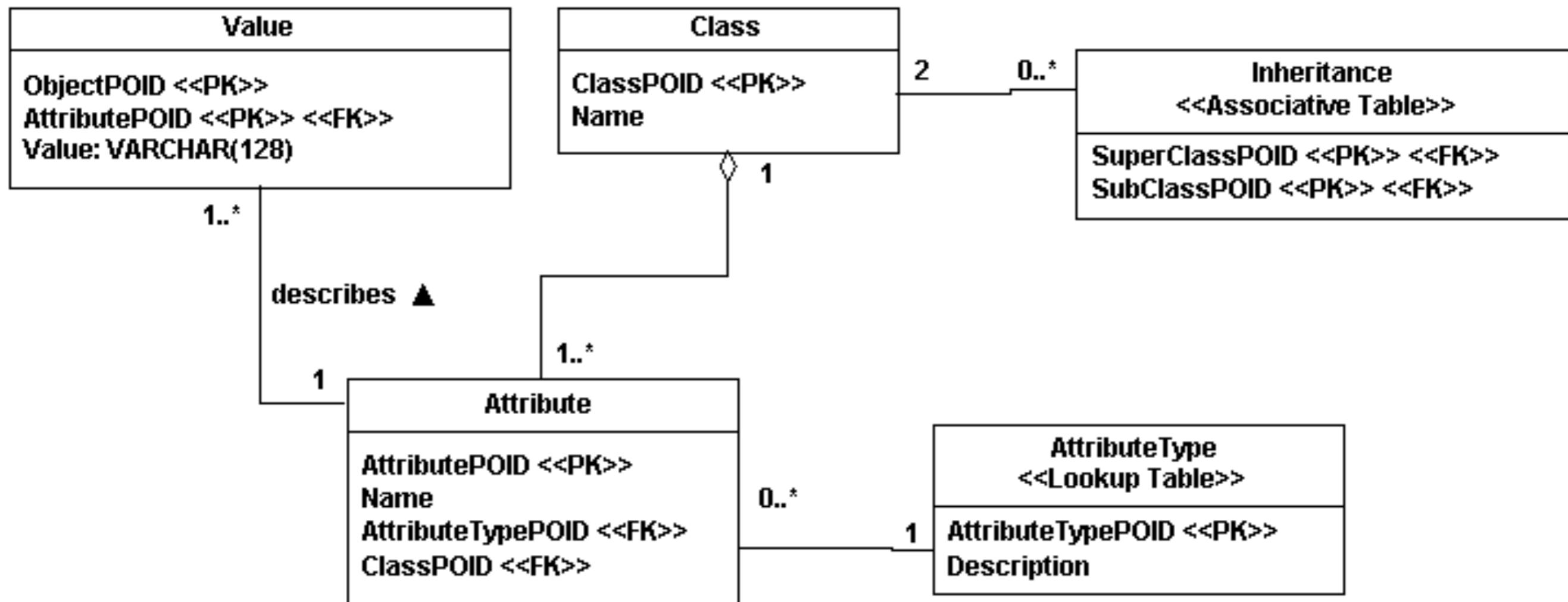
**Bowlers**

PK	Name	BattlingAvg	BowlingAvg
3	B B	21	47
5	H H	12	23

# Tabelă pentru fiecare clasă concretă

- Avantaje:
  - Fiecare tabelă păstrează toate informațiile relevante și nu are câmpuri irelevante. Este ușor de înțeles și de alte aplicații care nu folosesc obiecte.
  - Nu este nevoie de operații join pentru citirea datelor.
  - Fiecare tabelă este accesată doar când clasa respectivă este accesată. Performanța este mai bună.
- Dezavantaje:
  - Gestiunea dificilă a cheilor primare.
  - Nu pot fi constrânse relațiile către clasele abstracte.
  - Dacă câmpurile din modelul obiectual sunt mutate în ierarhie, trebuie modificate definițiile tabelelor.
  - Dacă se modifică un câmp dintr-o clasă de bază, trebuie modificate toate tabelele corespunzătoare subclaszelor, pentru că aceste câmpuri sunt duplicate.
  - O operație de căutare folosind clasa de bază, necesită căutari în toate tabelele (accesări multiple ale bazei de date sau o operație de join complicată).

# Tabele generice



Copyright 2002-2006 Scott W. Ambler

# Tabele generice

- Avantaje:
  - Poate fi extinsă pentru a oferi suport pentru o gamă largă de mapări, inclusiv maparea relațiilor.
  - Este flexibilă, permite modificarea ușoară a modului în care sunt păstrate obiectele (trebuie modificate doar metadatele din tabelele *Class*, *Inheritance*, *Attribute* și *AttributeType*).
- Dezavantaje:
  - Este fezabilă doar pentru date de dimensiuni mici, deoarece necesită accesări dese ale bazei de date doar pentru reconstruirea unui singur obiect).
  - Interogările pot fi dificile deoarece necesită accesarea mai multor înregistrări pentru un singur obiect.

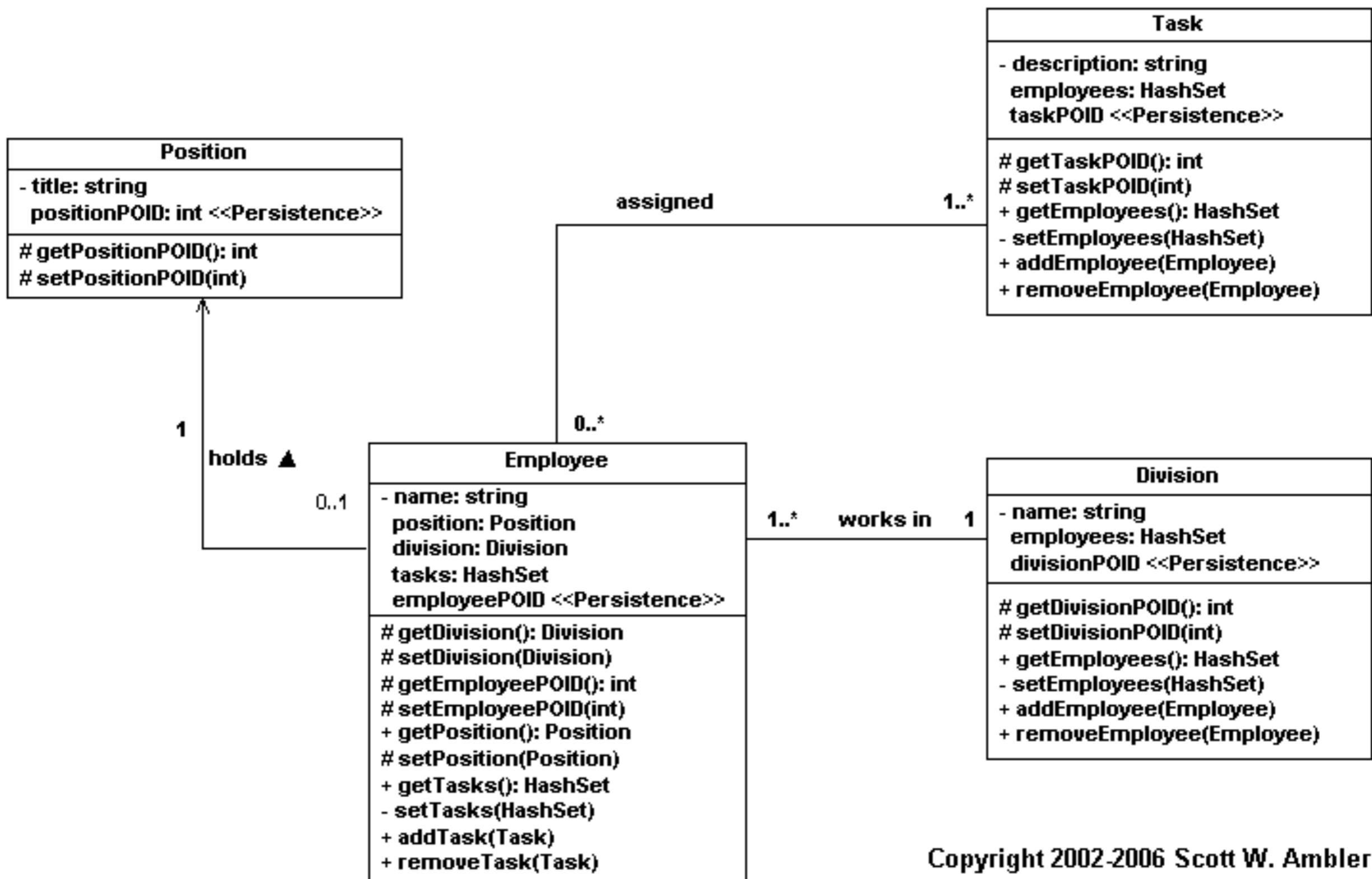
# Maparea relațiilor dintre obiecte

- *Problema:* modul în care paradigma orientată obiect și modelul relational tratează legăturile dintre obiecte/înregistrari, care cauzează două situații.
  1. *Diferența reprezentării.* Obiectele tratează legăturile prin păstrarea referințelor care există în timpul execuției (**referințele sunt temporare**). Bazele de date relationale tratează legăturile prin păstrarea cheilor în tabele (**cheile sunt permanente**).
  2. *Obiectele pot folosi colecții pentru a gestiona mai multe referințe într-un singur atribut.* Normalizarea obligă ca toate legăturile/valorile să nu fie multiple. Se inversează structura de date dintre obiecte și tabele.
- Exemplu: Un obiect *Order* are o colecție de obiecte de tip *line item* care nu păstrează o referință către obiectul de tip *order*.
  - ★ Structura tabelei este inversă, înregistrările *line item* includ o cheie străină către înregistrarea *order* corespunzătoare (câmpurile dintr-o înregistrare nu pot fi multivaloare).

# Tipuri de relații

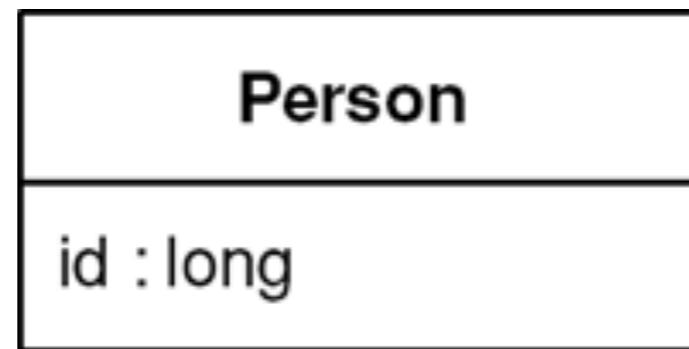
- Două categorii de relații între obiecte importante la mapare:
- **Multiplicitatea.** Include 3 tipuri:
  - Relații *unu-la-unu*. Maximul multiplicității la ambele capete este 1.
  - Relații *unu-la-n (sau n-la-unu)*. Multiplicitatea la unul dintre capete este 1, la celălalt capăt este n.
  - Relații *n-la-n*. Maximul multiplicității la ambele capete este mai mare decât 1.
- **Directia.** Include 2 tipuri:
  - Relații *unidirectionale*. Un obiect știe de obiectul (obiectele) cu care are o legătură, dar celălalt obiect (celealte obiecte) nu știe (nu știu) de el.
  - Relații *bidirectionale*. Ambele obiecte aflate într-o relație știu unul de celălalt.

# Tipuri de relații



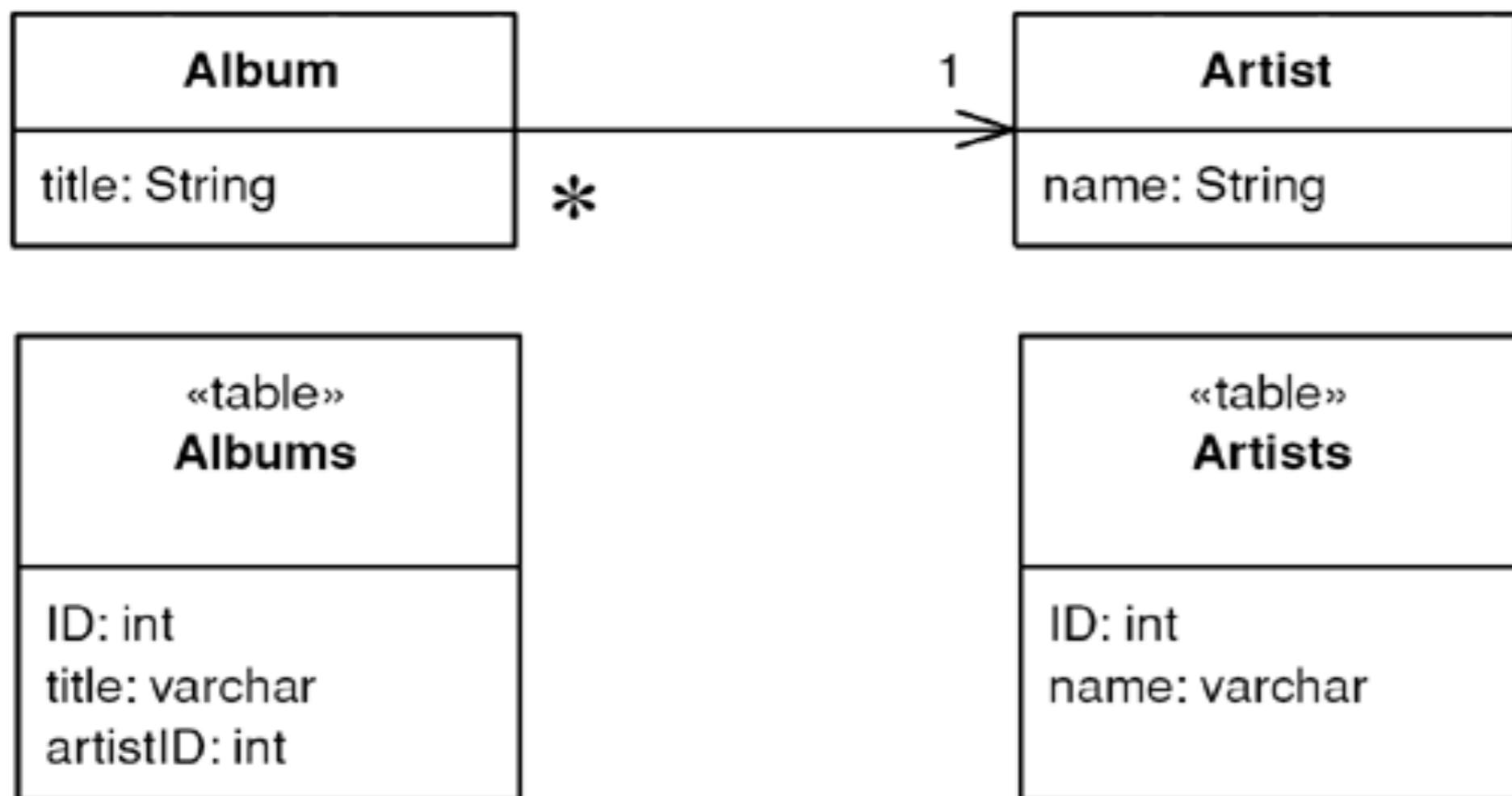
# Şablonul *Identity*

- Salvează un identificator (ID) din baza de date într-un obiect pentru a păstra legătura dintre un obiect în memorie și o înregistrare din baza de date.
- Cheia primară dintr-o bază de date relațională este păstrată printre atributele obiectului.
- Şablonul ar trebui folosit când există o mapare între obiectele din memorie și înregistrările din baza de date (cheia primară este diferită de atributele din obiect).



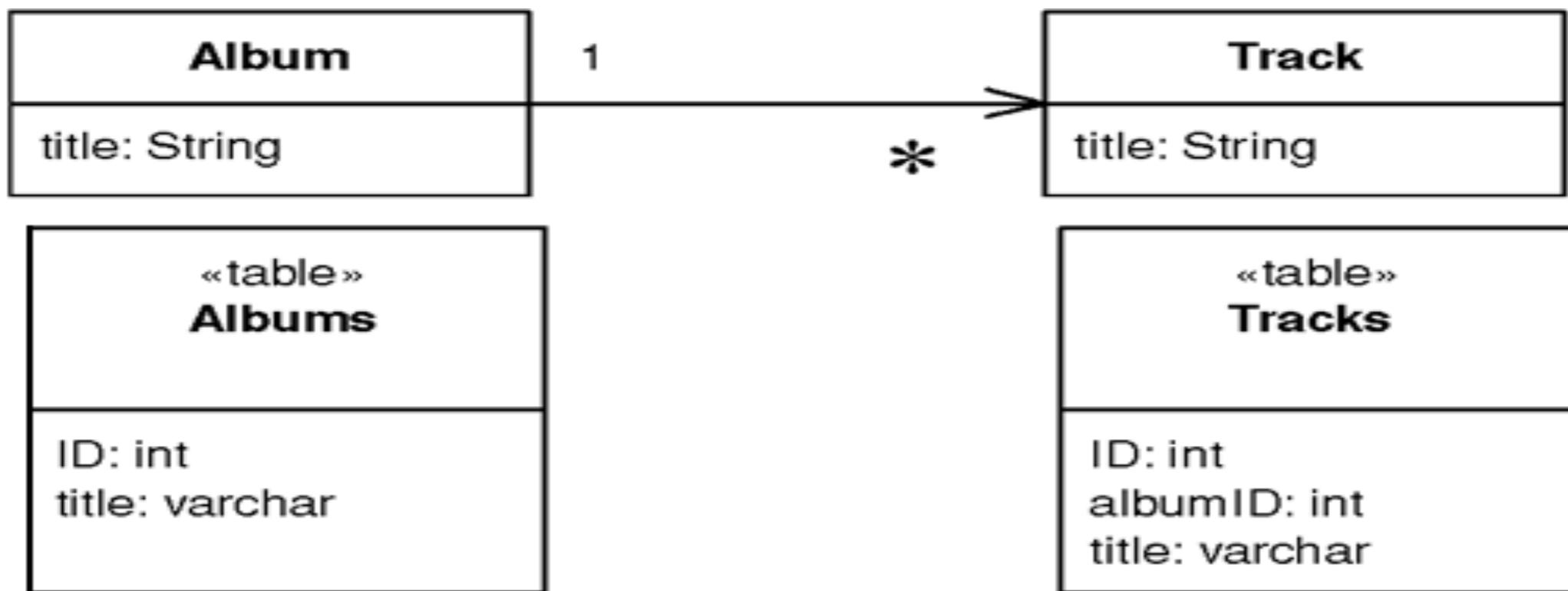
# Maparea cheii străine

- Mapează o asociere dintre obiecte ca și cheie străină între tabelele dintr-o bază de date relațională.
- Fiecare obiect conține cheia din tabela corespunzătoare.
- Dacă două obiecte sunt legate printr-o relație de asociere, relația poate fi înlocuită printr-o cheie străină în baza de date.



# Maparea cheii străine

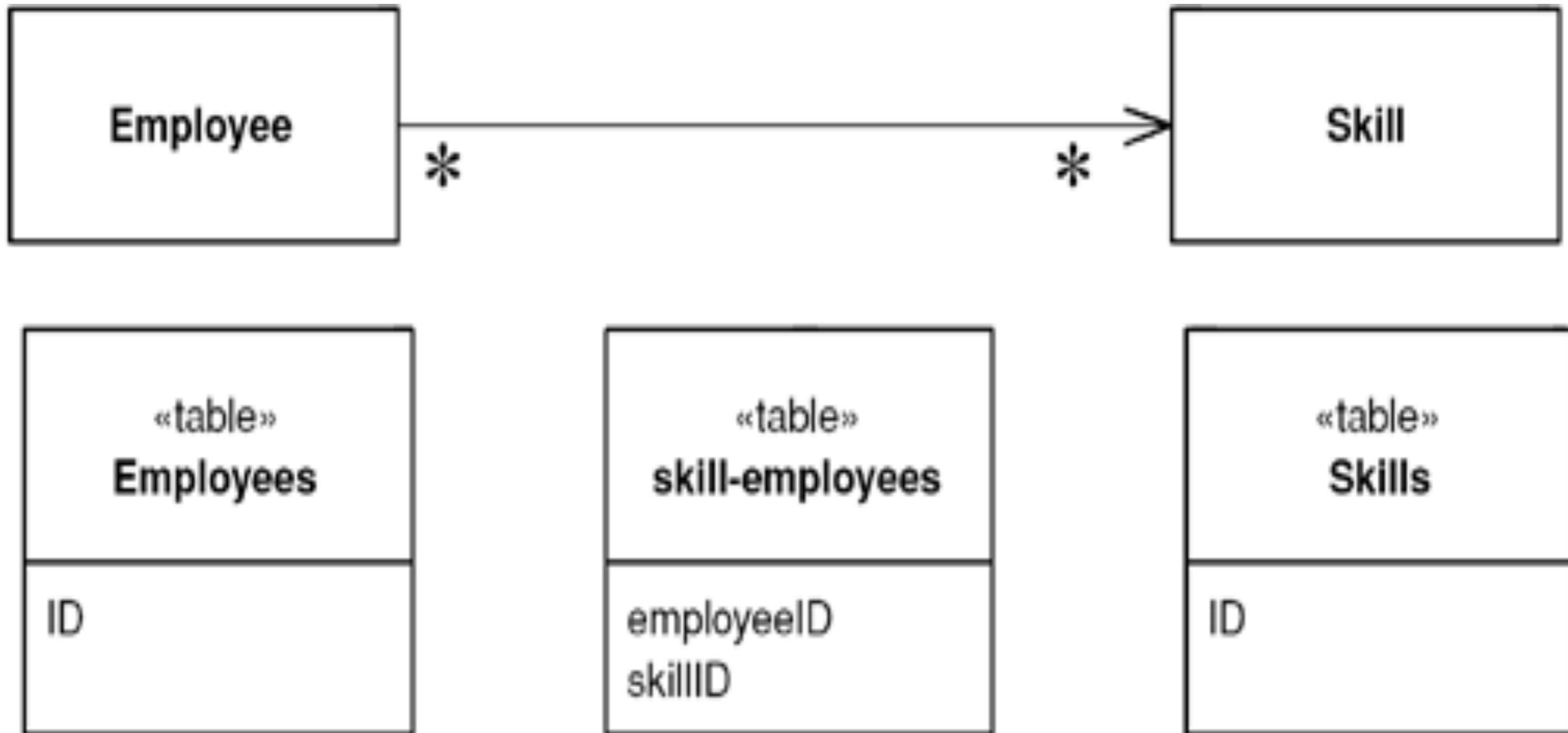
- Maparea unei colecții de obiecte.
- Într-o bază de date relațională nu se poate salva o colecție, trebuie inversată direcția referinței.
- Maparea cheii străine poate fi folosită pentru aproape toate asocierile dintre clase. Nu poate fi folosită pentru asocierea *n-la-n*.



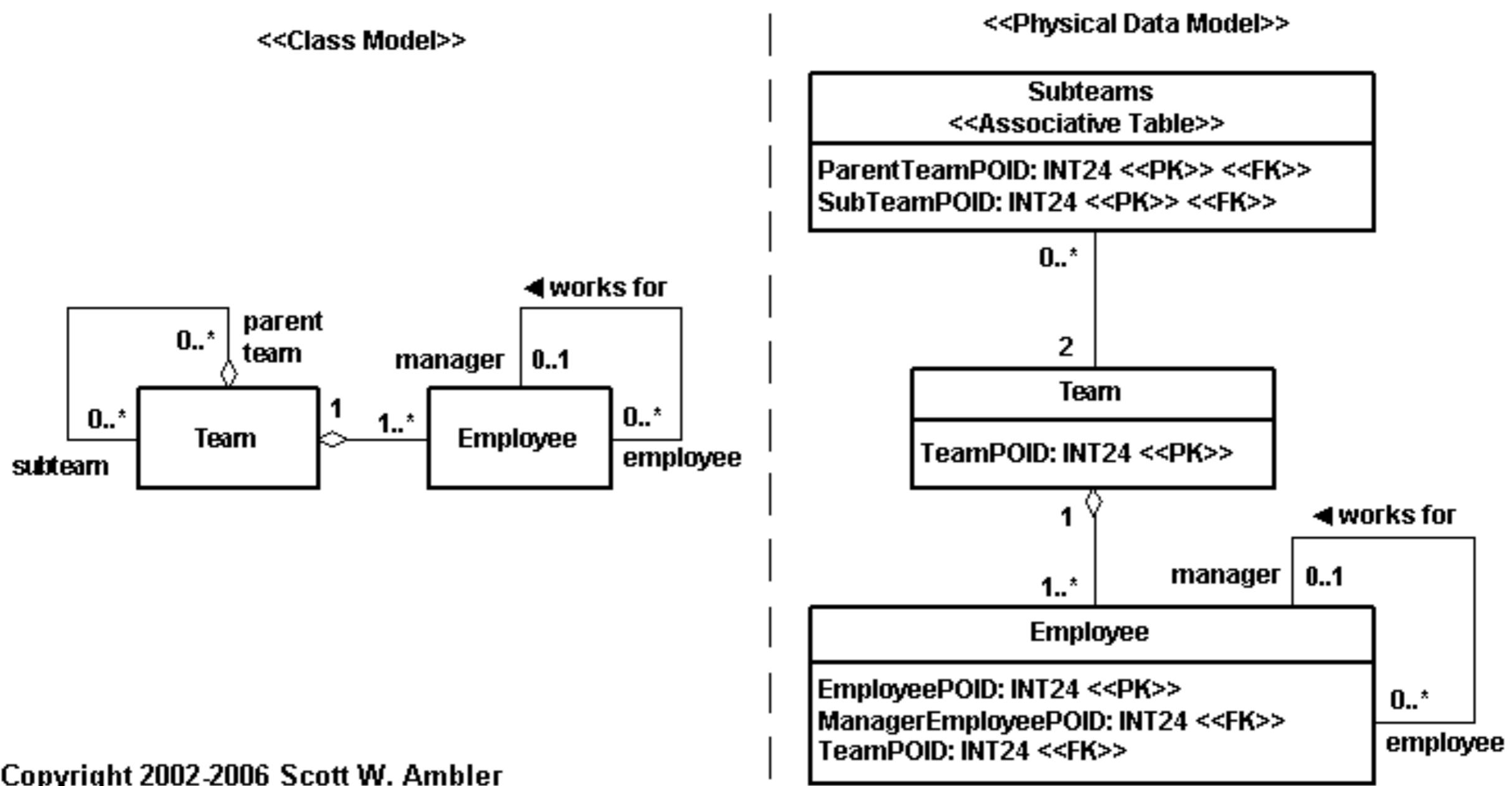
# Maparea folosind o tabelă de asociere

- Salvează o asociere ca o tabelă cu chei străine către tabelele legate prin asociere.
- Obiectele pot păstra mulțimi de valori folosind colecții. Bazele de date relationale nu au această caracteristică și sunt restrictionate la câmpuri cu o singură valoare.
- Idea este de a crea o tabelă de legătură/asociere pentru a stoca asocierea.
- Tabela are doar două coloane corespunzătoare cheilor străine, conține câte o înregistrare pentru fiecare pereche de obiecte asociate.
- Tabela de legătură nu are echivalentul unui obiect în memorie (nu are ID). Cheia primară este compusă din cheile primare ale celor două tabele asociate.
- Tabela de asociere este folosită cel mai des pentru maparea asocierii n-la-n, dar poate fi folosită și pentru alte tipuri de asocieri (mai dificil, complex).

# Tabela de asociere



# Maparea asocierilor recursive



# Maparea proprietăților statice

**TableA**

A
<u>StaticAttributeA</u>
name

PK	Name	StaticAttributeA
	AA	1
	BB	2
	CC	1

**TableA**

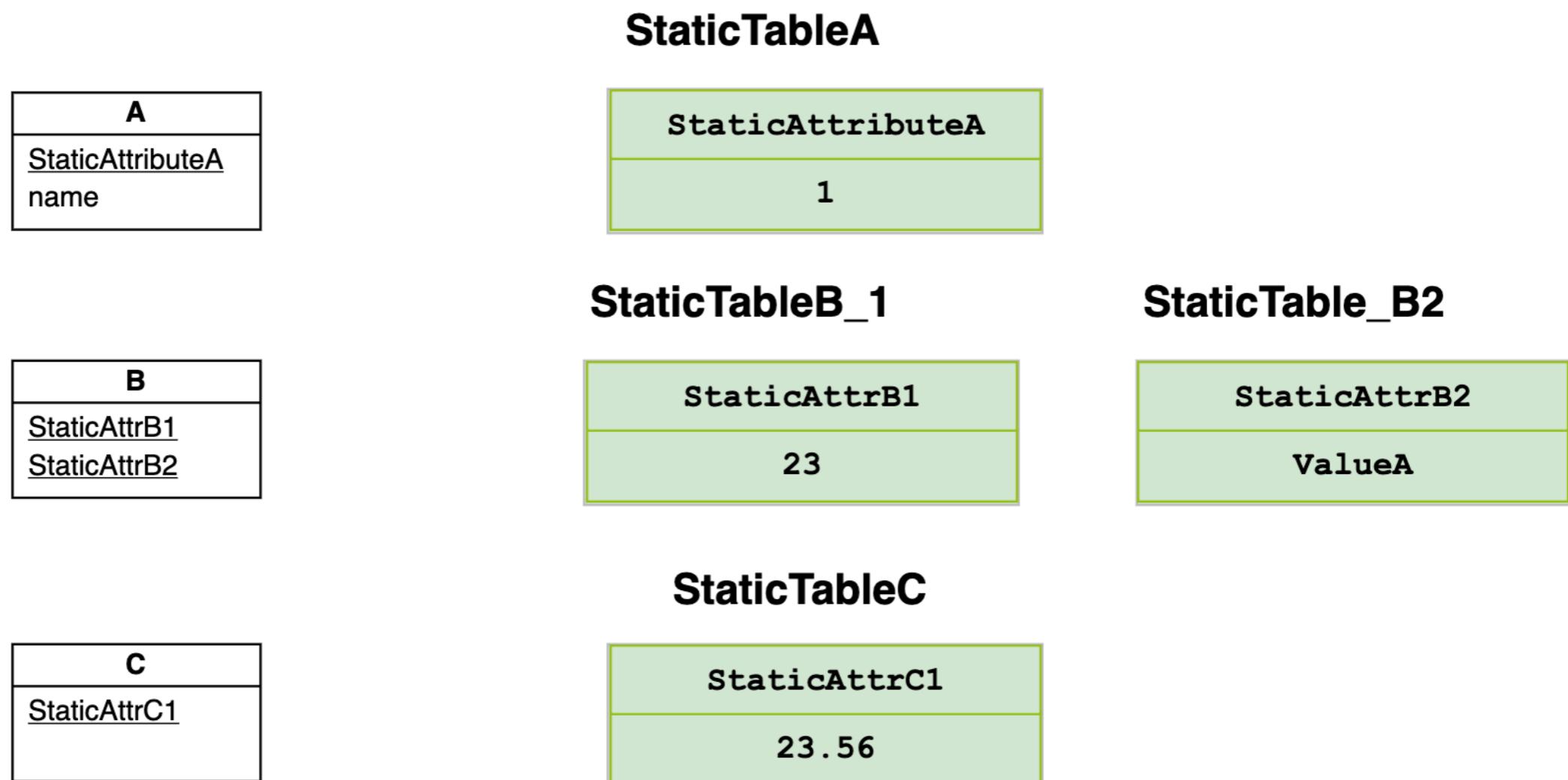
PK	Name
	AA
	BB
	CC

**StaticTableA**

StaticAttributeA
1

# Maparea proprietăților statice - Strategii (1)

- O tabelă cu o singură înregistrare, o singură coloană pentru fiecare proprietate statică:
  - Pro: Simplu, acces rapid
  - Con: multe tabele mici



# Maparea proprietăților statice - Strategii (2)

- O tabelă cu mai multe coloane, o singură înregistrare pentru fiecare clasă:
  - Pro: Simplu, acces rapid
  - Con: multe tabele mici, dar mai puține decât la strategia precedentă

**StaticTableA**

A
<u>StaticAttributeA</u>
name

StaticAttributeA
1

**StaticTableB**

B
<u>StaticAttrB1</u>
<u>StaticAttrB2</u>

StaticAttrB1	StaticAttrB2
23	ValueA

**StaticTableC**

C
<u>StaticAttrC1</u>

StaticAttrC1
23.56

# Maparea proprietăților statice - Strategii (3)

- O singură tabelă cu mai multe coloane - o singură înregistrare pentru toate clasele:
  - Pro: număr minim de tabele introdus
  - Con: potențiale probleme de concurență dacă mai multe clase trebuie să acceseze datele în același timp.

A	B	C
<u>StaticAttributeA</u> name	<u>StaticAttrB1</u> <u>StaticAttrB2</u>	<u>StaticAttrC1</u>

**StaticAttributesTable**

A.StaticAttributeA	B.StaticAttrB1	B.StaticAttrB2	C.StaticAttrC1
1	23	ValueA	23.56

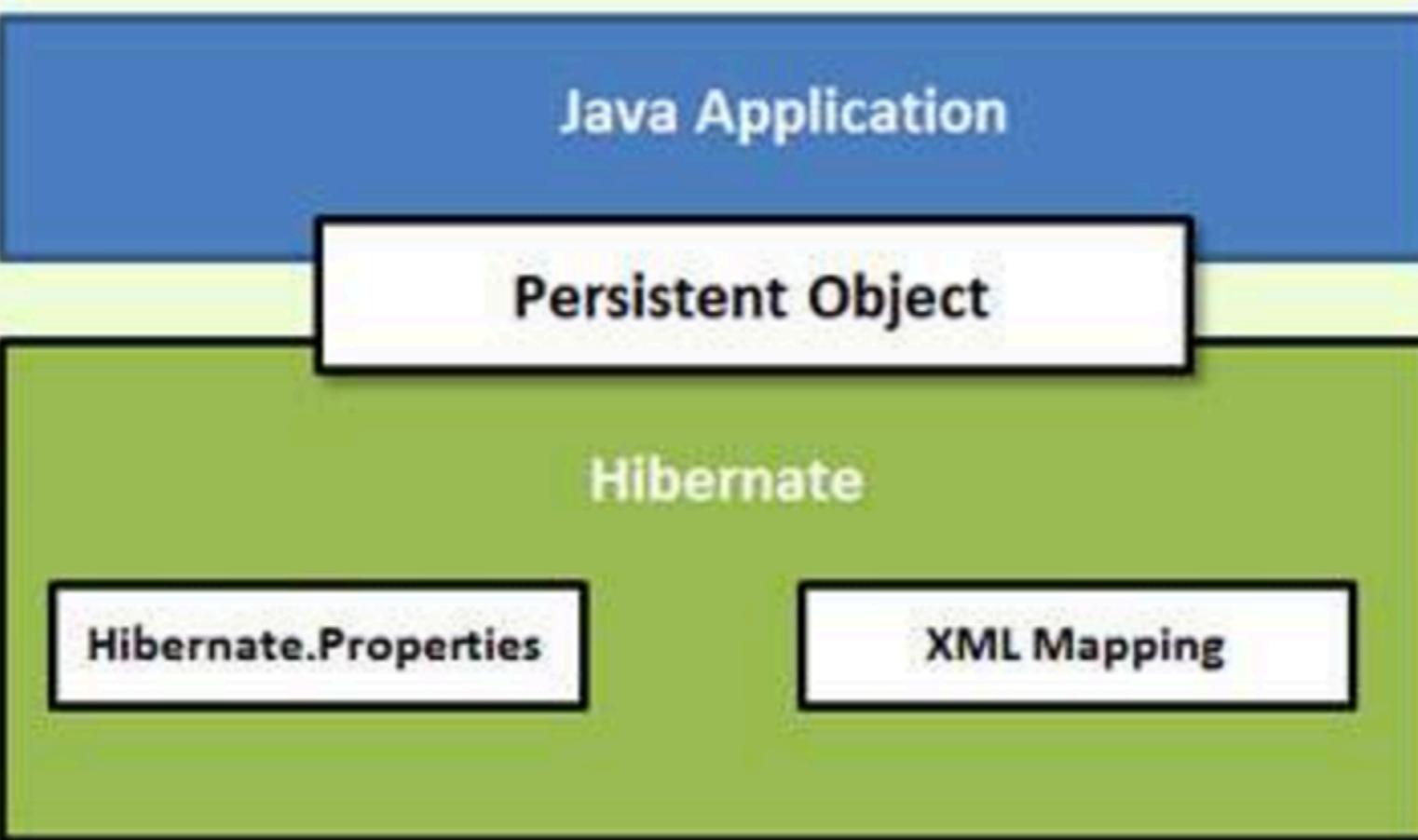
# Maparea proprietăților statice

- Strategii (cont):
  - Schemă generică cu mai multe înregistrări pentru toate clasele:
    - Pro: număr minim de tabele introdus. Reduce problemele legate de concurență.
    - Con: Necesitatea convertirii între tipuri de date. Schema este asociată cu numele claselor și a proprietăților statice.

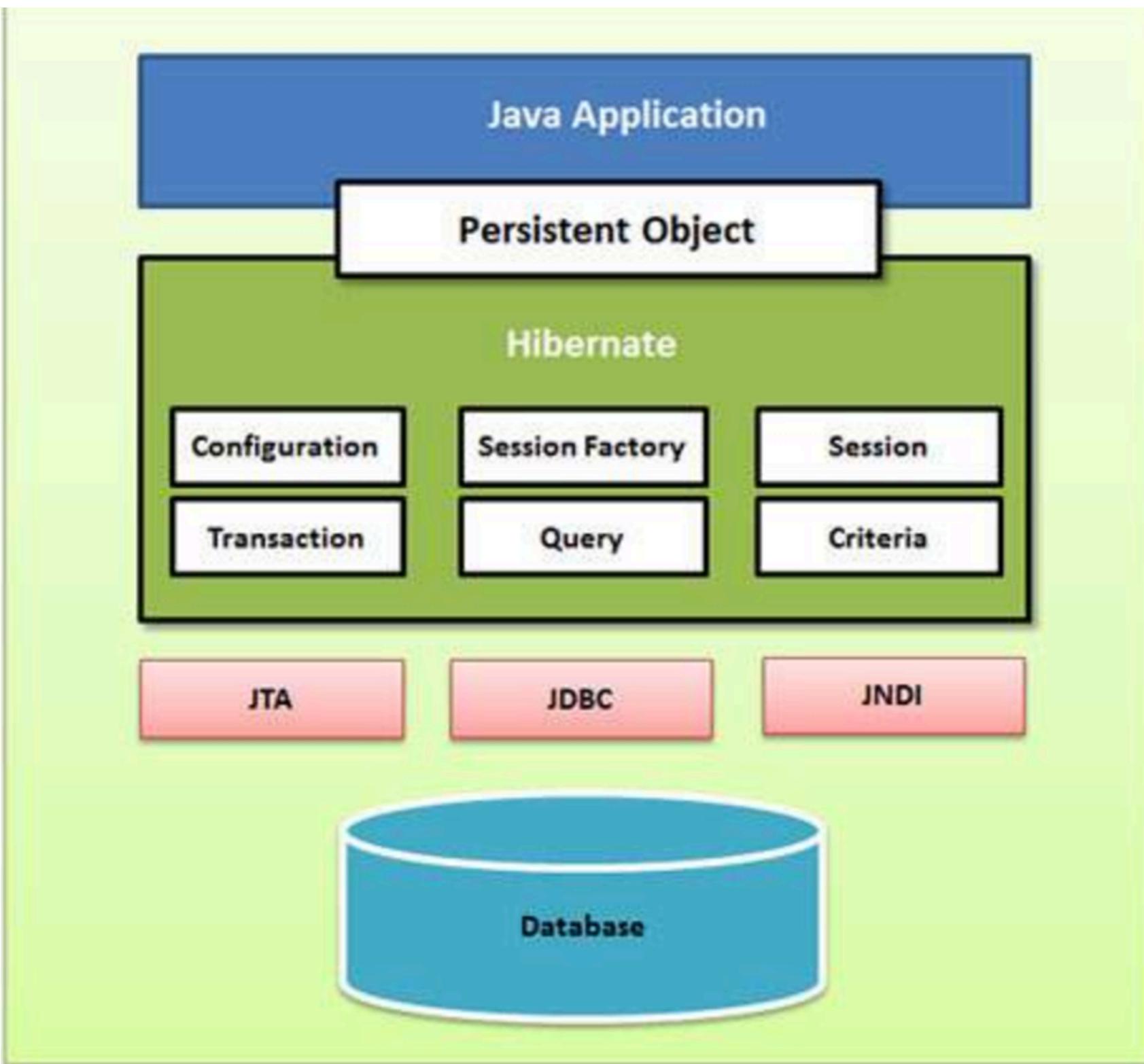
# Hibernate

- Instrument open source pentru ORM.
- Aplicațiile care folosesc Hibernate definesc/specifică clasele persistente ce vor fi mapate în tabele într-o bază de date relațională.
- Toate instrucțiunile SQL sunt generate automat în timpul execuției.
- Informațiile despre mapare sunt transmise ca și un document de mapare în format XML sau prin adnotări.
- Documentul de mapare specifică:
  - cum vor fi mapate proprietățile la coloanele din tabelă (tabele).
  - strategia aleasă de dezvoltator pentru mapare (unde este cazul).

# Arhitectura



# Arhitectura



# Arhitectura

- Interfețele/Clasele definite de Hibernate pot fi clasificate astfel:
  - Interfețe folosite de aplicații pentru a efectua operații CRUD și interogări. Logica aplicației corespunzătoare nivelului de persistență va fi strâns cuplată de acestea: *Session*, *Transaction* și *Query*.
  - Clase apelate de aplicație pentru configurarea instrumentului Hibernate: clasa *Configuration*.
  - Interfețe callback care permit aplicațiilor să reacționeze la evenimente ce apar în timpul execuției instrumentului Hibernate: *Interceptor*, *Lifecycle* și *Validatable*.
  - Interfețe care permit extinderea Hibernate: *UserType*, *CompositeUserType* și *IdentifierGenerator*.
- Hibernate folosește Java API existent: JDBC, Java Transaction API (JTA) și Java Naming and Directory Interface (JNDI).

# Interfețe de bază

- Interfața *Session*: este cea mai folosită interfață de către aplicațiile care folosesc Hibernate. O instanță de tip Session nu consumă multe resurse și poate fi ușor creată/distrusă.
- Interfața *SessionFactory*. Aplicațiile obțin instanțe de tip Session folosind un obiect de tip SessionFactory. Păstrează în cache instrucțiunile SQL generate dinamic și alte metadate legate de mapări folosite în timpul execuției.
- Clasa *Configuration*. Un obiect de tip Configuration este folosit pentru configurarea și pornirea Hibernate. Aplicațiile pot folosi acest obiect pentru a specifica locația fișierelor/ documentelor de mapare și a altor proprietăți specifice Hibernate.
- Interfața *Transaction*. Abstractizează codul corespunzător unei tranzacții de implementarea specifică folosită: o tranzacție JDBC, o tranzacție JTA UserTransaction, etc.
- Interfața *Query*. Permite efectuarea de interogări asupra bazei de date și controlează modul în care este executată interogarea. Interogările pot fi scrise în HQL sau folosind direct limbajul SQL corespunzător bazei de date.

# Exemplu

- Etape:
  - Crearea claselor Java corespunzătoare entităților
  - Crearea fișierelor de mapare
  - Crearea fișierului de configurare Hibernate
  - Implementarea nivelului de persistență folosind funcțiile corespunzătoare
  - Testarea claselor

# Exemplu - Entitatea

```
package hello;

public class Message {
    private Long id;
    private String text;
    private Message urmMessage;
    public Message() {}
    public Message(String text) { this.text = text; }
    public Long getId() { return id; }
    private void setId(Long id) { this.id = id; }
    public String getText() { return text; }
    public void setText(String text) { this.text = text; }
    public Message getNextMessage() { return urmMessage; }
    public void setNextMessage(Message nextMessage) {
        this.urmMessage = nextMessage;
    }
}
```

# Exemplu - Fișierul de mapare

- Message.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-mapping>
    <class name="hello.Message" table="MESSAGES">
        <id name="id" column="MESSAGE_ID">
            <generator class="increment"/>
        </id>
        <property name="text" column="MESSAGE_TEXT"/>
        <many-to-one name="nextMessage"
                    cascade="all"
                    column="NEXT_MESSAGE_ID"/>
    </class>
</hibernate-mapping>
```

# Exemplu - Fișierul de configurare

```
<!--hibernate.cfg.xml -->
<?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="dialect">org.hibernate.dialect.SQLiteDialect</property>
    <property name="connection.driver_class">org.sqlite.JDBC</property>
    <property name="connection.url">jdbc:sqlite:events.db</property>
    <property name="hibernate.hbm2ddl.auto">update</property>

    <!-- Echo all executed SQL to stdout -->
    <property name="show_sql">true</property>
    <property name="format_sql">true</property>

    <mapping resource="hello/Message.hbm.xml"/>

  </session-factory>
</hibernate-configuration>
```

# Medii de proiectare și programare

2022-2023

Curs 9

# Continut

- Hibernate (cont.)
- Entity Framework
- Servicii Web
- Arhitectura REST
- REST Spring

# Exemplu

- Etape:
  - Crearea claselor Java corespunzătoare entităților
  - Crearea fișierelor de mapare (XML sau adnotări)
  - Crearea fișierului de configurare Hibernate
  - Implementarea nivelului de persistență folosind funcțiile corespunzătoare
  - Testarea claselor

# Exemplu - Entitatea

```
package hello;

public class Message {
    private Long id;
    private String text;
    private Message urmMessage;
    public Message() {}
    public Message(String text) { this.text = text; }
    public Long getId() { return id; }
    private void setId(Long id) { this.id = id; }
    public String getText() { return text; }
    public void setText(String text) { this.text = text; }
    public Message getNextMessage() { return urmMessage; }
    public void setNextMessage(Message nextMessage) {
        this.urmMessage = nextMessage;
    }
}
```

# Exemplu - Fișierul de mapare

- Message.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-mapping>
    <class name="hello.Message" table="MESSAGES">
        <id name="id" column="MESSAGE_ID">
            <generator class="increment"/>
        </id>
        <property name="text" column="MESSAGE_TEXT"/>
        <many-to-one name="nextMessage"
                    cascade="all"
                    column="NEXT_MESSAGE_ID"/>
    </class>
</hibernate-mapping>
```

# Exemplu - Fisierul de configurare

```
<!--hibernate.cfg.xml -->
<?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="dialect">org.hibernate.dialect.SQLiteDialect</property>
    <property name="connection.driver_class">org.sqlite.JDBC</property>
    <property name="connection.url">jdbc:sqlite:events.db</property>
    <property name="hibernate.hbm2ddl.auto">update</property>

    <!-- Echo all executed SQL to stdout -->
    <property name="show_sql">true</property>
    <property name="format_sql">true</property>

    <mapping resource="hello/Message.hbm.xml"/>
    <!-- <mapping class="hello.Message"/> configurare cu adnotări -->

  </session-factory>
</hibernate-configuration>
```

# Exemplu – Salvarea unei entități

```
//INSERT  
void addMessage() {  
    try(Session session = sessionFactory.openSession()) {  
        Transaction tx=null;  
        try{  
            tx = session.beginTransaction();  
            Message message = new Message("New Hello World www");  
            session.save(message);  
            tx.commit();  
        }catch(RuntimeException ex){  
            if (tx!=null)  
                tx.rollback();  
        }  
    }  
}
```

# Exemplu – Modificarea unei entități

```
//UPDATE  
void updateMessage() {  
    try(Session session = sessionFactory.openSession()) {  
        Transaction tx=null;  
        try{  
            tx = session.beginTransaction();  
            Message message =  
                (Message) session.load( Message.class, new Long(3) );  
            message.setText("New Text 3");  
            Message nextMessage = new Message("Next message");  
            message.setNextMessage( nextMessage );  
            tx.commit();  
        } catch(RuntimeException ex) {  
            if (tx!=null)  
                tx.rollback();  
        }  
    }  
}
```

# Exemplu – ștergerea unei entități

```
//DELETE
void deleteMessage() {
    try(Session session = sessionFactory.openSession()) {
        Transaction tx=null;
        try{
            tx = session.beginTransaction();
            Message crit= session.createQuery("from Message where text like
'Ne%', Message.class)
                .setMaxResults(1)
                .uniqueResult();
            System.out.println("Stergem mesajul "+crit.getId());
            session.delete(crit);
            tx.commit();
        } catch(RuntimeException ex) {
            if (tx!=null)
                tx.rollback();
        }
    }
}
```

# Exemplu – selectarea unei entități

```
//SELECT  
void getMessages() {  
    try(Session session = sessionFactory.openSession()) {  
        Transaction tx=null;  
        try{  
            tx = session.beginTransaction();  
            List<Message> messages =  
                session.createQuery("from Message as m order by m.text asc",  
Message.class).setFirstResult(1).setMaxResults(5).list();  
            System.out.println( messages.size() + " message(s) found:" );  
            for (Message m:messages ) {  
                System.out.println( m.getText()+' '+m.getId() );  
            }  
            tx.commit();  
        }catch(RuntimeException ex){  
            if (tx!=null)  
                tx.rollback();  
        }  
    }  
}
```

# Exemplu – MessageMain

```
class MessageMain{
    public static void main(String[] args) {
        try {
            initialize();
            MessageMain test = new MessageMain();
            test.addMessage(); test.getMessages(); test.updateMessage();
            test.deleteMessage();
            test.getMessages();
        }catch (Exception e){System.err.println(e);}
        finally {
            close();
        }
    }

    static SessionFactory sessionFactory;
    static void initialize() {
        // A SessionFactory is set up once for an application!
        // configures settings from hibernate.cfg.xml
        final StandardServiceRegistry registry = new
                StandardServiceRegistryBuilder().configure().build();
        try {
            sessionFactory = new MetadataSources(registry).buildMetadata().buildSessionFactory();
        }
        catch (Exception e) {
            System.err.println("Eroare "+e);
            StandardServiceRegistryBuilder.destroy( registry );
        }
    }

    static void close(){
        if ( sessionFactory != null ) {
            sessionFactory.close();
        }
    }
}
```

# Interogări ale bazei de date

- Două posibilități:
  - Hibernate Query Language

```
session.createQuery("from Category c where c.name like 'Laptop%'");
```
  - SQL

```
session.createNativeQuery("select {c.*} from CATEGORY {c} where NAME like 'Laptop%'", "c", Category.class);
```

# Obținerea rezultatelor

- Metoda `list()` execută interogarea și returnează rezultatul ca și o listă:

```
List<User> result = session.createQuery("from User",
    User.class).list();
```

- Un singur obiect ca și rezultat :

```
Bid maxBid =(Bid) session.createQuery("from Bid b order by
    b.amount desc")
    .setMaxResults(1)
    .uniqueResult();
```

# Interogări cu parametri

- Parametrii cu nume

```
String queryString = "from Item item where item.description  
    like :searchString and item.date > :minDate";  
  
List result = session.createQuery(queryString)  
    .setString("searchString", searchS)  
    .setDate("minDate", minD).list();
```

- Parametrii cu poziție:

```
String queryString = "from Item item where item.description  
    like ? and item.date > ?";  
  
List result = session.createQuery(queryString)  
    .setString(0, searchString)  
    .setDate(1, minDate)  
    .list();
```

# Hibernate Query Language (HQL)

Suportă aproape toate funcțiile și operațiile SQL:

- from clause: `from Cat as cat`
- select clause: `select foo from Foo foo, Bar bar where foo.startDate = bar.date`
- where clause: `from Cat as cat where cat.name='Fritz'`
- aggregate functions:

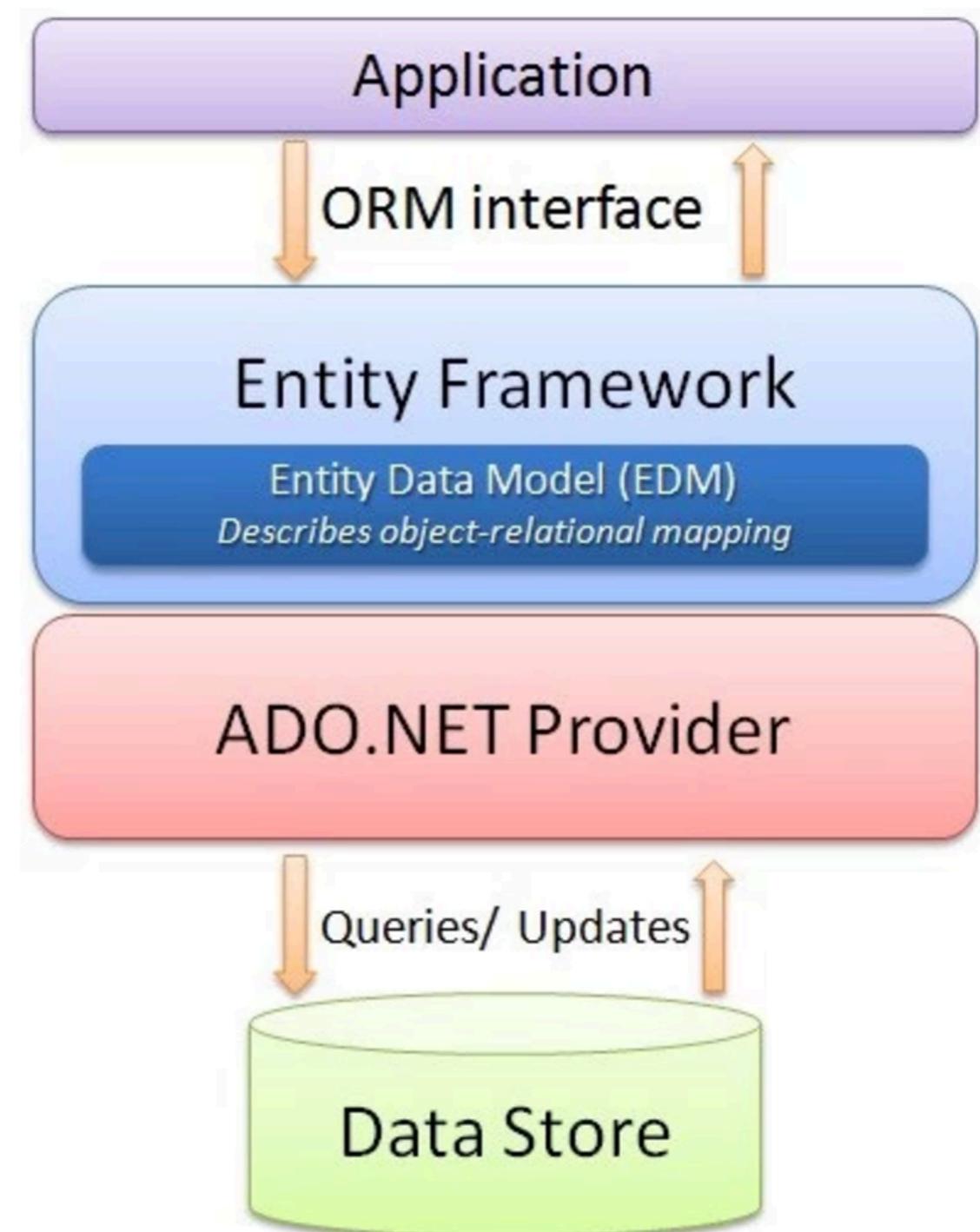
```
select cat.color, sum(cat.weight), count(cat) from Cat cat  
group by cat.color
```

- order by clause
- group by clause
- expressions
- etc.

# Exemplu Hibernate

# .NET ORM

- Instrumente ORM bazate pe LINQ:
  - [LINQ to SQL \(L2S\)](#)
  - [Entity Framework \(EF\)](#)
- EF permite o mai bună decuplare a claselor de modelul relational



# .NET L2S

- Decorarea claselor folosind attribute .NET
- Spațiul de nume `System.Data.Linq.Mapping`

```
[Table (Name="Customers")]
public class Customer
{
    [Column(IsPrimaryKey=true)]
    public int ID {get;set;}
    [Column (Name="FullName")]
    public string Name {get; set;}
}
```

# .NET L2S

```
var context = new DataContext ("database connection string");

Table<Customer> customers = context.GetTable <Customer>();

// numărul de înregistrări din tabelă.
Console.WriteLine (customers.Count());

// Clientul cu Id-ul 2.
Customer cust = customers.Single (c => c.ID == 2);

Customer cust = customers.OrderBy (c => c.Name).First();
cust.Name = "Updated Name";
context.SubmitChanges();
```

# .NET EF

- Decorarea claselor folosind atribute .NET
- Referință către [System.Data.Entity.dll](#)

```
[EdmEntityType (NamespaceName = "EFModel", Name = "Customer")]
public partial class Customer
{
    [EdmScalarPropertyAttribute (EntityKeyProperty=true,
        IsNullable=false)]
    public int ID { get; set; }

    [EdmScalarProperty (EntityKeyProperty = false, IsNullable = false)]
        public string Name { get; set; }
}
```

# .NET EF

```
var context = new ObjectContext ("entity connection string");
    context.DefaultContainerName = "EntitiesContainer";
ObjectSet<Customer> customers =
context.CreateObjectSet<Customer>();

// numărul de înregistrări din tabelă
Console.WriteLine (customers.Count ());

// Clientul cu ID-ul 2
Customer cust = customers.Single (c => c.ID == 2);

Customer cust = customers.OrderBy (c => c.Name).First ();
cust.Name = "Updated Name";
context.SaveChanges ();
```

# Medii de proiectare și programare

2022-2023

Curs 10

# Continut

- Servicii Web
- Arhitectura REST
- REST Spring

# Servicii Web

- Definiții:

1. Un **serviciu web** este o aplicație/componentă soft disponibilă pe internet și care folosește un sistem standardizat de transmitere a mesajelor în format XML. XML este folosit pentru comunicarea datelor către/de la un serviciu web.
  - Un client apelează un serviciu web trimițând un mesaj în format XML și așteaptă un răspuns în format XML.
  - Comunicarea are loc folosind XML, serviciile web nu sunt dependente de un anumit sistem de operare sau de un anumit limbaj de programare.
2. **Serviciile web** sunt aplicații *self-contained*, modulare și distribuite care pot fi descrise, publicate, localizate și apelate prin rețea pentru a crea produse, procese, etc. Aceste aplicații pot fi locale, distribuite sau pe web. Serviciile web sunt dezvoltate folosind standarde ca TCP/IP, HTTP și XML.
3. **Serviciile web** sunt sisteme de schimbare a mesajelor bazate pe XML care folosesc Internetul pentru interacțiunea dintre aplicații. Aceste sisteme pot fi: programe, obiecte, mesaje, documente, etc.
4. Un **serviciu web** este o colecție de protocoale și standarde folosite pentru transmiterea datelor între aplicații sau sisteme. Aplicații dezvoltate în diferite limbi de programare și rulând pe diferite platforme (sisteme de operare) pot folosi serviciile web pentru a schimba date în rețea (ex. Internet) într-o manieră similară comunicării între procese pe același calculator. Interoperabilitatea acestora este posibilă datorită folosirii standardelor.

# Servicii Web

- Un [serviciu web](#) este un [serviciu](#):
  - Disponibil pe Internet sau într-o rețea privată.
  - Folosește un sistem standardizat de schimbare a mesajelor bazat pe XML.
  - Nu este dependent de un anumit limbaj de programare sau de un anumit sistem de operare.
  - Este self-describing folosind o gramatică XML (DTD, XML Schema).
  - Poate fi descoperit folosind mecanisme simple.
- [Componentele](#): structura de baza pentru serviciile web este XML + HTTP. Toate serviciile web standard funcționează folosind componente:
  - [XML](#) pentru marcarea informației
  - [SOAP](#) (Simple Object Access Protocol) pentru transmiterea mesajelor
  - [UDDI](#) (Universal Description, Discovery and Integration)
  - [WSDL](#) (Web Services Description Language) pentru a descrie disponibilitatea serviciului

# Exemplu WSDL

```
<definitions name = "HelloService"
  targetNamespace = "http://www.examples.com/wsdl/HelloService.wsdl"
  xmlns = "http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap = "http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns = "http://www.examples.com/wsdl/HelloService.wsdl"
  xmlns:xsd = "http://www.w3.org/2001/XMLSchema">

  <message name = "SayHelloRequest">
    <part name = "firstName" type = "xsd:string"/>
  </message>

  <message name = "SayHelloResponse">
    <part name = "greeting" type = "xsd:string"/>
  </message>

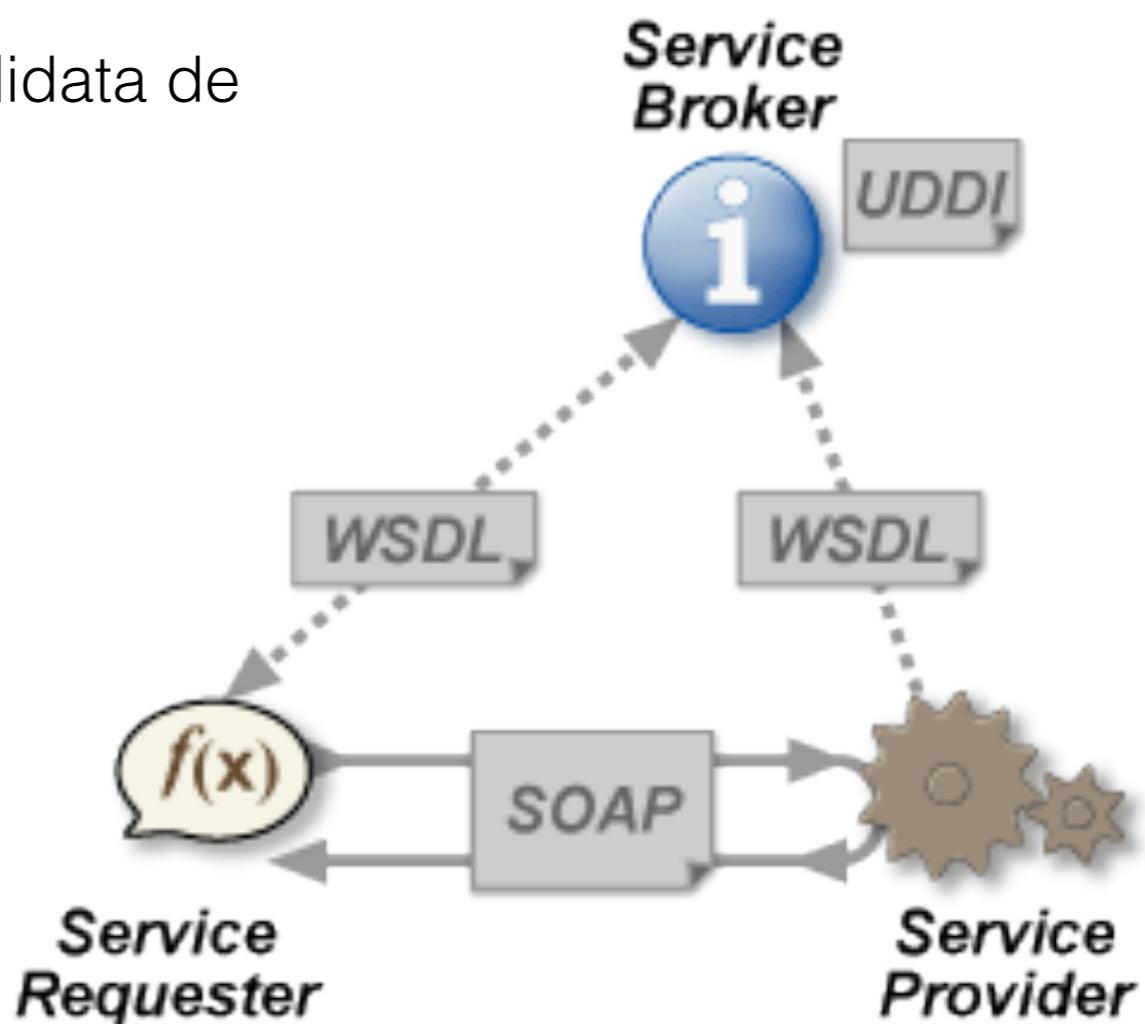
  <portType name = "Hello_PortType">
    <operation name = "sayHello">
      <input message = "tns:SayHelloRequest"/>
      <output message = "tns:SayHelloResponse"/>
    </operation>
  </portType>

  <binding name = "Hello_Binding" type = "tns:Hello_PortType">
    <soap:binding style = "rpc"
      transport = "http://schemas.xmlsoap.org/soap/http"/>
    <operation name = "sayHello">
      <soap:operation soapAction = "sayHello"/>
      <input>
        <soap:body
          encodingStyle = "http://schemas.xmlsoap.org/soap/encoding/"
          namespace = "urn:examples:helloservice"
          use = "encoded"/>
      </input>
      <output>
        <soap:body
          encodingStyle = "http://schemas.xmlsoap.org/soap/encoding/"
          namespace = "urn:examples:helloservice"
          use = "encoded"/>
      </output>
    </operation>
  </binding>

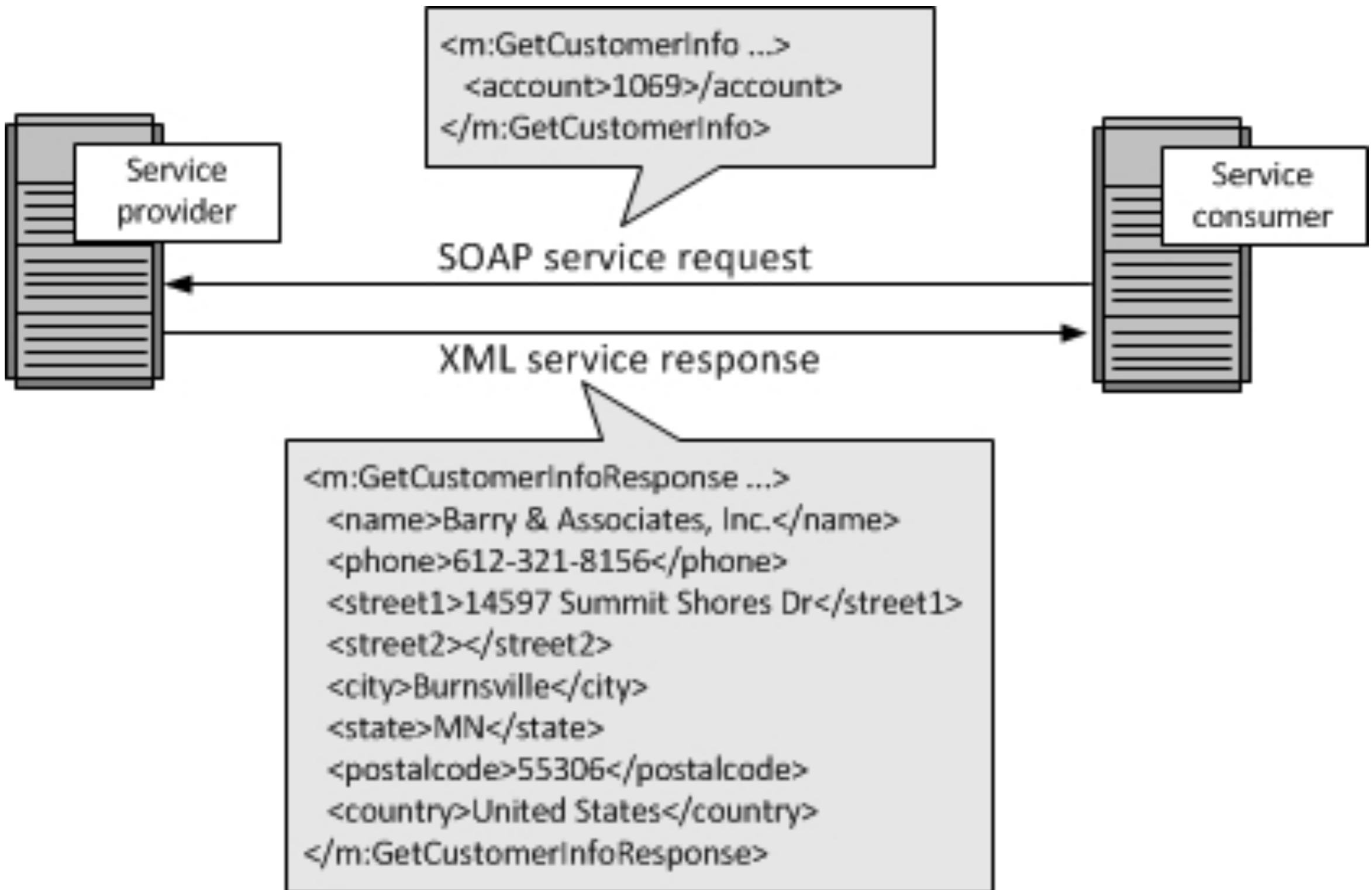
  <service name = "Hello_Service">
    <documentation>WSDL File for HelloService</documentation>
    <port binding = "tns:Hello_Binding" name = "Hello_Port">
      <soap:address
        location = "http://www.examples.com/SayHello/" />
    </port>
  </service>
</definitions>
```

# Servicii Web - Arhitectura

- Furnizorul serviciului trimit un fișier WSDL serviciului UDDI.
- Clientul serviciului (consumatorul) contactează UDDI pentru a descoperi cine este furnizorul datelor de care are nevoie, iar apoi contacteaza furnizorul serviciului folosind protocolul SOAP.
- Furnizorul serviciului validează cererea și trimit datele cerute în format XML folosind protocolul SOAP.
- Structura datelor în format XML ar trebui validata de client folosind un fișier XSD (XML Schema).



# Servicii Web - Arhitectura



# REST

- REpresentational State Transfer (REST).
- Este un stil arhitectural pentru sisteme distribuite hipermedia.
- A fost propus de Roy T. Fielding în teza sa de doctorat “*Architectural Styles and the Design of Network-based Software Architectures*” (2000).
- Conține 6 constrângeri ce trebuie satisfăcute pentru a fi numit sistem REST veritabil:
  1. Client–server
  2. Fără stare (eng. *stateless*)
  3. Cacheable
  4. Interfață uniformă (eng. *uniform interface*)
  5. Sistem stratificat (eng. *layered system*)
  6. Cod la cerere (eng. *code on demand*) -optională

# REST - Resursa

- Conceptul de bază în REST este *resursa* (eng. *resource*).
- Orice informație care poate fi numită poate fi **resursă**: un document, o imagine, o colecție de alte resurse, un obiect real (ex. persoană, mașină), etc.
- REST folosește *identificatorul resursei* pentru a identifica resursa în momentul interacțiunii între componente.
- Starea unei resurse la un anumit moment de timp este numită **reprezentarea resursei**.
- Reprezentarea constă din *date*, *metadata* care descriu datele și *legături hipermedia* care ajută clientii în tranzitia la următoarea stare dorită.
- Formatul datelor dintr-o reprezentare este numit **tip media** (eng. *media type*).
- Tipul media identifică o specificație care definește modul în care reprezentarea va fi procesată.
- Un API RESTful veritabil seamănă cu hypertext. Fiecare informație ce poate fi obținută are *o adresă explicită* (prin legături și atribut id) sau *implicită* (obținută din definiția tipului media și structura reprezentării).
- **Reprezentările resurselor trebuie să se descrie pe ele** (eng. *self-descriptive*): clientul nu trebuie să știe că resursa este persoană sau mașină. Datele trebuie să poate fi procesate pe baza tipului media asociat resursei.

# REST - metode asupra resursei

- Un lucru important asociat cu REST sunt **metodele/operațiile** ce pot fi aplicate **resursei** pentru a efectua tranziția dorită.
- Toată **informația** necesară modificării stării resursei face parte din API-ul cererii pentru acea resursă (inclusiv operațiile/metodele și modul în care vor afecta reprezentarea resursei).
- Este **recomandat** ca rezultatele interogărilor să fie reprezentate folosind o listă cu legături conținând informații sumare, și **nu tablouri** conținând **toate informațiile corespunzătoare resursei**, deoarece interogarea nu este un substitut pentru identificarea resurselor.
- Adesea, metodele/operațiile asupra resursei sunt confundate cu metodele HTTP GET/PUT/POST/DELETE.

# REST și HTTP

- REST și HTTP nu reprezintă același lucru.
- Dacă un sistem satisface constrângerile REST, se poate spune ca sistemul este RESTful (modelul de maturitate Richardson, 2009).
- În stilul arhitectural REST, datele și funcționalitățile sunt considerate resurse ce pot fi accesate folosind *Uniform Resource Identifiers* (URIs).
- Resursele sunt prelucrate (adăugate, șterse, modificate, etc) folosind un set simplu, bine definit de operații.
- Clientii și serverele interschimbă (schimbă între ele) reprezentări ale resurselor folosind o interfață standardizată și un protocol standardizat (adesea HTTP).
- Resursele sunt decuplate de reprezentarea lor. Conținutul lor poate fi accesat folosind diferite formate: XML, text, PDF, JSON, HTML, etc.
- Metadata despre resurse este folosită pentru detecția erorilor, controlul caching-ului, negocierea celui mai convenabil format pentru reprezentare, autentificare și controlul accesului.
- Fiecare interacțiune este **stateless**.
- Aceste principii fac aplicațiile RESTful simple, lightweight și rapide.

# Constrângerile arhitecturale REST

- REST este un stil arhitectural ce permite proiectarea unor aplicații slab cuplate (loosely coupled ) folosind HTTP.
- Este adesea folosit pentru dezvoltarea serviciilor web.
- REST nu impune nici o regulă despre cum ar trebui implementat la nivelele inferioare, doar impune constrângeri la nivelul de proiectare și lasă implementarea dezvoltatorului.
- REST definește 6 constrângeri arhitecturale a căror satisfacere fac orice serviciu web un API RESTful veritabil:
  1. *Uniform interface*
  2. *Client–server*
  3. *Stateless*
  4. *Cacheable*
  5. *Layered system*
  6. *Code on demand* (optional)

# REST - Uniform interface

- Aplicând principiul generalității interfeței componentelor care interacționează, arhitectura întregului sistem este simplificată și interacțiunea între componente este îmbunătățită.
- Pentru a obține o interfață uniformă, se folosesc mai multe constrângeri arhitecturale pentru a ghida execuția componentelor.
- REST este definit de 4 constrângeri ale interfeței:
  - *Identificarea resurselor*
  - *Manipularea resurselor folosind reprezentările*
  - *Mesaje ce se descriu pe sine (self-descriptive)*
  - *Hypermedia ca și motor al stării aplicației.*
- Dezvoltatorii trebuie să decidă interfață pentru resursele sistemului făcute publice clienților și să folosească întotdeauna această interfață.
- O resursă din sistem trebuie să aibă un singur URI, și trebuie să ofere o modalitate de a obține date adiționale sau asociate.
- O resursă nu ar trebui să fie foarte mare și ar trebui să conțină legături către alte informații adiționale (URI relativ) care permit obținerea acelor informații.
- Reprezentările resurselor trebuie să respecte anumite recomandări (convenții de nume, formatul legăturilor, formatul datelor -xml și/sau json).
- Toate resursele ar trebui să fie accesibile folosind o abordare comună (ex. folosind HTTP GET) și, în mod similar, ar trebui să poată fi modificate folosind o abordare consistentă.

# REST - Client–server

- Aplicațiile client și server trebuie să poată evoluă independent fără dependențe între ele.
- Clientul ar trebui să știe doar de URI-ul resursei.
- Prin separarea interfeței clientului de modul de păstrare a resurselor, se îmbunătățește portabilitatea interfeței cu utilizatorul pe platforme diferite și se îmbunătățește scalabilitatea prin simplificarea componentelor de pe server.
- “*Servers and clients may also be replaced and developed independently, as long as the interface between them is not altered.*”

# REST - Stateless

- Stilul REST a fost inspirat din HTTP.
- Toate interacțiunile client-server trebuie să fie **stateless**.
- Serverul nu va păstra informații despre ultima cerere HTTP de la client.
- Fiecare cerere va fi tratată ca și cum ar fi o cerere nouă (fără sesiune, fără istoric).
- Fiecare cerere trimisă de la client la server trebuie să conțină toate informațiile necesare pentru a înțelege cererea și nu poate folosi informații legate de context păstrate pe server.
- Starea sesiunii este păstrată în întregime la client.
- Dacă o aplicație trebuie să păstreze starea pentru client (clientul se autentifică iar apoi efectuează operații), fiecare cerere de la client trebuie să conțină toate datele necesare pentru îndeplinirea cererii (inclusiv detalii legate de autentificare și autorizare).
- “*No client context shall be stored on the server between requests. Client is responsible for managing the state of application.*”

# Constrângeri REST

- *Cacheable*: constrângerea cere ca datele dintr-un răspuns la o cerere să fie **implicit sau explicit marcate ca cacheable sau non-cacheable**. Dacă răspunsul este cacheable, atunci cache-ul de pe client are permisiunea de a refolosi datele ulterior (echivalând cu obținerea lui folosind cereri către server).
- Caching îmbunătășește performanța clientului și îmbunătășește scalabilitatea serverului deoarece numărul de cereri se reduce.
- Caching trebuie aplicat tuturor resurselor ce se declară cacheable. Caching poate fi implementat atât pe server cât și pe client.
- *“Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance.”*
- *Sistem stratificat*:
  - REST permite folosirea unei arhitecturi stratificate a sistemului: API public (URIs) este pe serverul A, datele sunt păstrate pe server B, iar cererile de autentificare se fac pe serverul C.
  - Un client nu știe dacă este conectat direct cu serverul destinație sau cu un intermediar.
- Arhitectura stratificată permite ca sistemul final să fie compus din nivele ierarhice care permit unui nivel comunicarea doar cu nivelul inferior (superior)

# Constrângeri REST

- *Code on demand*
- Constrângerea este **optională**. Adesea, serverele vor trimite reprezentarea statică a resurselor în format XML sau JSON.
- Serverul poate trimite și cod executabil, pentru a suporta o parte a aplicației (ex. cod pentru afișarea unei componente grafice)
- REST permite extinderea funcționalității clienților prin descărcarea și executarea codului (sub forma unor applet-uri sau a unor scripturi).
- Poate simplifica clienții prin reducerea numărului de funcționalități ce trebuie implementate ulterior.

# Recomandări resurse REST

- Se recomandă folosirea consecventă a unei strategii pentru denumirea resurselor.
- API REST folosește URIs pentru accesarea resurselor. URI-urile folosite ar trebui să sugereze clientilor modelul resurselor.
- Dacă resursele sunt denumite sugestiv, API-ul corespunzător va fi ușor de folosit și intuitiv. Altfel poate fi dificil de înțeles și utilizat.
- O resursă poate fi *singulară* sau *o colecție de alte resurse*.

Exemplu:

- “**articles**” - colecție de resurse
- “**article**” - singular.

- Identificarea resurselor de tip colecție se poate face folosind substantivul la plural:

Resursa “**articles**” : URI “**/articles**”.

- Identificarea unei resurse singulare se poate face folosind identificatorul:

Resursa “**article**” are URI “**/articles/{articleID}**”.

- O resursă poate conține o colecție de alte resurse (subcolecție).

ex. autorii unui articol pot fi identificați prin:

“**/articles/{articleID}/authors**”.

- Identificarea unei resurse singulare dintr-o subcolecție:

“**/articles/{articleID}/authors/{authorID}**”.

# REST Recomandări

- *Folosirea substantivelor pentru reprezentarea resurselor*
- **URI RESTful trebuie să refere resursa, nu să se refere la o acțiune.** Substantivele au proprietăți, și asemănător resursele au attribute.

`http://api.example.com/conference/articles`

`http://api.example.com/conference/getArticles` - nu este URI corespunzător REST

`http://api.example.com/conference/articles/{id}/title`

`http://api.example.com/user-management/users`

`http://api.example.com/user-management/users/{id}`

`http://api.example.com/user-management/users/{id}/name`

- *Folosirea consecventă a convențiilor pentru denumirea resurselor și a modului de formare a URI-ului pentru a reduce ambiguitățile și îmbunătățirea înțelegerii și a întreținerii:*

- Folosirea caracterului / pentru a indica relații ierarhice

- A NU se folosi caracterul / la finalul URI-ului

`http://api.example.com/device-management/managed-devices/`

`http://api.example.com/device-management/managed-devices`

# REST Recomandări

- *Folosirea cratimei (-) pentru îmbunătățirea înțelegerii URI*

http://api.example.com/inventory-management/managed-entities/{id}/install-script-location

http://api.example.com/inventory-management/managedEntities/{id}/installScriptLocation

- *Folosirea literelor mici în URI*

http://api.example.org/my-folder/my-doc

http://api.example.org/My-Folder/my-doc

- *A nu se folosi extensii de fișier:* Nu aduc informații suplimentare clientului și pot induce eroare dezvoltatorii.
- Pentru a preciza *tipul media* corespunzător unei reprezentări se folosesc antetele *Accept* și *Content-Type*.
- Acestea sunt folosite și pentru a determina modul de procesare a conținutului.

http://api.example.com/device-management/managed-devices.xml

http://api.example.com/device-management/managed-devices

# REST Recomandări

- **A nu se folosi nume de funcții CRUD (get, add/save, delete, update/modify) în URI-uri**
- URI-urile nu ar trebui folosite pentru a preciza operația/operațiile efectuate.
- URI-urile ar trebui folosite pentru a identifica în mod unic resursele, nu pentru a le manipula.
- Metodele HTTP ar trebui folosite pentru a indica operația CRUD ce trebuie efectuată.

//obținerea tuturor articolelor (echivalentul lui getAll/findAll)

HTTP GET și URI `http://api.example.com/conference/articles`

//Crearea unui nou articol (echivalentul lui save/add)

HTTP POST și URI `http://api.example.com/conference/articles`

//Obținerea articolului cu id-ul dat (echivalentul lui findOne)

HTTP GET și URI `http://api.example.com/conference/articles/{id}`

//Actualizarea articolului cu id-ul dat (echivalentul lui update/modify)

HTTP PUT și URI `http://api.example.com/conference/articles/{id}`

//Stergerea articolului cu id-ul dat (echivalentul lui delete)

HTTP DELETE și URI `http://api.example.com/conference/articles/{id}`

# REST Recomandări

- *Folosirea parametrilor interogării HTTP pentru a filtra o colecție.*
  - Sortarea elementelor din colecție
  - Filtrarea elementelor din colecție
  - Limitarea numărului de elemente returnate (paginarea)
  - Nu se creează URI-uri noi, ci se folosesc parametrii de tip query:

`http://api.example.com/conference/articles`

`http://api.example.com/conference/articles?domain=AI`

`http://api.example.com/conference/articles?domain=AI&subdomain=Genetic`

`http://api.example.com/conference/articles?  
domain=AI&subdomain=Genetic&sort=submission-date`

`//gresit`

`http://api.example.com/conference/getArticlesSortedBy?  
domain=AI&subdomain=Genetic&sort=submission-date`

# REST API - Proiectare

- *Pașii proiectării serviciilor REST*

1. Identificarea modelului obiectual
2. Crearea modelului pentru URI-uri
3. Determinarea reprezentărilor
4. Atribuirea/asocierea metodelor HTTP
5. Alte acțiuni (Logging, Security, Discovery)

- Identificarea modelului obiectual

- Identificarea obiectelor care vor fi prezentate ca și resurse:

- Users
  - Messages,
  - Articles,
  - Authors,
  - Persons
  - etc.

# REST API Proiectare

- Crearea modelului URI-urilor
  - Deciderea URI-urilor pentru resursele identificate anterior.
  - Axarea pe relațiile dintre resurse și sub-resurse.
  - Aceste URI-uri asociate resurselor vor fi folosite ca și endpoint-uri pentru serviciile REST.
  - **URI-urile nu conțin verbe sau operații.**
  - **URI-urile ar trebui să conțină doar substantive.**

`/articles`

`/articles/{id}`

`/authors`

`/authors/{id}`

`/articles/{id}/authors`

`/articles/{id}/authors/{aid}`

# REST API - Proiectare

- Determinarea reprezentării resurselor

- Majoritatea reprezentărilor sunt definite în format XML sau JSON.

```
<author>
  <name> Pop Ion </name>
  <affiliation> zY Lab </affiliation>
</author>          {
                    "name": "Pop Ion"
                    "affiliation": "zY Lab"
}
```

- Asocierea metodelor HTTP

- Obținerea tuturor articolelor sau autorilor

**HTTP GET** și URI **/articles**

**HTTP GET** și URI **/authors**

- Paginarea, dacă rezultatul este prea mare.

**HTTP GET** și URI **/articles?startIndex=0&size=20**

**HTTP GET** și URI **/authors?startIndex=0&size=20**

- Obținerea tuturor autorilor unui articol (subcolecție)

**HTTP GET** și URI **/articles/{id}/authors**

# REST API -Proiectare

- Obținerea unui singur articol/autor

**HTTP GET** și URI `/articles/{id}`

**HTTP GET** și URI `/authors/{id}`

- Obținerea unui singur autor (subcolecție)

**HTTP GET** și URI `/articles/{id}/authors/{id}`

Reprezentarea subresursei poate fi diferită.

- Crearea unui articol/autor - metoda HTTP POST

**HTTP POST** și URI `/articles`

**HTTP POST** și URI `/authors`

## IMPORTANT!

- **De obicei, la crearea unei resurse, corpul cererii NU conține informații legate de id, deoarece serverul este responsabil de determinarea acestuia.**
- **Exceptie: id-ul este informatie relevantă pentru resursa (username, CNP)**

# REST API - Proiectare

- Actualizarea unui articol/autor - metoda HTTP PUT

**HTTP PUT** si URI `/articles/{id}`

**HTTP PUT** si URI `/authors/{id}`

- Ștergerea unui articol/autor - metoda HTTP DELETE

**HTTP DELETE** si URI `/articles/{id}`

**HTTP DELETE** si URI `/authors/{id}`

- Un răspuns ar trebui să returneze:

- 202 (Accepted) dacă resursa a fost pusă într-o coadă și urmează a fi ștearsă (asincron).
- 200 (OK) / 204 (No Content) dacă resursă a fost ștearsă permanent (syncron).

- Adăugarea unui autor la articol - Metoda HTTP PUT

**HTTP PUT** si URI `/articles/{id}/authors`

- Ștergerea unui autor de la articol - Metoda HTTP DELETE

**HTTP DELETE** si URI `/articles/{articleId}/authors/{authorId}`

# HTTP Response Status Code

- **2XX -Success** acțiunea cerută de client a fost primită, înțeleasă, acceptată și procesată cu succes.
  - **200 OK:** Răspunsul standard pentru cereri HTTP executate cu succes.
  - **201 Created:** Cererea a fost îndeplinită, și a fost creată o nouă resursă.
  - **202 Accepted:** Cererea a fost acceptată pentru procesare, dar procesarea nu a fost încă efectuată.
  - **204 No Content:** Serverul a procesat cererea cu succes, dar procesarea nu a returnat date.
  - **205 Reset Content:** Serverul a procesat cererea cu succes, dar nu a returnat date. Spre deosebire de răspunsul 204, acest răspuns necesită ca clientul să reafifice vederea/informatia.
  - etc.

# HTTP Response Status Code

- **4XX** Erori client:
  - **400 Bad Request:** Serverul nu poate sau nu va procesa cererea din cauza unei erori din partea clientului (ex. sintaxa cererii este greșită, conținutul cererii este prea mare, etc.)
  - **401 Unauthorized:** Similar cu 403, dar se folosește când este necesară autentificarea, care încă nu s-a efectuat sau a eșuat.
  - **403 Forbidden:** Cererea este validă, dar serverul refuză procesarea. (Utilizatorul nu are drepturile necesare pentru a accesa resursa respectivă).
  - **404 Not Found:** Resursa ceruta nu a fost găsită, dar poate deveni disponibilă în viitor. Sunt permise cereri ulterioare din partea clientului.
  - **405 Method Not Allowed:** Metoda cerută nu este disponibilă pentru resursa cerută. Ex. o cerere GET pentru un formular(eng. *form*) care necesită ca datele să fie transmise folosind POST, sau o cerere PUT pentru o resursă care poate fi doar citită.
  - **406 Not Acceptable:** Reprezentările disponibile pentru resursa cerută nu sunt acceptate de client (folosind antetul *Accept*).
  - etc.
- **5XX** Erori la implementarea serviciilor

# HTTP Media Types

- Antetele HTTP **Accept** și **Content-Type** pot fi folosite pentru a descrie reprezentarea folosită pentru schimbarea informației (reprezentarea resurselor).
- Clientul trimite o cerere în care în antetul **Accept** specifică lista tipurilor de reprezentare acceptate (separate prin virgulă).

Exemplu : **Accept: application/json, application/xml**

- Serverul trimite răspunsul folosind ca și reprezentare primul tip mime întăres din lista precizată (dacă întelege unul sau mai multe). Tipul folosit este precizat în antetul **Content-Type**.

Exemplu: **Content-Type: application/xml**

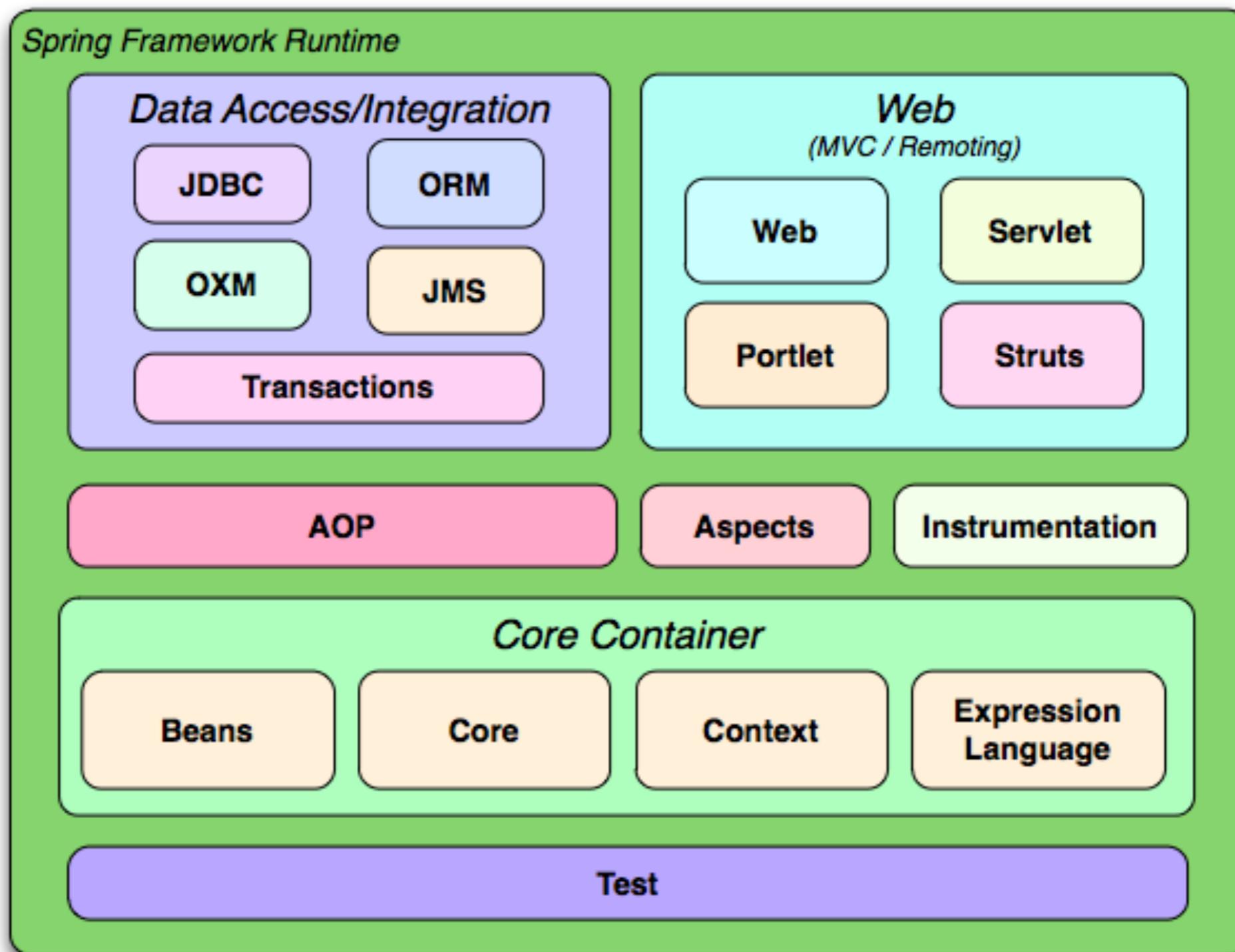
# Referinte,

- Roy Fielding, PhD Thesis, Chapter 5. Representational State Transfer (REST)

[https://www.ics.uci.edu/~fielding/pubs/dissertation/rest\\_arch\\_style.htm](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm)

- <http://restfulapi.net/>
- <http://www.service-architecture.com/articles/web-services/>
- Alte tutoriale

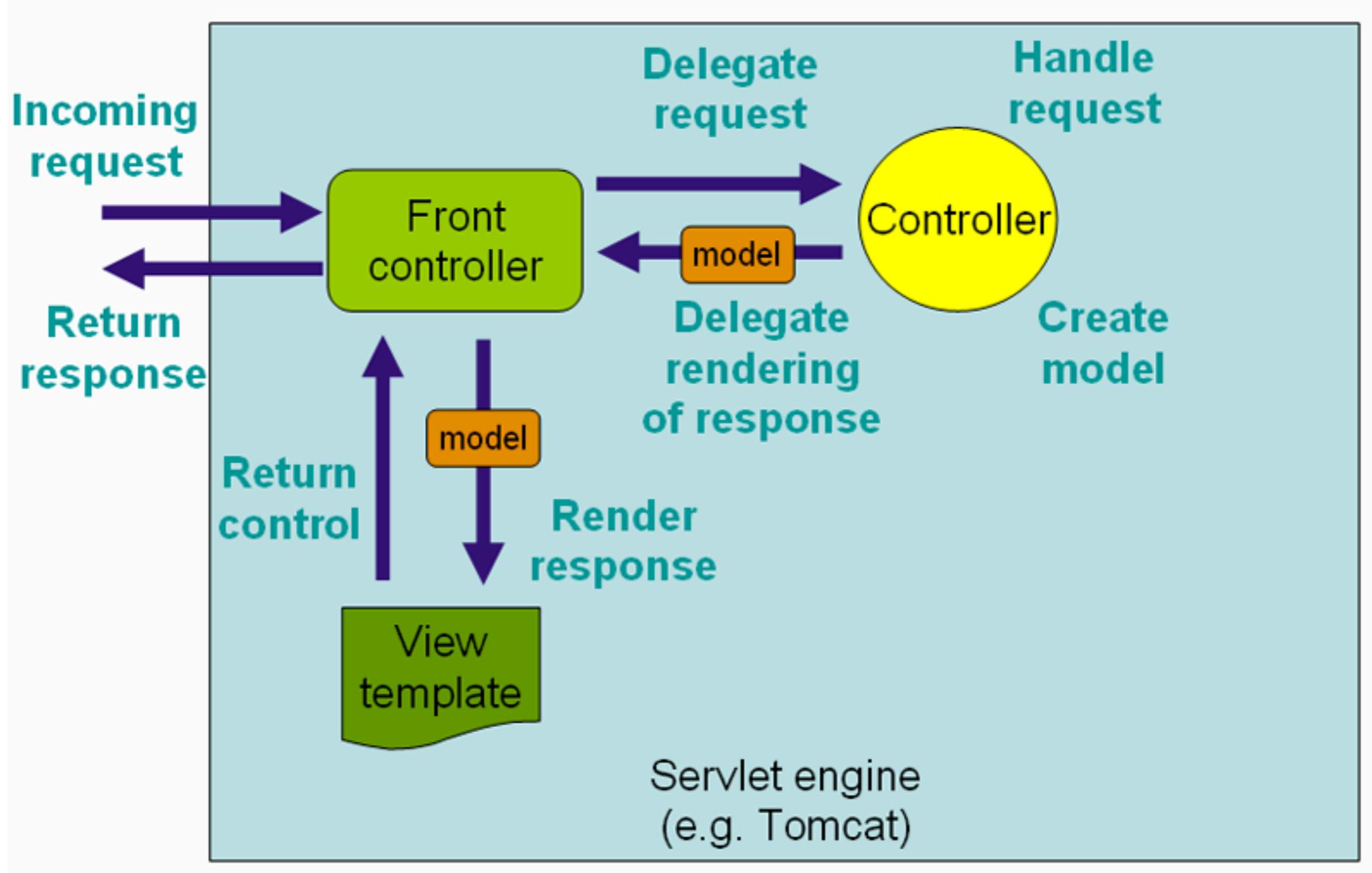
# Spring Web MVC



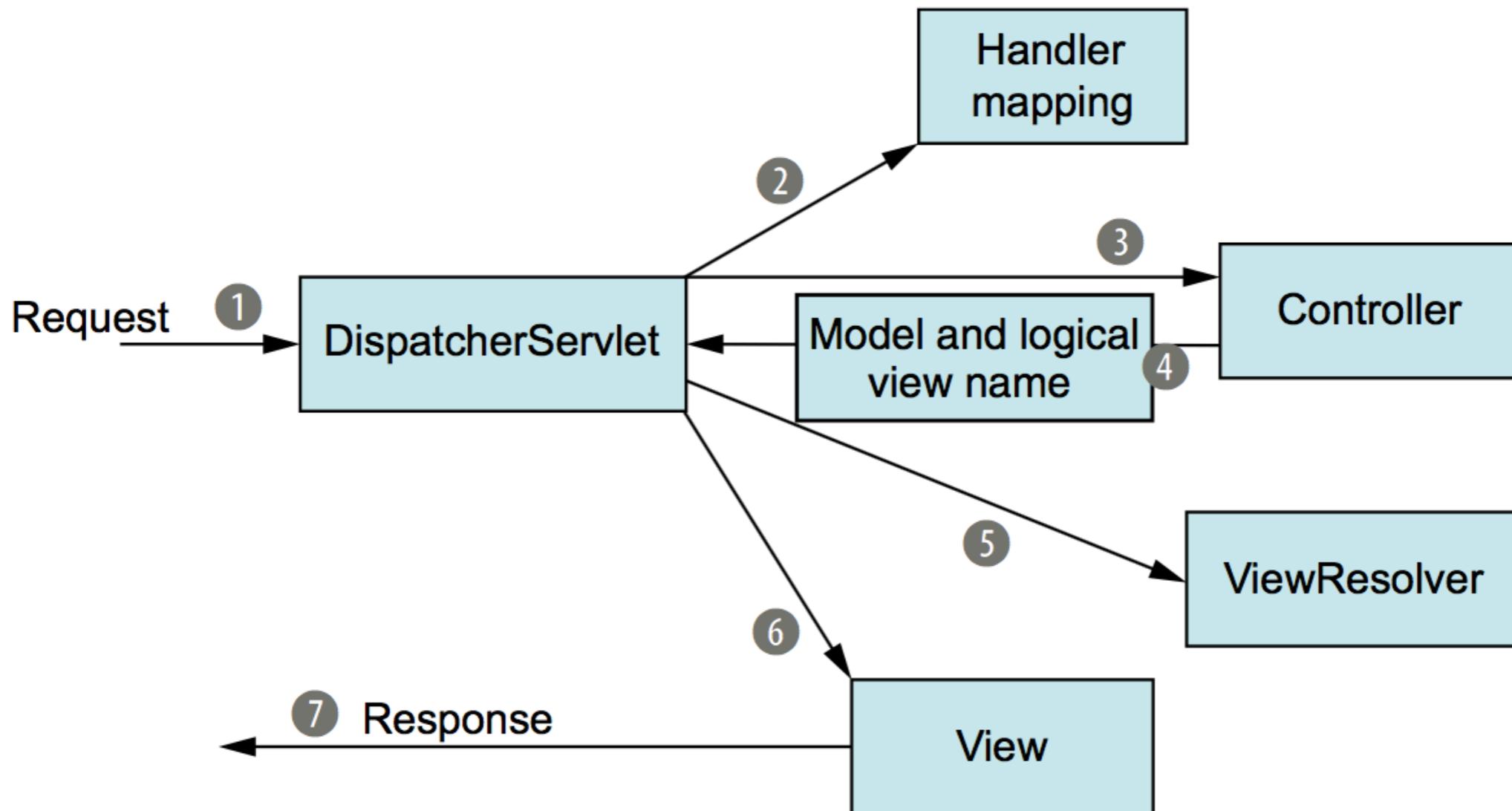
# Spring Web MVC

- Frameworkul Spring Web model-view-controller (MVC) este proiectat în jurul unui *DispatcherServlet* care trimitе cererile handler-elor, cu mapări configurabile, soluționarea vederilor (eng. *view*), a temei, a datelor și suport pentru încărcarea fișierelor.
- Handler-ul implicit folosește adnotările `@Controller` și `@RequestMapping` ce permit folosirea diferitor metode pentru procesarea cererilor.
- Orice obiect poate fi folosit ca și comandă sau pentru procesarea unei forme.
- Soluționarea vederilor este foarte flexibilă.
- Un controller este responsabil cu pregătirea modelului ce conține datele ce urmează a fi afișate și cu selectarea vederii folosită pentru afișare (folosind un nume pentru vedere).
- Soluționarea numelui asociat unei vederi se poate face prin extensii de fișiere, folosind antetul `Accept`, fișiere de proprietăți, bean-uri, etc.
- Modelul care păstrează datele este o interfață Map, care permite decuplarea de tehnologia folosită pentru afișarea acestora.

# Spring Web MVC



# Spring Web MVC



# Spring Web MVC - RestController

- Clase anotate cu **@RestController**
- Adnotarea **@RestController**: metodele anotate din controller sunt considerate ca fiind HTTP endpoints și folosesc metadata furnizată de adnotarea **@RequestMapping** (**@GetMapping**, **@PostMapping**, etc) folosită la fiecare metodă/clasă pentru identificarea metodei ce trebuie să proceseze o cerere.
- O metodă va fi folosită pentru procesarea unei cereri HTTP dacă cererea satisface condițiile precizate folosind adnotarea **@RequestMapping/@GetMapping/...** asociată metodei.
- Adnotarea **@RestController** furnizează și valori implicate (informații metadata) pentru toate metodele din clasa anotată.
- Fiecare metodă poate decide suprascrierea adnotărilor specificate la nivel de clasă, dar anumite informații depind de context.

# Spring Web MVC - RestController

```
@RestController  
  
@RequestMapping("/articles/{articleId}/authors")  
  
public class AuthorRestController {  
  
    @RequestMapping(method = RequestMethod.GET) // @GetMapping  
  
    Collection<Author> readAuthors(@PathVariable String articleId,  
  
        @RequestParam(value="sorted", defaultValue="name")String sortCrit) { ... }  
  
    @RequestMapping(method = RequestMethod.POST) // @PostMapping  
  
    ResponseEntity<?> add(@PathVariable String articleId, @RequestBody Author input) { ... }  
  
    @RequestMapping(value = "/{authorId}", method = RequestMethod.GET)  
  
    Author readAuthor(@PathVariable String articleId, @PathVariable Long authorId) { ... }  
  
    private void validateUser(String userId) { ... }  
}
```

# Spring Web MVC - RestController

- **{articleId}** și **{authorId}** sunt **variabile de cale** (eng. *path variables*). Sunt asemănătoare caracterelor speciale (eng. *wildcards*).
- Spring MVC va extrage acele părți din URI și le va face disponibile ca și parametrii ai metodelor ce procesează cererea. **Parametrii vor avea același nume cu variabila de cale și sunt adnotăți cu anotarea @PathVariable.**

Exemplu: O cerere HTTP GET cu URI **/articles/art123/authors/4234**

- Parametrul **@PathVariable String articleId** va avea valoarea **art123**,
- Parametrul **@PathVariable Long authorId** va avea valoare **4234**.
- **@RequestParam** indică **parametrii cererii** (eng. *request query parameters*).

Exemplu: O cerere HTTP GET cu URI **/art123/authors/4234?sorted=affiliation**

- **@RequestParam String sortCrit** va avea valoarea **affiliation**

# Tipul parametrilor /returnat

- Datele returnate după procesarea cererii pot avea orice tip.
  - Tip primitiv/ obiect: informația va fi convertită la reprezentarea negociată de client și server (json, xml, text, etc).
  - @RequestBody: Datele transmise de client vor fi convertite la tipul precizat.
  - RequestEntity<E>: Este un wrapper pentru cererea primită de la client. Conține reprezentarea informației și datele adiționale din cererea HTTP (antetele, dimensiunea, etc).
  - @ResponseBody: Obiectul returnat va fi convertit la reprezentarea negociată.
  - ResponseEntity<E>: Este un wrapper pentru obiectul de tipul E și optional poate seta tipul de răspuns (HTTP Response Status Code) și alte antete HTTP.

# Tipul returnat

```
@RequestMapping(method = RequestMethod.GET)
String greeting(@RequestParam(value="name" defaultValue "") String name ){
    return "Hello "+name+"!";
}

@RequestMapping(method = RequestMethod.GET)
@ResponseBody String greeting(@RequestParam(value="name" defaultValue "") String name ){
    return "Hello "+name+"!";
}

@RequestMapping(method = RequestMethod.GET)
 ResponseEntity<String> greeting(@RequestParam(value="name" defaultValue "") String name )
{
    return new ResponseEntity<>("Hello "+name+"!");
}

@RequestMapping(method = RequestMethod.GET)
 ResponseEntity<String> greeting(RequestEntity<String> request, @RequestParam(value="name"
defaultValue "") String name ){ ...}
```

# Convertirea mesajelor

- Convertirea mesajelor este o modalitate de a transforma datele obținute de controller/de la controller într-o reprezentare ce va fi transmisă clientului.
- Spring conține diferite clase pentru convertirea mesajelor, pentru a putea trata cele mai întâlnite conversii obiect-reprezentare.
  - **BufferedImageHttpMessageConverter**: Obiect **BufferedImage** din/într-o imagine (păstrată ca și un tablou de octeți).
  - **MappingJackson2HttpMessageConverter**: Obiecte simple și HashMaps (String, <?>) în format JSON.
  - **MarshallingHttpMessageConverter** (XML)
  - **StringHttpMessageConverter** (String, text/plain)

# Tratarea exceptiilor

```
@RequestMapping(value = "/{id}", method = RequestMethod.GET)

public Article getById(@PathVariable String id){

    return articleRepository.findById(); //daca id-ul nu există, returnează null
}
```

- Id-ul nu există, răspunsul va fi 200 (OK), iar conținutul va fi null.

Soluția 1

```
@RequestMapping(value = "/{id}", method = RequestMethod.GET)

public ResponseEntity<?> getById(@PathVariable String id){

    Article article=articleRepository.findById();

    if (article==null)
        return new ResponseEntity<String>("Article not found",HttpStatus.NOT_FOUND);

    else
        return new ResponseEntity<Article>(article, HttpStatus.OK);

}
```

# Tratarea exceptiilor

Soluția 2

```
@RequestMapping(value = "/{id}", method = RequestMethod.GET)
public ResponseEntity<Article> getById(@PathVariable String id){
    Article article=articleRepository.findById(id);
    if (article==null)
        throw new RepositoryException("Article not found.");
    ...
}

//2.a - in aceeași clasa cu metoda getById
@ExceptionHandler(RepositoryException.class)
@ResponseStatus(HttpStatus.NOT_FOUND)
public String repoException(RepositoryException e) { return e.getMessage(); }

//2.b

@ResponseStatus(HttpStatus.NOT_FOUND)
class RepositoryException extends RuntimeException {
    public RepositoryException(String message) {
        super(message);
    }
}
```

# Pornirea aplicatiei

```
package conference;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class StartApplication {

    public static void main(String[] args) {
        SpringApplication.run(StartApplication.class, args);
    }
}

package conference;
@RestController
public class ArticleController{
    @Autowired private ArticleRepository articleRepository;
    // ...
}

@Component
public class ArticleRepositoryJdbc implements ArticleRepository{ ...}
```

# Pornirea aplicatiei

- Adnotarea `@SpringBootApplication` specifică containerului Spring următoarele:
  - **Adaugă adnotarea `@Configuration` clasei curente** (specifică că poate conține definițiile unor bean-uri).
  - **Adaugă adnotarea `@EnableAutoConfiguration`**: specifică beanului Spring Boot să adauge beanuri în funcție de setările classpath-ului, a altor beanuri sau a altor proprietăți.
  - **Adaugă adnotarea `@EnableWebMvc`** necesară pentru a preciza că este o aplicație SpringMVC.
  - **Adaugă adnotarea `@ComponentScan` pentru căutarea altor bean-uri pornind de la pachetul “**conference**”** (permite identificarea controllerelor REST). (“**conference**” reprezintă pachetul în care este definită clasă adnotată cu `@SpringBootApplication`)
- Metoda `main()` folosește `SpringApplication.run()` din Spring Boot pentru lansarea aplicației.
- **Serverul web (Tomcat) va fi inclus și pornit automat.**

# Configurări Gradle

```
plugins {
    id 'org.springframework.boot' version '2.6.3'
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'
    id 'java'
}

group 'Conference'
version '1.0'

repositories {
    mavenCentral()
}

dependencies {
    implementation project (':ConferenceModel')
    implementation project (':ConferencePersistence')

    implementation group: 'com.fasterxml.jackson.core', name: 'jackson-annotations', version: '2.13.1'

    implementation 'org.springframework.boot:spring-boot-starter-actuator'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    testImplementation('org.springframework.boot:spring-boot-starter-test')
    testImplementation('com.jayway.jsonpath:json-path')
}
```

# Exemple

- ComputerShop
- Chat

# Clienti REST

- API pentru apelarea unui serviciu REST:
  - Java
  - .NET
  - JavaScript
  - etc.
- Browser web
- Extensii ale browserelor web:
  - RESTED
  - POSTMAN
  - Advanced REST Client
  - etc.

# Spring REST Client

- RestTemplate definește 36 de metode pentru interacțiunea cu resurse REST.

Method	Description
delete()	Performs an HTTP DELETE request on a resource at a specified URL
exchange()	Executes a specified HTTP method against a URL, returning a <code>ResponseEntity</code> containing an object mapped from the response body
execute()	Executes a specified HTTP method against a URL, returning an object mapped from the response body
getForEntity()	Sends an HTTP GET request, returning a <code>ResponseEntity</code> containing an object mapped from the response body
getForObject()	Sends an HTTP GET request, returning an object mapped from a response body
headForHeaders()	Sends an HTTP HEAD request, returning the HTTP headers for the specified resource URL
optionsForAllow()	Sends an HTTP OPTIONS request, returning the <code>Allow</code> header for the specified URL
postForEntity()	POSTs data to a URL, returning a <code>ResponseEntity</code> containing an object mapped from the response body
postForLocation()	POSTs data to a URL, returning the URL of the newly created resource
postForObject()	POSTs data to a URL, returning an object mapped from the response body
put()	PUTs resource data to the specified URL

# Spring REST Client

```
public class StartRestClient {  
  
    public static void main(String[] args) {  
        RestTemplate restTemplate=new RestTemplate();  
  
        //Adding an article  
        Article article=new Article("AB CB","AI","Genetic");  
        try{  
            String articleId= restTemplate.postForObject("http://localhost:8080/  
conference/articles",article, String.class);  
  
            // Updating article ...  
            article.setTitle("New title");  
            restTemplate.put("http://localhost:8080/conference/articles/"+articleId,  
article);  
  
        }catch(RestClientException ex){  
            System.out.println("Exception ... "+ex.getMessage());  
        }  
    }  
}
```

# .NET REST Client

- Clasa **System.Net.Http.HttpClient**
  - Trebuie adăugat (folosind NuGet) pachetul **Microsoft.AspNet.WebApi.Client**
  - ***HttpClient ar trebui instantiat o singură dată într-o aplicație*** și refolosit pe parcursul rulării aplicației. Crearea unei noi instanțe pentru fiecare cerere, poate genera erori de tip `SocketException` (se atinge numărul maxim de conexiuni permise).

```
class MainClass{
    static HttpClient client = new HttpClient();
    public static void Main(string[] args){
        client.DefaultRequestHeaders.Accept.Clear();
        client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));
        // Get an article ...
        Article result = await GetArticleAsync("http://localhost:8080/conference/articles/art124");
    }
    static async Task<Article> GetArticleAsync(string path){
        Article article = null;
        HttpResponseMessage response = await client.GetAsync(path);
        if (response.IsSuccessStatusCode)
        {
            article = await response.Content.ReadAsAsync<Article>();
        }
        return article;
    }
}
```

# Referințe

- <https://docs.microsoft.com/en-us/aspnet/web-api/overview/advanced/calling-a-web-api-from-a-net-client>
- Building REST services with Spring:  
<https://spring.io/guides/tutorials/bookmarks/>
- <http://www.service-architecture.com/articles/web-services/>
- Craig Walls, Spring in Action, 6th Edition, Ed. Manning, 2022
- Alte tutoriale

# Medii de proiectare și programare

2022-2023

Curs 11

# Continut

- Servicii REST - Clienți web

- JavaScript:

- Promise

- Fetch

- React

# JavaScript - Evenimente

- JavaScript este **secvențial** (eng. *single threaded*), două părți de cod nu pot fi executate în același timp, trebuie să fie executate una după cealaltă.
- În browsere, JavaScript **împarte un fir de execuție cu alte sarcini**, care diferă de la browser la browser.
- JavaScript se comportă similar cu firul de execuție corespunzător interfețelor grafice: orice modificare a interfeței grafice duce la amânarea următoarele sarcini corespunzătoare ei.
- În JavaScript se pot folosi evenimente și funcții callback:

```
var img1 = document.querySelector('.img-1');
```

```
img1.addEventListener('load', function() {
  // imaginea a fost încărcată
});
```

```
img1.addEventListener('error', function() {
  // a apărut o situație excepțională
});
```

# JavaScript - Evenimente

- Este posibil ca un eveniment să apară înainte de a adăuga un listener. (Se poate folosi proprietatea "complete" pentru a trata situația):

```
var img1 = document.querySelector('.img-1');

function loaded() {
  // imaginea a fost încărcată

}

if (img1.complete) {
  loaded();
}
else {
  img1.addEventListener('load', loaded);

}

img1.addEventListener('error', function() {
  // a apărut o situație excepțională
});
```

- Această soluție nu tratează cazul apariției unei erori înaintea adăugării listenerului.
- Soluția devine complexă când dorim să tratăm cazul încărcării mai multor imagini.
- Evenimentele sunt utile pentru situații care apar de mai multe ori asupra aceluiași obiect (apăsarea unei taste, etc). În aceste cazuri nu ne interesează ce s-a întâmplat înaintea adăugării listenerului.

# JavaScript - Promise

- Pentru tratarea rezultatului unei operații asincrone (succes/eșec), este preferat următorul şablon:

```
img1.callThisIfLoadedOrWhenLoaded(function() {
    // încărcată
}).orIfFailedCallThis(function() {
    // eșec
});

// și...
whenAllTheseHaveLoaded([img1, img2]).callThis(function() {
    // toate imaginile au fost încărcate
}).orIfSomeFailedCallThis(function() {
    // încărcarea unei imagini sau a mai multor imagini a eșuat
});
```

- Conceptul de **promise** a fost introdus pentru astfel de situații.
- Promises sunt *asemănătoare cu listeneri*, exceptând:
  - Rezultatul execuției unui promise poate fi succes/eșec o singură dată. Nu poate fi semnalat succesul/eșecul de mai multe ori. Nu poate fi schimbat rezultatul succes -> eșec și invers.
  - Dacă rezultatul execuției este succes/eșec, dar funcția callback este adăugată ulterior, funcția va fi apelată cu rezultatul corect, chiar dacă rezultatul s-a obținut anterior.
- Este utilă obținerea rezultatului unei operații asincrone (succes/eșec), deoarece este mai important rezultatul obținut decât momentul de timp în care a fost obținut.

# JavaScript - Promise

- Un obiect **Promise** reprezintă **eventualul rezultat al unei operații asincrone**. Modalitatea principală de a interacționa cu un promise este de a adăuga funcții callback care vor fi apelate fie cu rezultatul execuției (succes), fie cu motivul eșecului.
- Un *Promise* reprezintă o valoare care poate fi disponibilă *acum* sau *în viitor* sau *niciodată*. Valoarea respectivă e posibil să nu fie cunoscută în momentul creării obiectului.
- Permite adăugarea handlerelor (funcțiile callback) pentru tratarea succesului/eșecului.
- Permite metodelor asincrone să returneze un rezultat asemănător cu metodele sincrone: în loc să returneze rezultatul imediat, metoda asincronă returnează un obiect Promise cu ajutorul căreia poate fi obținut rezultatul execuției.
- Concepte:
  - **Promise**: un obiect cu o metodă *then*, a cărui execuție este conformă cu specificația.
  - **Thenable**: un obiect care definește o metodă *then*.
  - **Valoare**: orice valoare permisă în JavaScript (inclusiv *undefined*, un *thenable* sau un *promise*)
  - **Excepție**: o valoare aruncată folosind *throw*.
  - **Motiv**: o valoare care indică motivul respingerii unei promises.

# JavaScript - Promise

```
new Promise( /* executor */ function(resolve, reject) { ... } );
```

- *executor*: O funcție cu doi parametri *resolve* și *reject* (funcții callback).
  - Funcția *executor* este executată imediat de implementarea Promise, transmițând funcțiile *resolve* și *reject* (Executorul este apelat înainte de ieșirea din constructorul corespunzător obiectului promise).
  - Funcțiile *resolve/reject* când sunt apelate acceptă/resping promise-ul.
  - De obicei, executorul inițiază cod asincron, iar după încheierea execuției apelează fie funcția *resolve(succes)* sau *reject(eșec, eroare)*.
- Dacă o eroare este aruncată în timpul execuției executorului, promise-ul este respins (*rejected*). Valoarea returnată de executor în acest caz este ignorată.

# JavaScript - Stările Promise

- Un obiect *Promise* poate fi în una din următoarele **stări**:
  - **pending**: stare initială, încă nu a fost îndeplinit sau respins (operația asincronă încă se execuță).
  - **fulfilled**: execuția operației asincrone s-a încheiat cu succes (îndeplinit).
  - **rejected**: execuția operației asincrone a eșuat (eroare) (respins).
- Un promise în starea *pending* poate fi îndeplinit cu o anumită valoare, sau poate fi respins pe baza unui motiv (eroarea).
- În oricare dintre cele două situații, handlerele asociate folosind metoda *then* sunt apelate.
- Dacă un promise a fost deja îndeplinit sau respins în momentul atașării unui handler, handler-ul va fi apelat cu rezultatul execuției promise-ului.
- Un promise este **stabilit** (eng. *settled*) dacă a fost fie îndeplinit, fie respins, dar nu este în starea de pending.
- Se mai folosește termenul **rezolvat** (eng. *resolved*) - înseamnă că obiectul promise este stabilit, sau este într-o înlanțuire de obiecte promise.

# JavaScript - Promise

- Crearea unui obiect Promise:

```
var promise = new Promise(function(resolve, reject) {  
    // codul (asincron)  
  
    if /* totul s-a executat cu succes */ {  
        resolve("Succes!");  
    }  
    else {  
        reject(Error("Eroare"));  
    }  
});
```

- Constructorul primește un singur parametru, un executor. La finalul execuției codului (asincron) se apelează fie funcția *resolve*, fie funcția *reject*.
- Se recomandă respingerea unui promise folosind un obiect de tip **Error** (posibilitatea obținerii stivei de execuție, depanare mai ușoară).

# JavaScript - Promises

- Folosirea unui promise:

```
promise.then(function(result) {  
    console.log(result); // "Succes!"  
, function(err) {  
    console.log(err); // Error: "Eroare"  
});
```

- Funcția *then()* primește doi parametri, un callback pentru execuția cu succes, și un callback pentru eșec. Ambele funcții sunt opționale, se poate adăuga doar un callback pentru succes sau pentru eșec.
- JavaScript promises pot fi executate și în afara browserelor (ex. folosind NodeJS).
- DOM folosește promises. Funcțiile asincrone din noul DOM APIs folosesc promises (ex. ServiceWorker, Streams, etc.).

# Înlătuirea Promise

- Obiectele promise pot fi înlătuite, pentru a transforma valorile obținute sau pentru a executa asincron alt cod, unul după altul.

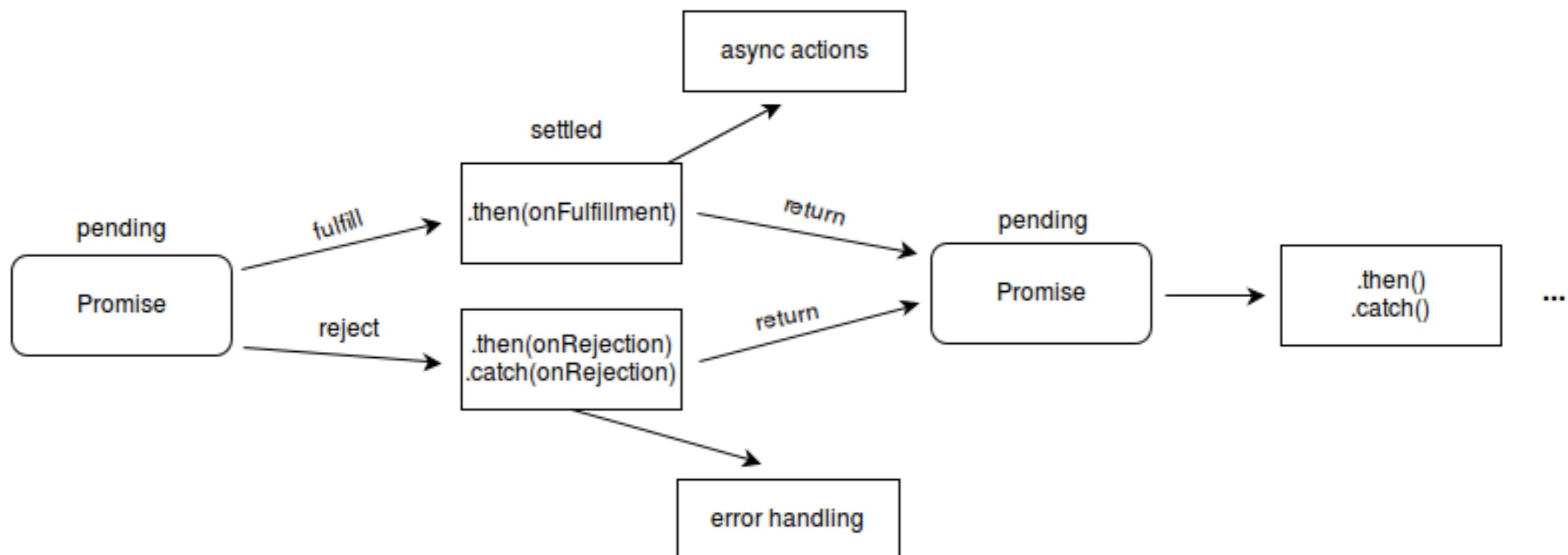
```
var promise = new Promise(function(resolve, reject) {  
    resolve(1);  
});
```

```
promise.then(function(val) {  
    console.log(val); // 1  
    return val + 2;  
}).then(function(val) {  
    console.log(val); // 3  
})
```

```
// —
```

```
get('url/story.json').then(function(response) {  
    console.log("Success!", response); //response e in format text  
})
```

# Înlăntuire Promises (*chaining*)



# JavaScript - Promises chaining

```
get('url/story.json').then(function(response) {  
    return JSON.parse(response);  
}).then(function(response) {  
    console.log("Format JSON!", response);  
})
```

- *JSON.parse()* primește un singur parametru și returnează o valoare transformată, se poate folosi și formatul:

```
get('url/story.json').then(JSON.parse).then(function(response) {  
    console.log("Format JSON:", response);  
})
```

```
functiongetJSON(url) {  
    return get(url).then(JSON.parse);  
}
```

- Funcția *getJSON()* returnează un alt promise, care conține obiectul JSON obținut după parsarea răspunsului.

# Promises - Înlăntuirea acțiunilor asincrone

- Înlăntuirea folosind funcția *then* poate fi folosită și pentru a executa secvențial mai multe operații asincrone (contează ordinea executării lor).
- Dacă metoda *then()* returnează o valoare, următorul apel al funcției *then* primește valoarea respectivă ca și parametru.
- Dacă returnează un alt obiect promise, următorul apel al funcției *then* așteaptă terminarea execuției acestuia, și va fi apelat când obiectul devine *settled* (succes/eșec).

```
getJSON('url/story.json').then(function(story) {  
  return getJSON(story.chapterUrls[0]);  
}).then(function(chapter1) {  
  console.log("Got chapter 1!", chapter1);  
})
```

# Promises - Tratarea excepțiilor

- Funcția `then()` primește doi parametri: callback succes și callback eșec:

```
get('url/story.json').then(function(response) {  
  console.log("Success!", response);  
, function(error) {  
  console.log("Failed!", error);  
})
```

- Se poate folosi și funcția `catch()`:

```
get('url/story.json').then(function(response) {  
  console.log("Success!", response);  
}).catch(function(error) {  
  console.log("Failed!", error);  
})
```

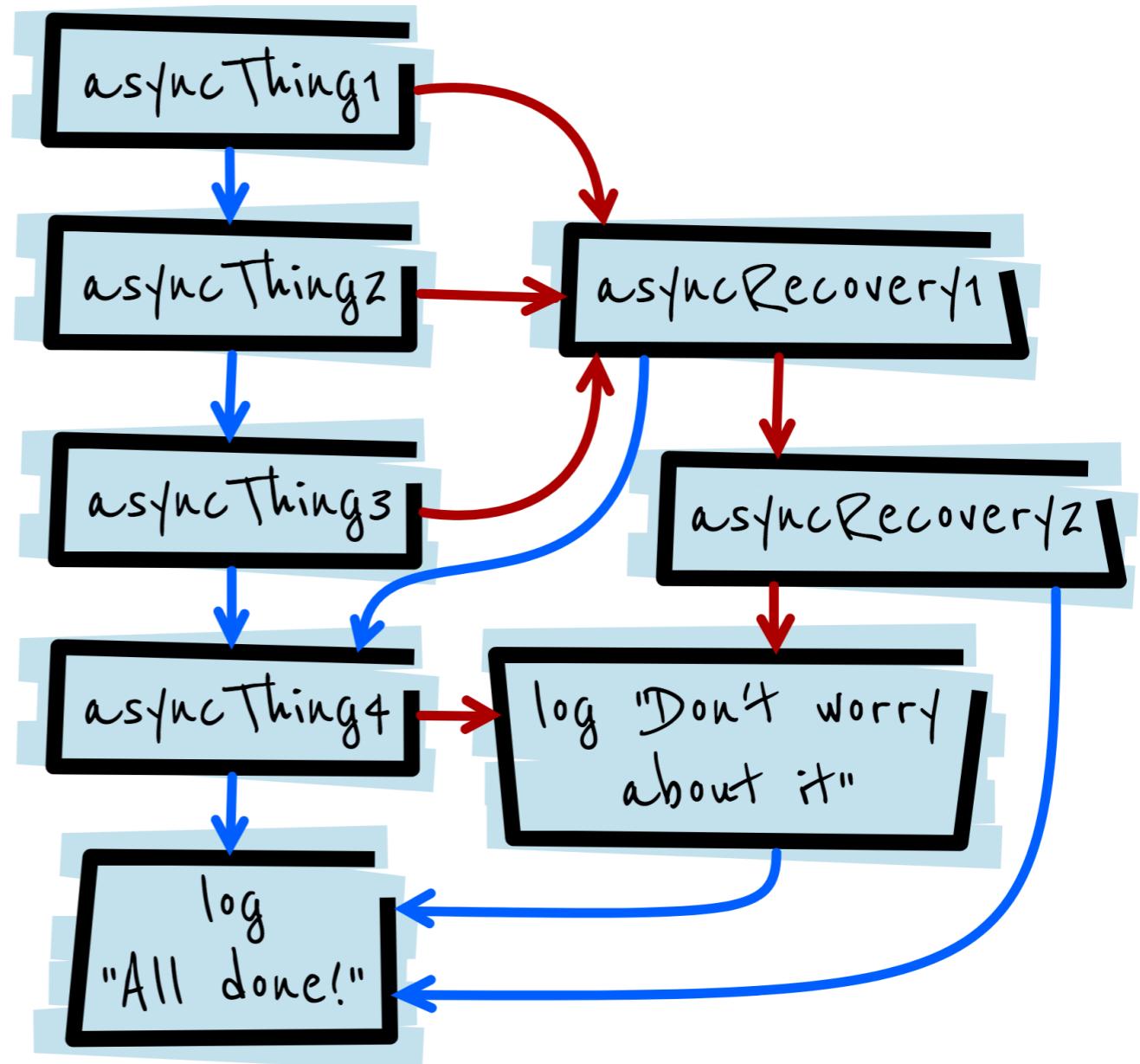
- `Catch()` este sinonim cu `then(undefined, func)`, dar este mai ușor de înțeles.
- Cele două exemple de cod, nu sunt echivalente (ultimul este echivalent cu):

```
get('url/story.json').then(function(response) {  
  console.log("Success!", response);  
}).then(undefined, function(error) {  
  console.log("Failed!", error);  
})
```

# Promises - Tratarea excepțiilor

- Dacă un promise este respins (rejected), execuția se mută la următorul apel *then* care conține un callback pentru cazul respingerii, sau până la întâlnirea unui apel *catch*.
- În situația *then(func1, func2)*, se apelează *func1* sau *func2*, dar nu amândouă.
- În situația *then(func1).catch(func2)*, amândouă funcțiile vor fi apelate dacă *func1* este respinsă (fac parte din obiecte promise diferite)

```
asyncThing1().then(function() {
  return asyncThing2();
}).then(function() {
  return asyncThing3();
}).catch(function(err) {
  return asyncRecovery1();
}).then(function() {
  return asyncThing4();
}, function(err) {
  return asyncRecovery2();
}).catch(function(err) {
  console.log("Don't worry about it");
}).then(function() {
  console.log("All done!");
})
```



# JavaScript excepții și promises

- Respingerea are loc când un obiect promise este respins explicit (prin apelul callbackului *reject*), dar și când o eroare este aruncată în executorul obiectului promise:

```
var jsonPromise = new Promise(function(resolve, reject) {  
    // JSON.parse aruncă eroare dacă stringul nu este în format json  
    // JSON invalid, obiectul promise este respins (rejected) implicit:  
    resolve(JSON.parse("Nu e format JSON"));  
});
```

```
jsonPromise.then(function(data) {  
    // NU va fi apelat niciodată:  
    console.log("Succes!", data);  
}).catch(function(err) {  
    // Va fi apelat:  
    console.log("Esec!", err);  
})
```

- Se recomandă execuția codului asincron în executor, astfel erorile vor fi prinse și vor genera automat respingerea.

# JavaScript exceptii și promises

- Aceeași situație pentru erorile aruncate în funcțiile callback transmise funcției then():

```
get('/').then(JSON.parse)
  .then(function() {
    // Eroare, '/' este o pagina HTML, nu este in format JSON
    // JSON.parse arunca exceptie
    console.log("Succes!", data);
  }).catch(function(err) {
    // Se va executa:
    console.log("Esec!", err);
  })
```

# JavaScript Promises

- Funcția `Promise.resolve(val)`, creează un promise care se va îndeplini cu valoarea transmisă ca și parametru.
  - Dacă parametrul este un alt promise, acesta va fi returnat.
  - Exemplu: `Promise.resolve('Ana')` creează un obiect promise, care va fi îndeplinit cu valoarea 'Ana'.
  - Dacă este apelat fără parametri, va fi îndeplinit cu valoarea "`undefined`".
- Funcția `Promise.reject(val)`, creează un promise care va fi respins cu valoarea transmisă ca și parametru (sau `undefined`).
- Funcția `Promise.all` primește un tablou de obiecte promise și creează un obiect promise care va fi îndeplinit dacă toate sunt îndeplinite. Valoarea este un tablou cu rezultatele obținute la îndeplinirea obiectelor promise, în ordinea acestora din tabloul inițial.

```
Promise.all(arrayOfPromises).then(function(arrayOfResults) {  
  //...  
})
```

# JavaScript Promises

## Constructor

```
new Promise(function(  
  resolve, reject) {});
```

**resolve(thenable)**

Your promise will be fulfilled/rejected with the outcome of **thenable**

**resolve(obj)**

Your promise is fulfilled with **obj**

**reject(obj)**

Your promise is rejected with **obj**. For consistency and debugging (e.g., stack traces), obj should be an **instanceof Error**. Any errors thrown in the constructor callback will be implicitly passed to **reject()**.

## Instance Methods

```
promise.then(  
  onFulfilled,  
  onRejected)
```

**onFulfilled** is called when/if "promise" resolves. **onRejected** is called when/if "promise" rejects. Both are optional, if either/both are omitted the next **onFulfilled/onRejected** in the chain is called. Both callbacks have a single parameter, the fulfillment value or rejection reason. **then()** returns a new promise equivalent to the value you return from **onFulfilled/onRejected** after being passed through **Promise.resolve**. If an error is thrown in the callback, the returned promise rejects with that error.

```
promise.catch(  
  onRejected)
```

Sugar for **promise.then(undefined, onRejected)**

# JavaScript Promises

- Metode statice

## Method summaries

<b>Promise.resolve( promise);</b>	Returns promise (only if <code>promise.constructor == Promise</code> )
<b>Promise.resolve( thenable);</b>	Make a new promise from the thenable. A thenable is promise-like in as far as it has a `then()` method.
<b>Promise.resolve(obj);</b>	Make a promise that fulfills to <code>obj</code> . in this situation.
<b>Promise.reject(obj);</b>	Make a promise that rejects to <code>obj</code> . For consistency and debugging (e.g. stack traces), <code>obj</code> should be an <code>instanceof Error</code> .
<b>Promise.all(array);</b>	Make a promise that fulfills when every item in the array fulfills, and rejects if (and when) any item rejects. Each array item is passed to <b>Promise.resolve</b> , so the array can be a mixture of promise-like objects and other objects. The fulfillment value is an array (in order) of fulfillment values. The rejection value is the first rejection value.
<b>Promise.race(array);</b>	Make a Promise that fulfills as soon as any item fulfills, or rejects as soon as any item rejects, whichever happens first.

# JavaScript Promises async/await

- **async** definește o funcție asincronă. Rezultatul funcției va fi implicit un Promise.

```
async function name([param[, param[, ... param]]]) {  
    statements  
}
```

- Operatorul **await** așteaptă rezultatul execuției unui Promise. **Poate fi folosit doar în interiorul unei funcții asincrone.**

```
[value] = await expression;
```

- *expression* poate fi un Promise sau orice valoare
- Dacă *expression* nu este un Promise, ea este convertită folosind `Promise.resolve(expression)`
- *value* va primi rezultatul execuției cu succes a Promise-ului sau valoarea expresiei (dacă expresia nu este un Promise)
- Dacă Promise-ul este respins, expresia *await* aruncă excepție cu valoarea respingerii.

# JavaScript Promises async/await

```
function resolveAfter2Seconds(x) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(x);
    }, 2000);
  });
}

async function f1() {
  var x = await resolveAfter2Seconds(10);
  console.log(x); // 10
}

f1();
```

# JavaScript -Fetch

- Funcția `fetch()` permite executarea unor apeluri prin rețea asemănătoare cu `XMLHttpRequest` (XHR).
- Diferență: Fetch API folosește obiecte promise, care permit scrierea unui cod mai simplu și mai clar, evitând callbackurile complexe și API complex al `XMLHttpRequest`.
- Exemplu `XMLHttpRequest`: cererea unui URL, obținerea unui răspuns și parsarea lui ca și JSON.

```
function reqListener() {  
  var data = JSON.parse(this.responseText);  
  console.log(data);  
}  
  
function reqError(err) {  
  console.log('Eroare:', err);  
}  
  
var oReq = new XMLHttpRequest();  
oReq.onload = reqListener;  
oReq.onerror = reqError;  
oReq.open('get', './api/some.json', true);  
oReq.send();
```

# JavaScript -Fetch

- Solutia fetch()

```
fetch('./api/some.json')
  .then(
    function(response) {
      if (response.status !== 200) {
        console.log('Eroare. Status Code: ' + response.status);
        return;
      }

      // Examinarea textului din răspuns
      response.json().then(function(data) {
        console.log(data);
      });
    }
  )
  .catch(function(err) {
    console.log('Eroare Fetch ', err);
  });
}
```

- Rezultatul unui apel *fetch()* este un obiect de tip Stream. Se poate apela metoda *json()*, iar rezultatul va fi un obiect de tip Promise, deoarece citirea streamului se va face asincron.

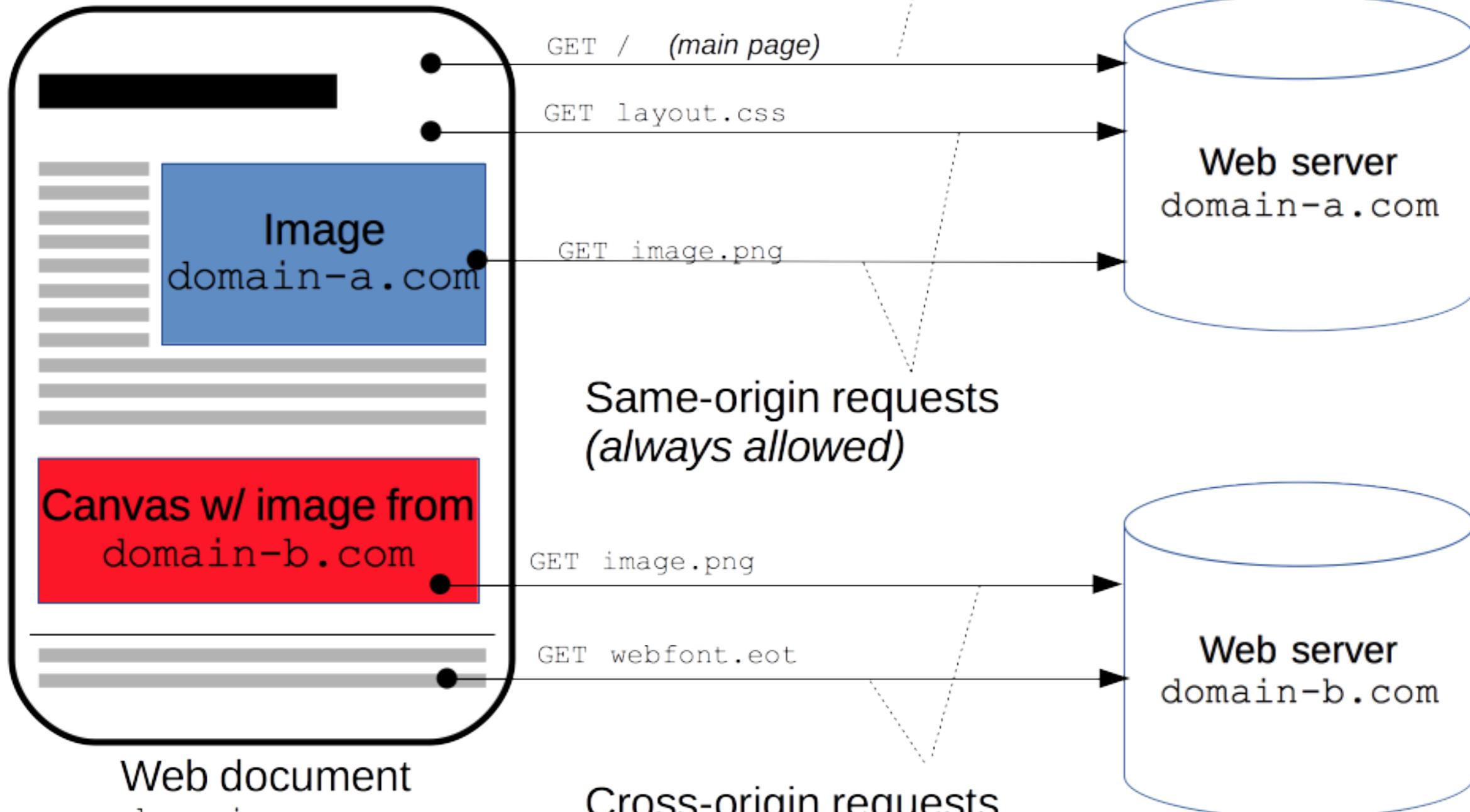
# Fetch Response

- Apelul funcției *fetch* returnează un obiect de tip **response**, dacă obiectul de tip promise asociat este îndeplinit.
- Proprietăți utile:
  - **Response.status** - status code asociat răspunsului (întreg, implicit 200).
  - **Response.statusText** - textul asociat status code-ului respectiv (string, implicit "OK").
  - **Response.ok** - permite verificarea rapida daca status-ul este între 200-299 (boolean).
- Alte informații care pot fi obținute (ex. antetele):

```
fetch('users.json').then(function(response) {  
  console.log(response.headers.get('Content-Type'));  
  console.log(response.headers.get('Date'));  
  
  console.log(response.status);  
  console.log(response.statusText);  
  console.log(response.type);  
  console.log(response.url);  
});
```

# Cross Origin Resource Sharing (CORS)

Main request: defines origin.



# Fetch Response Types

- Când se face o cerere fetch, răspunsul va primi un tip (*response.type*) de tip "basic", "cors" sau "opaque". Aceste tipuri indică de unde provine resursa și poate fi folosit pentru a decide modalitatea de tratare a obiectului de tip response.
  - **basic**: se face o cerere pentru o resursă având aceeași origine ca și cererea. Nu există restricții asupra informațiilor ce pot fi obținute din răspuns.
  - **cors**: se face o cerere pentru o resursă a cărei origine este diferită și care returnează **Cross Origin Resource Sharing (CORS)** în antet. Un răspuns de tip *cors* restricționează antetele care pot fi accesate la `Cache-Control`, `Content-Language`, `Content-Type`, `Expires`, `Last-Modified`, și `Pragma`.
  - **opaque**: se face o cerere pentru o resursă având altă origine și care nu returnează CORS în antet. **La acest tip de răspuns nu se pot obține datele returnate sau status-ul răspunsului.**
    - Nu se poate verifica dacă cererea s-a efectuat cu succes.

# Fetch Response Types

- Se poate defini un mod pentru o cerere fetch, astfel încât doar anumite cereri vor fi apelate:
  - *same-origin*: se execută cu succes doar pentru cereri având aceeași origine, alte cereri vor fi respinse.
  - *cors*: permite cereri având aceeași origine, sau către alte origini care conțin antetele CORS corespunzătoare.
  - *cors-with-forced-preflight*: se verifică anterior că cererea poate fi făcută.
  - *no-cors*: pentru cereri către alte origini care nu au setat antetul CORS, răspunsul va fi de tip opaque.
- Pentru definirea modului se adaugă un parametru cererii fetch:

```
fetch('http://some-site.com/cors-enabled/some.json', {mode: 'cors'})  
  .then(function(response) {  
    return response.text();  
  })  
  .then(function(text) {  
    console.log('Request successful', text);  
  })  
  .catch(function(error) {  
    log('Request failed', error)  
  });
```

# Fetch - Înlătuirea promise

- Pași cerere HTTP:
  - verificarea statusului răspunsului
  - parsarea conținutului pentru a obține obiectul JSON.
- Soluție care necesită doar folosirea informației obținute.

```
function status(response) {  
  if (response.status >= 200 && response.status < 300) {  
    return Promise.resolve(response)  
  } else {  
    return Promise.reject(new Error(response.statusText))  
  }  
}  
  
function json(response) {  
  return response.json()  
}  
  
fetch('api/users.json')  
  .then(status)  
  .then(json)  
  .then(function(data) {  
    console.log('Success - raspuns JSON:', data);  
  }).catch(function(error) {  
    console.log('Cerere esuata', error);  
});
```

# Fetch - Verificarea răspunsului

- Un obiect promise *fetch()* va respinge cu un obiect de tip `TypeError`, când apare o eroare de rețea.
- Un apel al funcției *fetch* ar trebui să verifice execuția cu succes a acesteia, verificând că obiectul de tip promise a fost îndeplinit, iar apoi verificând valoarea proprietății `Response.ok`.
- Exemplu:

```
fetch('api/some.json').then(function(response) {  
  if(response.ok) {  
    return response.json();  
  }  
  throw new Error('Raspunsul nu a fost ok.');
```

`}).then(function(data) {  
 //folosirea informatiei`

```
).catch(function(error) {  
  console.log('Eroare cu functia fetch: ' + error.message);  
});
```

# Fetch - Request

- Funcția `fetch()` poate fi apelată și folosind un parametru de tip Request.
- `Request()` primește aceeași parametrii ca și funcția `fetch()`.

```
var myHeaders = new Headers();
```

```
var myInit = { method: 'GET',
               headers: myHeaders,
               mode: 'cors',
               cache: 'default' };
```

```
var myRequest = new Request('api/some.json', myInit);
```

```
fetch(myRequest).then(function(response) {
  return response.json();
}).then(function(myData) {
  console.log(myData);
});
```

- Observație: **Parametrii de tip query trebuie adăugați explicit la cerere (url, antet).**  
Nu există metode speciale care permit transmiterea acestora.

# Antete Fetch

- *Headers* permite crearea/păstrarea/accesarea antetelor unei cereri/unui răspuns.
- Este un dicționar de perechi (nume-antet, valoare-antet):

```
var content = "Hello World";
var myHeaders = new Headers();
myHeaders.append("Content-Type", "text/plain");
myHeaders.append("Content-Length", content.length.toString());
myHeaders.append("X-Custom-Header", "ProcessThisImmediately");
```

- Se poate transmite și un tablou conținând valorile antetelor:

```
myHeaders = new Headers({
  "Content-Type": "text/plain",
  "Content-Length": content.length.toString(),
  "X-Custom-Header": "ProcessThisImmediately",
});
console.log(myHeaders.has("Content-Type")); // true
console.log(myHeaders.has("Set-Cookie")); // false
myHeaders.set("Content-Type", "text/html");
myHeaders.append("X-Custom-Header", "AnotherValue");

console.log(myHeaders.get("Content-Length")); // 11
console.log(myHeaders.get("X-Custom-Header")); // ["ProcessThisImmediately", "AnotherValue"]

myHeaders.delete("X-Custom-Header");
console.log(myHeaders.get("X-Custom-Header")); // []
```

# Fetch - Body

- Cererile și răspunsurile pot conține informație/date.
- Informația poate fi de următoarele tipuri:
  - ArrayBuffer
  - ArrayBufferView
  - Blob/File
  - string
  - URLSearchParams
  - FormData
- Sunt definite metode pentru obținerea informației dintr-un răspuns. Toate metodele returnează un obiect de tip promise, care conțin informația.
  - arrayBuffer()
  - blob()
  - json()
  - text()
  - formData()

# Fetch - Feature detection

- Suportul pentru Fetch API poate fi detectat prin verificarea existenței (în fereastră sau scopul workerului) a obiectelor:
  - Headers
  - Request
  - Response
  - funcția fetch().

Exemplu

```
if (self.fetch) {  
    // execuția cererii fetch  
} else {  
    // folosirea XMLHttpRequest?  
}
```

# React

- React este o bibliotecă JavaScript declarativă, flexibilă și eficientă care permite construirea ușoară și rapidă a interfețelor cu utilizatorul.
- Permite dezvoltatorilor să creeze aplicații web mari, care folosesc date ce se pot modifica în timp, fără a reîncărca toată pagina.
- Obiectivele principale: rapiditate, simplitate, scalabilitate.
- React procesează doar interfețe grafice în aplicații.
  - Corespunde View-ului din şablonul Model-View-Controller (MVC) și poate fi folosit împreună cu alte biblioteci sau frameworkuri JavaScript din MVC (ex. Angular).
- Este dezvoltat și întreținut de o comunitate formată din dezvoltatori de la Facebook, Instagram și dezvoltatori individuali.
- Folosit pentru site-uri precum Netflix, Airbnb, Walmart, etc.
- Mai multe versiuni disponibile
  - React 18 with hooks (latest)

# React - Elemente

- Elementele React sunt cele mai mici construcții ale unei aplicații React. Un element descrie ceea ce trebuie afișat pe ecran.
- Elementele React sunt obiecte simple și ușor de creat. React DOM se ocupă de actualizarea DOM pentru potrivirea cu elementele React.

```
const element = <h1>Hello, world</h1>;
```

- Este numit un nod DOM, pentru că tot ce este conținut de el va fi gestionat de React DOM.
- Aplicațiile dezvoltate folosind React conțin de obicei un singur nod root.
- Pentru redarea unui element React într-un nod root:

```
ReactDOM.createRoot(document.getElementById('root')).render(  
  <React.StrictMode>  
    <h1>Hello, world</h1>  
  </React.StrictMode>  
)
```

- Actualizarea unui element: **elementele React sunt immutable**. Un element odată creat, datele sale nu se mai pot modifica.
- Majoritatea aplicațiilor React apelează *render()* o singură dată.
- Biblioteca React actualizează la afișare doar datele modificate. React DOM compară un element și descendenții săi cu datele anterioare și invocă doar actualizările necesare pentru ca DOM să aibă starea dorită.

# JSX

```
//nu este nici un string, nici HTML.  
const element = <h1>Hello, world!</h1>;
```

- Sintaxa este numită **JSX**, și este o extensie a sintaxei JavaScript.
- Mai strictă decât HTML (<br />).
- Este modul recomandat de descriere în React a interfețelor grafice.
- JSX produce elemente React.
- O componentă poate returna doar un singur element JSX.
- Dacă trebuie returnate mai multe se recomandă folosirea <div>...</div> sau <> ... </>
- Se pot include orice expresii JavaScript în JSX, prin includerea lor între acolade ('{}')

```
const user = {  
  firstName: 'Popescu',  
  lastName: 'Ana'  
};  
  
function formatName(user) {  
  return user.firstName + ' ' + user.lastName;  
}  
  
const element = (  
  <h1>  
    Hello, {formatName(user)}!  
  </h1>  
);  
ReactDOM.createRoot(document.getElementById('root')).render(  
  <React.StrictMode>  
    element  
  </React.StrictMode>  
,
```

# Specificarea atributelor folosind JSX

- Se pot folosi ghilimele pentru specificarea atributelor de tip string:

```
const element = <div tabIndex="0"></div>;
```

- Se folosesc accoladele pentru includerea unei expresii JavaScript ca și valoarea a unui atribut:

```
const element = <img src={user.avatarUrl}></img>;
```

- **Nu se pun ghilimele când se includ expresii JavaScript** (expresiile vor fi considerate stringuri)

```
const element = </img>; //!!!
```

- Dacă un tag este gol, se închide cu `>`, asemănător XML:

```
const element = <img src={user.avatarUrl} />;
```

- Tagurile JSX pot conține fii:

```
const element = (  
  <div>  
    <h1>Salut</h1>  
    <h2>Bine ai venit!</h2>  
  </div>  
)
```

# Componente React

- React are la bază definirea unor **componente React**.
- O componentă poate păstra mai multe instanțe a altelor componente (relația *părinte-copil*).
- Componentele permit împărțirea UI în părți independente și reutilizabile, și dezvoltarea independentă a acestora.
- Definirea unei componente (React 18):

```
function Greeting(props){  
  return (<h1>Hello, {props.name}</h1>);  
 // sau return (<>Hello, {props.name}</>);  
  
 //gresit  
 //return (Hello, {props.name});  
}
```

- Rezultatul returnat de funcție specifică modul de afișare a componentei.
- **Numele componentelor încep întotdeauna cu litera mare.**
- Exemplu, `<div />` reprezintă un tag DOM, dar `<Greeting />` reprezintă o componentă și este necesar ca *Greeting* să fie disponibil în scop.
- Datele afișate de o componentă React sunt obținute prin proprietăți (*props*) sau din starea componentei (*state*).

# Componente și Props

- Componentele React au două tipuri de date: **starea** și **proprietățile**.
- Când React întâlnește un element reprezentând o componentă definită de dezvoltator, îi transmite atributele JSX ca și un singur obiect (adesea numit “props”), prin parametrii funcției.
- **props** - conține proprietățile definite de componenta care a apelat această componentă.
- **Proprietățile sunt *read-only***, o componentă nu trebuie să-și modifice propriile proprietăți:

```
ReactDOM.createRoot(document.getElementById('root')).render(  
  <React.StrictMode>  
    <Greeting name="Ana"/> //folosirea componentei  
  </React.StrictMode>  
)
```

```
function Greeting(props){  
  return (<> {props.name} </>);  
}
```

```
function Greeting({name}){  
  return (<> {name} </>);  
}
```

```
<button onClick={handleClick}> //fara paranteze când se transmite o funcție!  
  Say hello  
</button>
```

# Componența componentelor

- Componentele pot referi alte componente.
- Permite folosirea același nivel de abstractizare a componentelor: un buton, un form, un dialog, etc sunt toate exprimate ca și componente.
- Exemplu -crearea unei componente care afișează componenta *Greeting* de mai multe ori:

```
function App(){
  return (
    <div>
      <Greeting name="Ana" />
      <Greeting name="Maria" />
      <Greeting name="Ion" />
    </div> );
}

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

# React Component -States

- **Starea** (eng. state) unei componente este informația pe care componenta se așteaptă să o gestioneze singură.
- Contine date specifice componentei, care se pot modifica în timp.
- **Starea este definită de dezvoltator.** Dacă informația nu este folosită în **return**, atunci nu face parte din starea componentei.

```
import { useState } from 'react';
function Counter(){
  const [count, setCount] = useState(0); //count -variabila de stare, 0-valoarea initiala, setCount-functie
  function handleClick(){
    setCount(count+1);
  }
  return (<button onClick={handleClick}> Apasat de {count} ori</button>);
}
```

- Variabila `count` va păstra o parte din starea componentei (pot fi mai multe variabile)
- *Variabilele de stare trebuie considerate immutable*, și nu trebuie modificate în mod direct.
- Orice actualizare a unei variabile de stare a componentei (ex. `count`) se face folosind funcția asociată (ex. `setCount`). De fiecare dată când `setCount()` este apelat, React actualizează starea, determină diferențele dintre starea anterioară și noua stare și injectează o mulțime de modificări DOM-ului corespunzător paginii. În acest fel actualizările UI sunt rapide și eficiente.

# React Component -States

- Observații legate de starea unei componente:

- **NU se modifică starea direct.**

```
const [count, setCount] = useState(0);
count=count+1; //eroare
```

Orice modificare se face prin funcția asociată variabilei (ex. setCount)

- **O componentă poate avea mai multe variabile de stare, fiecare având funcția ei.**

```
function UserForm(props){

  const [name, setName] = useState('');
  const [username, setUsername] = useState('');
  const [passwd, setPassword] = useState('');

  //...
}
```

- Batch processing pentru stare și actualizarea interfeței

```
function handleClick(){
  setName(newName);
  setUsername(newUsername)
  setPassword(newPassword)
}
//interfața grafică se va actualiza o singura dată
```

# React Component -Starea

- Datele sunt transmise de la componenta părinte la descendenți.
  - Nici părinții, nici descendenții nu știu dacă o anumită componentă este *stateful* sau *stateless*.
  - Adesea, starea este numită *stare locală* sau *încapsulată*. Starea nu este disponibilă altor componente, doar componentei care a creat-o.
  - O componentă poate alege să transmită starea sa componentelor fii prin intermediul proprietăților:

```
function FormattedDate({date}){  
    return (<h2>Data este {date.getDate()}/{date.getMonth()+1}/{date.getFullYear()}</h2>);  
}
```

```
function Calendar(){  
    const [date, setDate]=useState(new Date());  
    return (<FormattedDate date={date}/>);  
}
```

- Acest mod de a transmite informația este numit flux de date "top-down" sau "unidirectional". Orice stare este totdeauna definită de o anumită componentă, și orice informație sau UI derivat din acea stare poate afecta doar componentele descendente din arbore.
- În aplicațiile React, faptul că o componentă este *stateful* sau *stateless* este considerat detaliu de implementare care se poate schimba în timp.
- Componente stateless pot conține componente stateful și invers.

# Tratarea evenimentelor

- Asemănătoare cu tratarea evenimentelor corespunzătoare elementelor DOM.
- **Diferențe:**

- Evenimentele React sunt numite folosind camelCase, nu doar litere mici.
- Cu JSX se poate transmite o funcție ca și un event handler.

HTML:

```
<button onclick="activateLasers()">  
  Activate Lasers  
</button>
```

React:

```
<button onClick={activateLasers}> //fara paranteze!  
  Activate Lasers  
</button>
```

- Nu se poate returna *false* pentru a preveni comportamentul implicit din React. Trebuie apelată explicit metoda *preventDefault*.

HTML: pentru a preveni comportamentul implicit de a deschide o nouă pagină:

```
<a href="#" onclick="console.log('The link was clicked.'); return false">  
  Click me  
</a>
```

React:

```
function ActionLink() {  
  function handleClick(e) {  
    e.preventDefault();  
    console.log('The link was clicked.');  }  
  
  return (  
    <a href="#" onClick={handleClick}>  
      Click me  
    </a>  
  );  
}
```

# Tratarea evenimentelor

- O altă modalitate de asociere este folosirea sintaxei arrow function:

```
function LoggingButton() {  
  handleClick = () => {  
    //...  
  }  
  //sau  handleClick = (e) => { console.log(e)}  
}
```

```
  return (  
    <button onClick={handleClick}>  
      Click me  
    </button>  
  );  
}
```

# Input Forms

```
function UserForm(props){  
  
    const [name, setName] = useState('');  
    const [username, setUsername] = useState('');  
    const [passwd, setPasswd] = useState('');  
  
    function handleSubmit (event){  
        let user={id:username,  
                  name:name,  
                  passwd:passwd  
        }  
        //alte operații  
        event.preventDefault();  
    }  
    return(  
        <form onSubmit={handleSubmit}>  
            <label>  
                Username:  
                <input type="text" value={username} onChange={e=>setUsername(e.target.value)} />  
            </label><br/>  
            <label>  
                Name:  
                <input type="text" value={name} onChange={e=>setName(e.target.value)} />  
            </label><br/>  
            <label>  
                Passwd:  
                <input type="password" value={passwd} onChange={e=>setPasswd(e.target.value)} />  
            </label><br/>  
  
            <input type="submit" value="Add user" />  
        </form>);  
    }  
}
```

# Crearea unei aplicații simple React

- **Instalare:** NodeJs: <https://nodejs.org/en/>
  - npm: package manager (inclus in nodejs)
- **Creare aplicatie:** `npm create vite@latest my-app -- -- template react`
- Numele directorului cu litere mici!
- Urmati instructiunile ...
  - `cd my-app`
- **Pornirea aplicăiei:**
  - `npm run dev` //va fi disponibilă la <http://localhost:5173>
  - `npm run build`
- Nu e necesară configurarea folosind babel, webpack/browserify, typescript, etc.

**Important:** NU uitati sa **setați variabila PATH pentru a include și calea către npm**

# Ciclul de viață - Versiuni anterioare

- **Versiuni anterioare** - fiecare componentă avea câteva metode care erau apelate în ciclul său de viață.
  - Aceste metode puteau fi redefinite pentru a particulariza componenta.
  - Metodele care începeau cu ‘will’ erau apelate înaintea apariției evenimentului, iar cele cu ‘did’ erau apelate după apariția evenimentului.
  - Metodele apelate când o componentă era creată și adăugată la DOM.
    - *constructor()*, *componentWillMount()*, *render()*, *componentDidMount()*
  - Actualizarea: cauzată de modificări ale proprietăților sau a stării componentei (metodele erau apelate când componenta era re-afișată.)
    - *componentWillReceiveProps()*, *shouldComponentUpdate()*,  
*componentWillUpdate()*, *render()*, *componentDidUpdate()*
  - Dezasamblarea: metoda era apelată când componentă era ștearsă din DOM:
    - *componentWillUnmount()*

# Ciclul de viață - Versiunea curentă

- metoda useEffect()

```
import { useState } from 'react';
import {useEffect} from 'react';

function UserApp(){
const [users, setUsers] = useState([{"passwd":"m","name":"Marinescu Maria","id":"maria"}]);

//...
useEffect(()=>
{
  //actualizarea interfeței ...
  GetUsers().then(users=>setUsers(users));
}, []);
}
```

Permite executarea unei secvențe de cod după afișarea unei componente și de fiecare dată când este actualizată

- **useState, useEffect - hooks (vers. React 16.8)**

Altele:

- useContext
- useId
- useDeferredValue
- useTransition
- useSyncExternalStore
- useInsertionEffect

# React - arrow functions

```
import React, {useState} from 'react';

const App = () => {
  const greeting = 'React functional component!';
  return <Headline title={greeting} />;
};

const Headline = (props) => {
  const [votes, setVotes] = useState(0);
  const upVote = (event) => {setVotes(votes + 1)};
  return (<div>
    <h1 className="Votes">{props.title}</h1>
    <p>Votes: {votes}</p>
    <p>
      <button onClick={upVote}>Up Vote</button>
    </p>
  </div>);
}
```

# Referințe

- Jake Archibald, *JavaScript Promises: an Introduction*,

<https://developers.google.com/web/fundamentals/getting-started/primers/promises>

- Using Fetch:

[https://developer.mozilla.org/en-US/docs/Web/API/Fetch\\_API/Using\\_Fetch](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch)

- Matt Gaunt, *Introduction to fetch()*,

<https://developers.google.com/web/updates/2015/03/introduction-to-fetch>

- Documentație React, <https://react.dev/>
- Versiuni mai vechi React: <https://legacy.reactjs.org/>

# Medii de proiectare și programare

2022-2023

Curs 12

# Continut

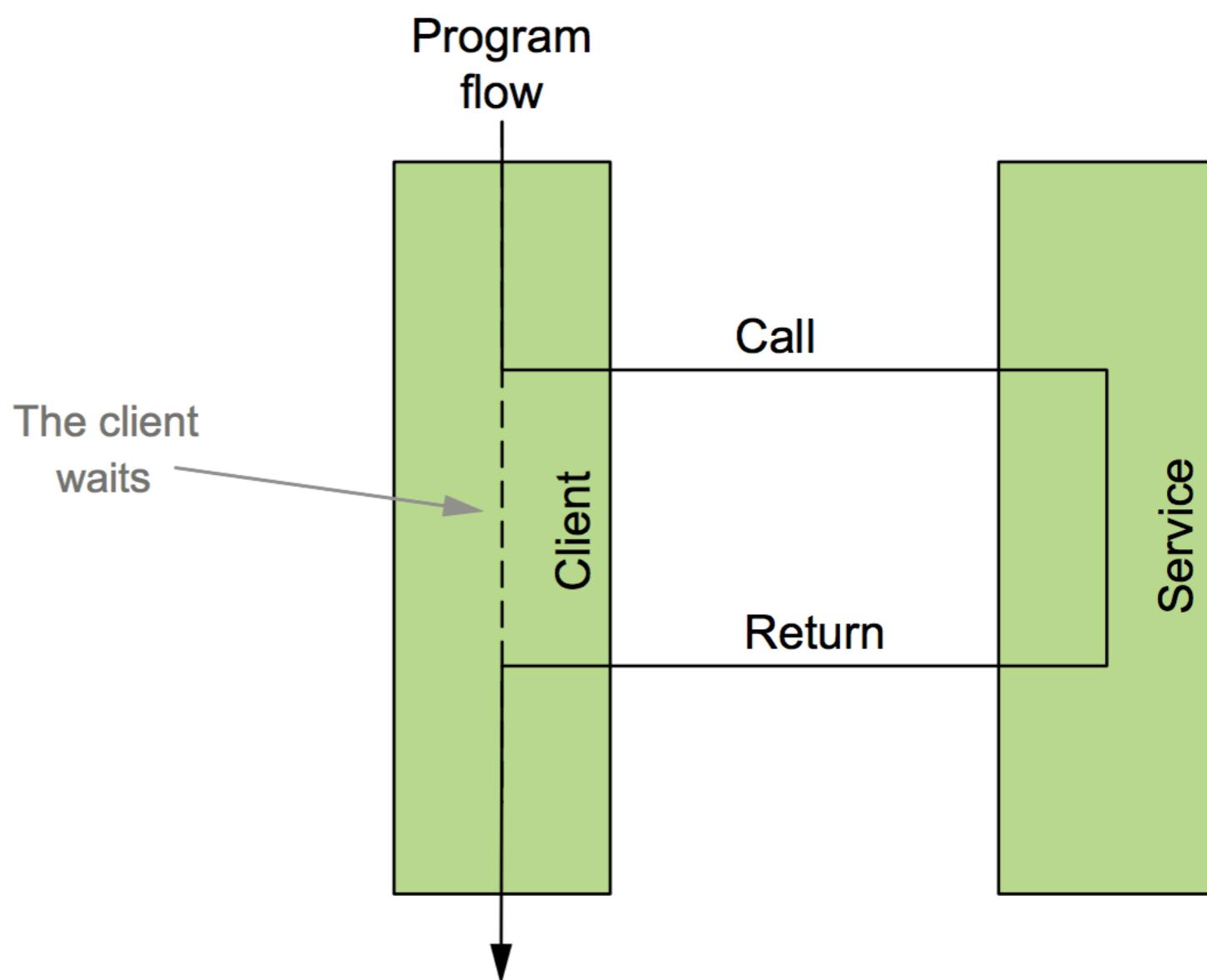
- Asynchronous messaging
- ActiveMQ - JMS, Spring JMS
- AQMP - RabbitMQ

# Asynchronous messaging

- *Asynchronous messaging* este o modalitate de a trămite indirect mesaje de la o aplicație la altă aplicație, fără a aștepta un răspuns.
- Este legată de comunicarea dintre aplicații.
- Diferă de alte mecanisme/modalități de comunicare prin modul în care informația este transmisă între sisteme.
- Are câteva avantaje față de comunicarea sincronă (eng. *synchronous messaging*).

# Comunicare sincronă

- Tehnologiile RPC ca gRPC și Thrift folosesc comunicarea sincronă.
- Când un client apelează o metodă a unui obiect remote, clientul trebuie să aștepte terminarea execuției înainte de a trece mai departe.
- Chiar dacă metoda nu returnează nimic, clientul tot trebuie să aștepte terminarea apelului.



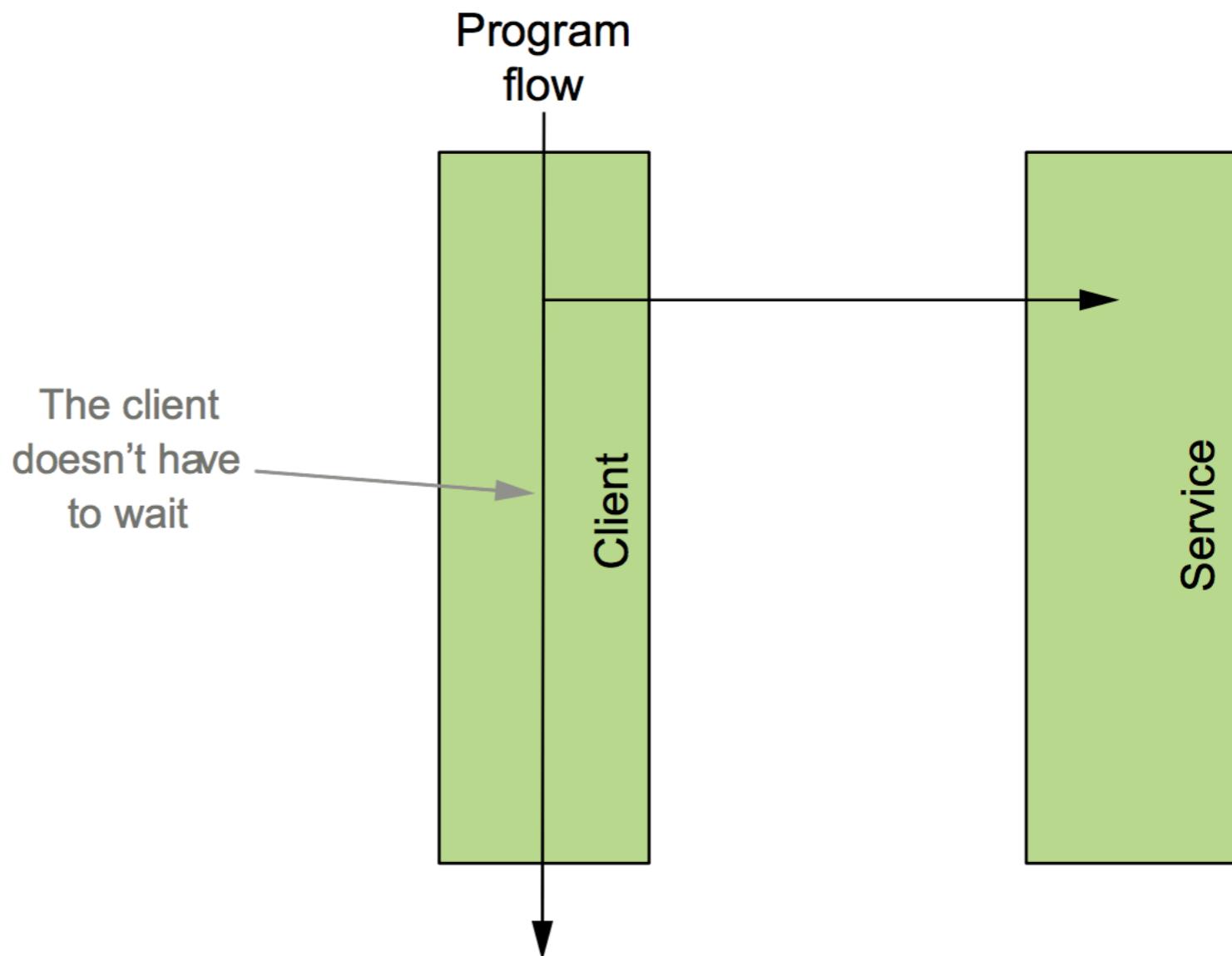
# Limitările comunicării sincrone

- Limitările clientului unui apel la distanță:

- *Comunicarea sincronă implică așteptare.* Când un client apelează o metodă a unui obiect/serviciu remote, acesta trebuie să aștepte terminarea execuției acesteia înainte de a continua execuția. Dacă clientul face apeluri dese sau dacă apelul se execută mai lent, acest apel ar putea afecta performanța clientului.
  - *Clientul este cuplat cu serviciul prin interfața serviciului.* Dacă interfața serviciului se modifică, toți clienții acestuia trebuie să se modifice corespunzător.
  - *Clientul este dependent de locația serviciului.* Clientul trebuie configurat pentru a specifica locația serviciului (IP și port) pentru a ști unde este localizat serviciul. Dacă locația serviciului se modifică, clientul trebuie reconfigurat pentru a specifica noua locație.
  - *Clientul este dependent de disponibilitatea (eng. availability) serviciului.* Dacă serviciul devine indisponibil, clientul va eșua.
- Aceste limitări trebuie luate în considerare în momentul deciderii folosirii tipului de comunicare de mesaje potrivit aplicației.

# Comunicarea asincronă

- Când mesajele sunt trimise asincron, clientul nu trebuie să aștepte terminarea procesării mesajului sau livrarea mesajului.
- Clientul trimită mesajul și trece la execuția următoarei instrucțiuni, presupunând că serviciul va primi și va prelucra la un moment dat mesajul trimis.
- Acest tip de comunicare are câteva avantaje asupra comunicării sincrone.

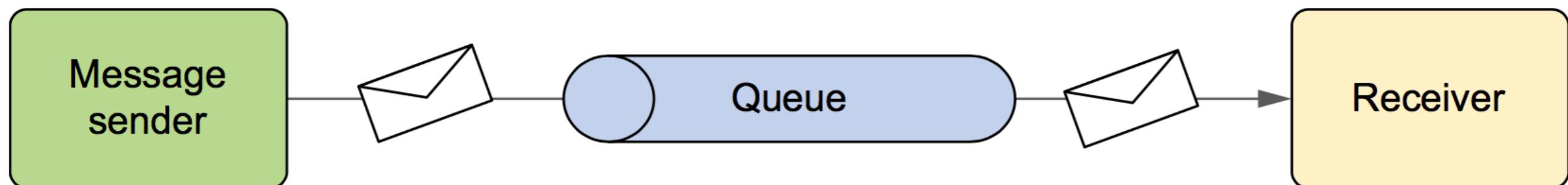


# Asynchronous messaging

- Când o aplicație trimite un mesaj altei aplicații, nu există o legătura directă între cele două aplicații.
- Aplicația care trimite mesajul, îl trimită unui serviciu care asigură livrarea acestuia aplicației care îl așteaptă.
- Există doi actori principali: **brokerii de mesaje** și **destinațiile**.
- Când o aplicație trimite un mesaj, îl trimită unui broker de mesaje.
- Brokerul de mesaje asigură livrarea mesajului la destinația specificată, permitând expeditorului continuarea execuției.
- Mesajele trimise asincron au specificată o destinație.
- Destinația specifică doar locația de unde mesajele pot fi preluate, nu și cine le va prelua.
- Tipuri diferite de sisteme de trimitere a mesajelor asincrone pot folosi scheme diferite de rutare a acestora.
- Există două tipuri de bază de destinații: **cozi** și **topic-uri**.
- Fiecare tip de destinație are asociat un model specific de trimitere a mesajelor:
  - **point-to-point** (corespunzător *cozilor*)
  - **publish/subscribe** (corespunzător *topicurilor*).

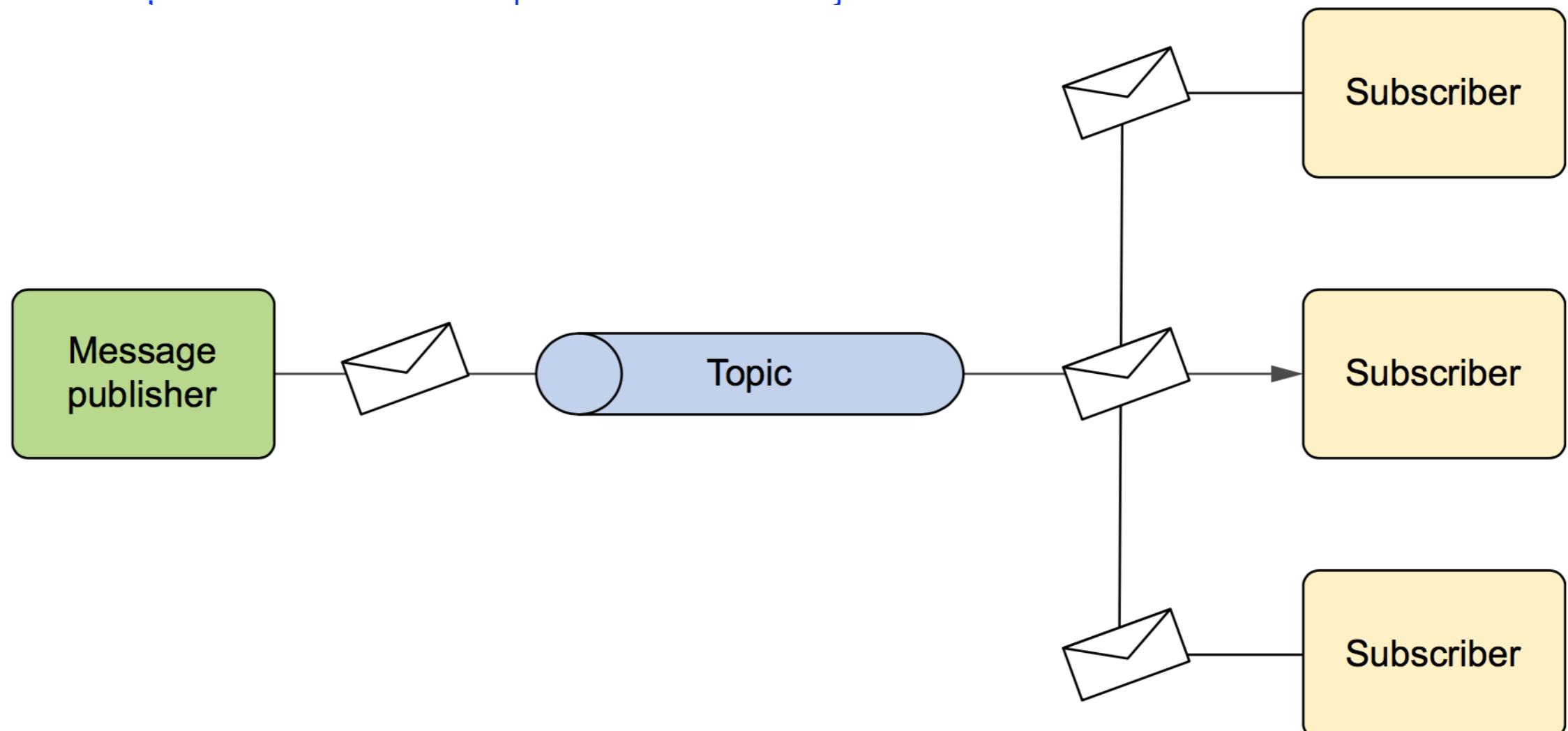
# Point-To-Point Model

- În **modelul point-to-point**, *fiecare mesaj are exact un expeditor și un destinatar*. Când brokerul de mesaje primește un mesaj, pune mesajul într-o coadă. Când destinatarul cere următorul mesaj, mesajul este scos din coadă și trimis acestuia.
- Deoarece mesajul este șters din coadă în momentul trimiterii, se garantează livrarea acestuia unui singur destinatar.
- Chiar dacă fiecare mesaj din coada de mesaje este livrat unui singur destinatar, nu înseamnă că doar un singur destinatar scoate mesaje din coadă.
- Este posibil ca mai mulți destinatari să proceseze mesaje din coadă, dar fiecare va primi propriul mesaj.
- **Dacă mai mulți destinatari așteaptă procesarea unui mesaj din coadă, nu se știe exact ce mesaj va procesa fiecare.** Acest lucru permite aplicațiilor să mărească capacitatea de procesare a mesajelor prin adăugarea unui nou destinatar.



# Publish-Subscribe Model

- În **modelul publish/subscribe**, mesajele sunt trimise unui *topic*. Asemănător cozilor, mai mulți destinații pot aștepta livrarea unor mesaje de la un topic. Spre deosebire de cozi, **când un mesaj este trimis unui topic toți destinații care așteaptă vor primi o copie a mesajului respectiv.**
- Expeditorul (publisher) nu știe care sunt destinații (subscribers). El știe doar că mesajul va fi trimis unui topic, nu știe cine așteaptă mesaje la topicul respectiv.
- Expeditorul nu știe nici cum vor fi procesate mesajele.



# Avantajele comunicării asincrone

- **Nu există timpi de așteptare**

- Când un mesaj este trimis asincron, clientul nu trebuie să aștepte livrarea sau procesarea mesajului. Clientul trimită mesajul brokerului de mesaje, cu încrederea că mesajul va ajunge la destinația specificată.
- Deoarece nu trebuie să aștepte, clientul poate executa alte sarcini, îmbunătățind performanța acestuia.

- **Orientată pe mesaje și decuplată**

- Spre deosebire de comunicarea RPC (orientată pe apelul unei metode), mesajele trimise asincron sunt centrate pe date. Clientul nu este dependent de o anumită semnătură a metodei. Fiecare abonat la o coadă sau un topic care poate procesa datele trimise de client, poate procesa mesajul. Clientul nu trebuie să știe particularitățile specifice serviciului.

- **Independența locației**

- Serviciile RPC sincrone sunt localizate de obicei prin adresa lor (IP și port), clienții fiind afectați de orice modificare a locației. Dacă adresa IP sau portul se modifică, clientul trebuie să fie modificat corespunzător, altfel clientul nu va mai putea accesa serviciul.

# Avantajele comunicării asincrone

- ***Independența locației (cont.)***

- În contrast, clienții sistemelor bazate pe mesaje nu știu ce serviciu va procesa mesajele lor sau locația acestora. Clientul știe doar de coada sau topicul prin intermediul căreia/căruia mesajul va fi transmis. Nu contează locația serviciului, atât timp cât acesta poate procesa mesajele din coadă sau topic.
- Servicii multiple se pot abona la un singur topic, fiecare primind o copie a aceluiasi mesaj, dar fiecare poate procesa în mod diferit mesajul. **Fiecare serviciu va folosi în mod independent aceeași informație primită de la topic.**
- În modelul point-to-point, **se pot crea clustere de servicii**. Dacă clientul nu este dependent de locația serviciului, și singura cerință legată de serviciu este că acesta trebuie să poată accesa brokerul de mesaje, **mai multe servicii pot fi configurate să primească mesaje de la aceeași coadă**. Dacă la un moment dat un serviciu devine suprasolicitat, se pot porni noi instanțe ale aceluiasi serviciu care se abonează la aceeași coadă.

# Avantajele comunicării asincrone

- **Livrare garantată**

- Pentru ca un client să poată comunica cu un serviciu sincron, serviciul trebuie să fie disponibil la adresa și portul specificate. Dacă serviciul devine indisponibil (temporar sau pe o perioadă nedefinită), clientul nu își poate continua execuția.
- Dacă mesajele sunt trimise asincron, clientul are garanția că mesajele sale vor fi livrate. Chiar dacă un serviciu nu este disponibil în momentul trimiterii mesajului, mesajul va fi păstrat de brokerul de mesaje până când serviciul va deveni din nou disponibil.
- Disponibilitatea brokerului de mesaje

# Brokeri de Mesaje

- **Apache ActiveMQ** (<http://activemq.apache.org/>)

- Open source
- Suportă clienți din limbaje și protocoale diferite: Java, C, C++, C#, Ruby, Perl, Python, PHP.
- Oferă suport pentru frameworkul Spring, ActiveMQ putând fi ușor integrat și configurat în aplicații bazate pe Spring.
- Suportă concepte avansate: grupuri de mesaje, destinații virtuale, destinații compuse

- **RabbitMQ** (<https://www.rabbitmq.com/>)

- Open source
- Simplu și ușor de integrat și folosit (și in cloud).
- Suportă mai multe protocoale de comunicare bazate pe mesaje.
- Poate fi rulat pe diferite sisteme de operare și în cloud. Oferă instrumente de dezvoltare pentru diferite limbaje de programare (Java, .NET, PHP, Ruby, Python, etc).

- **Apache Kafka, Amazon MQ**, etc

# ActiveMQ

- Instalarea:
  - Arhiva <http://activemq.apache.org/>
- Pornirea:
  - MacOS: *activemq start*
- Verificarea pornirii corecte:
  - <http://127.0.0.1:8161/admin>
  - Cont: user *admin*; pass *admin*
- Oprirea
  - MacOS: *activemq stop*

# Java Message Service (JMS)

- Un standard Java care definește o interfață comună pentru lucrul cu brokeri de mesaje.
- Înainte de apariția JMS, fiecare broker de mesaje avea propriul API făcând codul aplicațiilor dependent de broker și greu de reutilizat pentru brokeri diferiți.
- Folosind JMS, toți brokerii de mesaje pot fi accesați folosind aceeași interfață comună (asemănător JDBC).

# Trimiterea unui mesaj

```
ConnectionFactory cf = new ActiveMQConnectionFactory("tcp://localhost:61616");
Connection conn = null;
Session session = null;
try {
    conn = cf.createConnection(); conn.start();
    session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
    Destination destination = new ActiveMQQueue("test.queue");
    MessageProducer producer = session.createProducer(destination);
    TextMessage message = session.createTextMessage();
    message.setText("Hello world!");
    producer.send(message); //trimiterea mesajului
} catch (JMSEException e) {
    // handle exception ...
} finally {
    try {
        if (session != null) { session.close(); }
        if (conn != null) { conn.close(); }
    } catch (JMSEException ex) { //... tratarea exceptiilor
    }
}
```

# Citirea unui mesaj

```
ConnectionFactory cf = new ActiveMQConnectionFactory("tcp://localhost:61616");
Connection conn = null;
Session session = null;
try {
    conn = cf.createConnection();
    conn.start();
    session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
    Destination destination = new ActiveMQQueue("test.queue");
    MessageConsumer consumer = session.createConsumer(destination);
    Message message = consumer.receive();
    TextMessage textMessage = (TextMessage) message;
    System.out.println("Primit mesaj: " + textMessage.getText());
} catch (JMSEException e) {
    // handle exception ...
} finally {
    try {
        if (session != null) { session.close(); }
        if (conn != null) { conn.close(); }
    } catch (JMSEException ex) { //... tratarea exceptiilor
    }
}
```

# Spring JMS

- Crearea unui *Connection Factory*
  - **ActiveMQConnectionFactory** este clasa JMS connection factory oferită de ActiveMQ

```
<bean id="connectionFactory"
      class="org.apache.activemq.spring.ActiveMQConnectionFactory" />
```

Implicit, **ActiveMQConnectionFactory** presupune că brokerul de mesaje așteaptă la locația **localhost** pe portul **61616**.

```
<bean id="connectionFactory"
      class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616"/>
</bean>
```

Schimbarea locației implicate se poate face folosind proprietatea **brokerURL** a brokerului de mesaje.

# Spring JMS

- Declararea unei destinații pentru mesaje folosind ActiveMQ
    - Pentru trimiterea și livrarea mesajelor este necesară definirea unei destinații.
    - În funcție de tipul aplicației, destinația poate fi o *coadă* sau un *topic*.
    - Independent de tipul de destinație folosit, destinația trebuie specificată folosind clasa specifică corespunzatoare brokerului de mesaje folosit.
    - Declararea unei cozi
- ```
<bean id="queue" class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg value="test.queue"/>
</bean>
```
- Declararea unui topic
- ```
<bean id="topic" class="org.apache.activemq.command.ActiveMQTopic">
    <constructor-arg value="test.topic"/>
</bean>
```
- Constructorul primește ca și parametru numele cozii de mesaje/topicului.
  - Numele va fi folosit de brokerul de mesaje pentru trimiterea/livrarea mesajelor.
  - **test.queue/test.topic**.

# Template-uri Spring JMS

- Clasa **JmsTemplate** este soluția oferită de Spring pentru a evita duplicarea codului JMS.
- **JmsTemplate** are ca și responsabilități: crearea conexiunii, obținerea unei sesiuni și trimitera și livrarea mesajelor.
- Programatorii se ocupă doar de construirea mesajelor ce trebuie trimise, respectiv de procesarea mesajelor primite.
- **JmsTemplate** tratează toate excepțiile de tip **JMSException** (pachetul `javax.jms`) ce pot să apară și le convertește în propriile excepții de tip **JMSException** (pachetul `org.springframework.jms`).
- Dacă o excepție **JMSException** este aruncată pe perioada lucrului cu **JmsTemplate**, obiectul de tip **JmsTemplate** va prinde excepția și o va rearunca sub forma unei excepții specifice Spring (subclasă a excepției proprii **JmsException**).
- **JMSException** (din pachetul `javax.jms`) are o mulțime de subclase care descriu tipul de excepție apărut, dar toate sunt de tip checked.
- **JmsTemplate** prinde excepțiile checked și le rearuncă folosind excepții de tip runtime.

# Template-uri Spring JMS

- Pentru a putea folosi obiecte de **JmsTemplate**, acestea trebuie declarate la configurare:

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate">

    <constructor-arg ref="connectionFactory"/>

</bean>
```

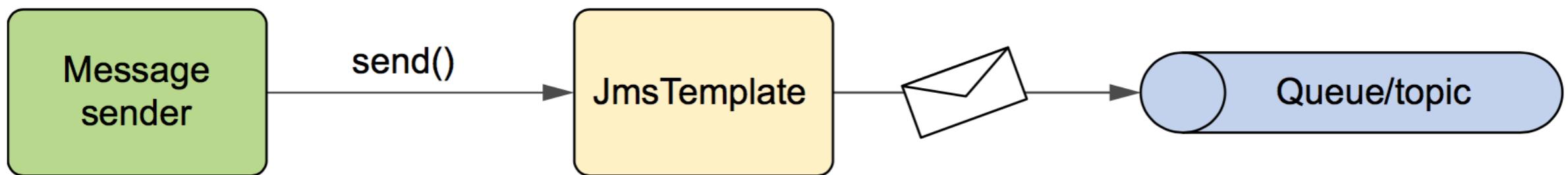
- Se pot seta și alte proprietăți.
- Clasa **JmsTemplate** implementează interfața **JmsOperations**
- Interfața **JmsOperations** specifică operațiile

# Spring JMS - trimiterea unui mesaj

```
public interface AlertService {  
    void sendAlert(Alert alert);  
}  
  
public class AlertServiceImpl implements AlertService {  
    private JmsOperations jmsOperations; //interfata implementata de JmsTemplate  
    public AlertServiceImpl(JmsOperations jmsOperatons) {  
        this.jmsOperations = jmsOperations;  
    }  
    public void sendAlert(Alert alert) {  
        //trimiteaza unui mesaj  
        jmsOperations.send("alert.queue", //Specificarea destinatiei  
                           new MessageCreator() //Crearea mesajului  
                           public Message createMessage(Session session)  
                               throws JMSEException {  
                               return session.createObjectMessage(alert);  
                           }  
        } );  
    }  
}//AlertServiceImpl
```

# Spring JMS - trimiterea unui mesaj

- Primul parametru al operației `send()` (interfața `JmsOperations`) specifică numele destinației (coadă sau topic).
- La apelul metodei `send()`, obiectul `JmsTemplate` se ocupă cu obținerea unei conexiuni și a unei sesiuni JMS și cu trimitera mesajului din partea expeditorului.
- Mesajul este construit folosind un obiect de tip `MessageCreator`
- Metoda `createMessage()` (din clasa `MessageCreator`), cere sesiunii crearea unui mesaj trimitând ca și parametru obiectul `Alert` care păstrează datele conținute de mesajul transmis.
- Obiectul `JmsTemplate` tratează toate complexitățile trimiterii unui mesaj.



# Spring JMS - trimiterea unui mesaj

- Specificarea unei destinații implicate:
  - De obicei, în cadrul unei aplicații mesajele se vor trimite la aceeași destinație.
  - Se poate specifica o destinație implicită în momentul instantierii unui obiect de tip **JmsTemplate**.
  - Nu mai este necesară specificarea destinației în momentul trimiterii unui mesaj.

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate">
    <constructor-arg ref="connectionFactory"/>
    <property name="defaultDestination" ref="queue"/>
</bean>
```

```
//Nu mai e necesara specificarea destinatiei
jmsOperations.send(
    new MessageCreator() {
        ...
    }
);
```

# Spring JMS - trimiterea unui mesaj

- Convertirea mesajelor la trimitere
  - **JmsTemplate** oferă **convertAndSend()**. Spre deosebire de **send()**, metoda **convertAndSend()** nu primește ca și parametru un obiect de tip **MessageCreator**.
  - Metoda **convertAndSend()** folosește un obiect de tip message converter predefinit pentru a crea mesaje.

```
public void sendAlert(Alert alert) {  
    jmsOperations.convertAndSend(alert);  
}
```

- **JmsTemplate** folosește o implementarea a interfeței **MessageConverter** pentru a converti obiectele în mesaje.
- **MessageConverter** este o interfață definită de Spring având metodele:

```
public interface MessageConverter {  
    Message toMessage(Object object, Session session)  
        throws JMSException, MessageConversionException;  
    Object fromMessage(Message message)  
        throws JMSException, MessageConversionException;  
}
```

# MessageConverters

- Implicit **JmsTemplate** folosește un **SimpleMessageConverter** pentru trimiterea mesajelor folosind **convertAndSend()**.

- Se poate specifica alt tip de convertor la configurarea obiectului de tip JmsTemplate.

```
<bean id="converter"
      class="org.springframework.jms.support.converter.MappingJackson2MessageConverter">
    <property name="targetType" value="TEXT"/>
    <property name=" typeIdPropertyName" value="_alert"/>
</bean>
```

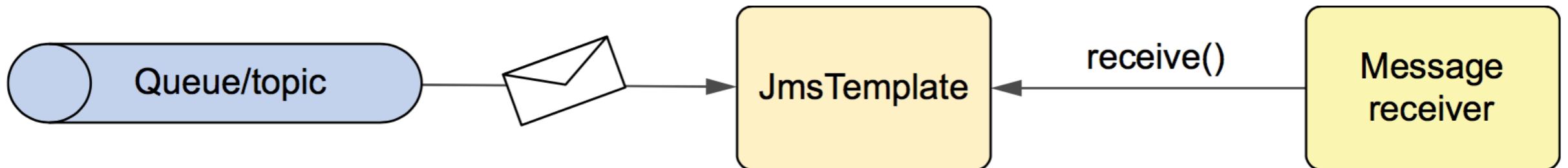
```
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
  <constructor-arg ref="connectionFactory"/>
  <property name="defaultDestination" ref="topic"/>
  <property name="messageConverter" ref="converter"/>
</bean>
```

- MappingJackson2MessageConverter** (JSON),
- MarshallingMessageConverter** (JAXB XML format),
- SimpleMessageConverter** (Strings to/from **TextMessage**, byte arrays to/ from **BytesMessage**, Maps to/from **MapMessage**, și Serializable objects to/from **ObjectMessage**)

# Primirea/Procesarea mesajelor

- Apelul metodei `receive()` din `JmsOperations`.
- La apelul metodei `receive()`, se încearcă obținerea unui mesaj de la brokerul de mesaje. **Dacă nici un mesaj nu este disponibil se așteaptă până când un mesaj este trimis la destinație.**

```
public Alert receiveAlert() {  
    try {  
        ObjectMessage receivedMessage = (ObjectMessage) jmsOperations.receive();  
        return (Alert) receivedMessage.getObject();  
    } catch (JMSEException jmsException) {  
        throw JmsUtils.convertJmsAccessException(jmsException);  
    }  
}
```



# Primirea/Procesarea mesajelor

- **MessageConverters** pot converti obiecte în mesaje la apelul metodei **convertAndSend()**.
- Pot fi folosiți și la procesarea mesajelor, prin apelul metodei **receiveAndConvert()** (**JmsTemplate**):

```
public Alert retrieveAlert() {  
    return (Alert) jmsOperations.receiveAndConvert();  
}
```

- Un **dezavantaj** al procesării mesajelor folosind **metodele receive()** și **receiveAndConvert()** este că ambele metode sunt sincrone.
- Apelantul metodelor trebuie să aștepte până când un mesaj devine disponibil (sau o condiție de timeout devine adevărată)
- **Message driven beans** (EJB) - pentru procesarea asincronă a mesajelor.

# Exemple

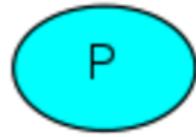
- Exemplu simplu ActiveMQ, producător/consumator Java, producător/consumator C#
- Implementarea notificării în aplicația MiniChat folosind un broker de mesaje.

# Advanced Message Queuing Protocol (AMQP)

- Protocol avansat pentru trimiterea mesajelor.
- În JMS există 3 tipuri de participanți:
  - *expeditorul mesajului* (producătorul),
  - *destinatarul* (destinatarii) mesajului (consumator/consumatori),
  - *destinația* (coadă sau topic) folosită pentru livrarea mesajelor între expeditori și destinatari.
- În JMS, destinația decuplează expeditorul de destinatar, dar amândoi sunt cuplați de destinație.
- Destinația are dublă responsabilitate:
  - *primirea mesajelor*
  - determinarea modului în care mesajele vor fi distribuite destinatarilor: coadă/ topic.

# RabbitMQ - Terminologie

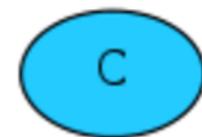
**Producător** (eng. *producer*): un program care trimite mesaje.



**Coadă** (eng. *queue*): o zonă tampon mare de mesaje, limitată doar de configurația mașinii pe care rulează RabbitMQ și dimensiunea HDD. Mai mulți producători pot trimite mesaje aceleiasi cozi. Mai mulți consumatori pot citi/primi mesaje de la aceeași coadă.

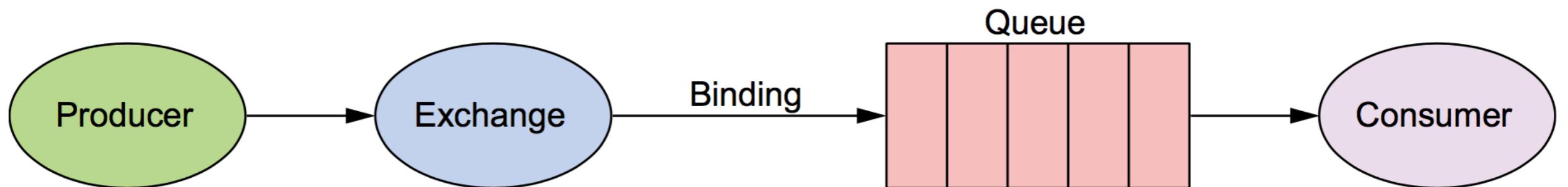


**Consumator** (eng. *consumer*): un program care așteaptă să primească mesaje.



# Advanced Message Queuing Protocol (AMQP)

- Producătorii AMQP nu trimit mesajele direct unei cozi.
- AMQP introduce un nou nivel între producător și cozile care păstrează mesajele: **exchange**.
- Un producător de mesaje trimite mesajul unui exchange.
- Exchange-ul (care are asociate una sau mai multe cozile), rutează mesajele cozilor corespunzătoare.
- Consumatorii iau mesajele din cozi și le procesează. Algoritmul de rutare folosit are un impact minor asupra modului de dezvoltare a producătorilor/consumatorilor.
- Producătorii trimit mesajele unui exchange cu o **cheie de rutare** (eng. *routing key*), consumatorii iau mesajele din coada (adesea mesajele nu mai conțin cheia de rutare).
- RabbitMQ este un broker de mesaje bazat pe AMQP.

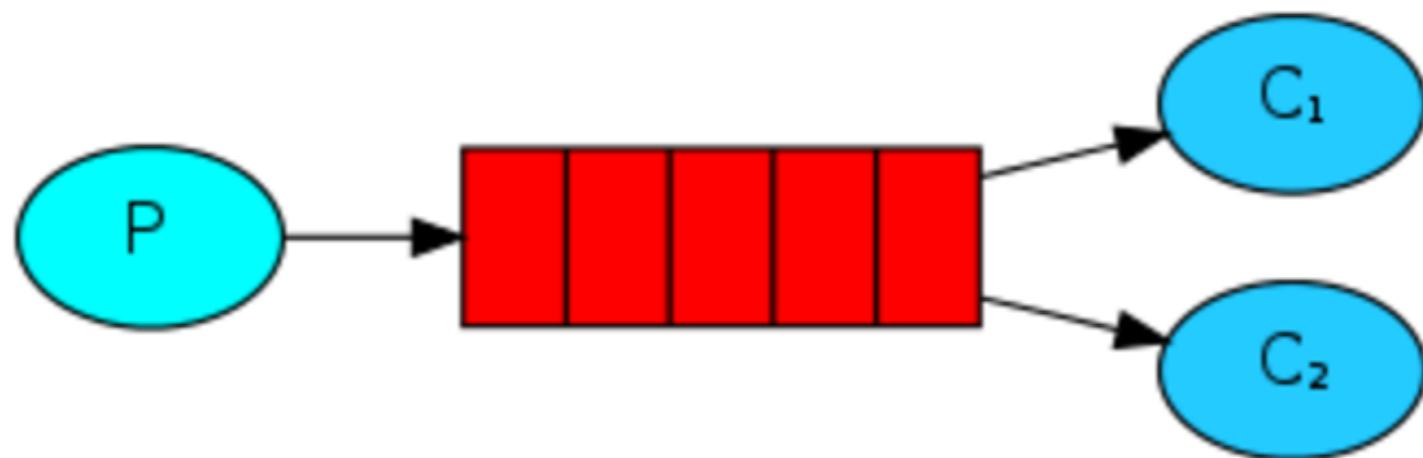


# RabbitMQ - Modele asemănătoare JMS

- Model simplu:

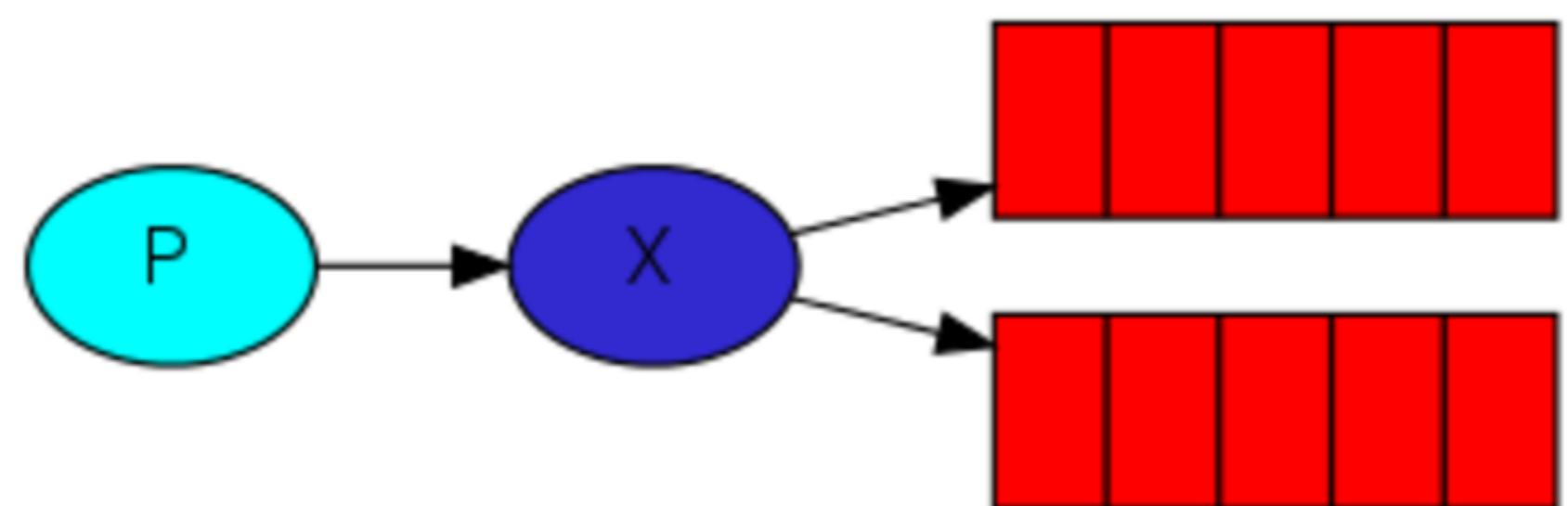


- Cozi de lucru:



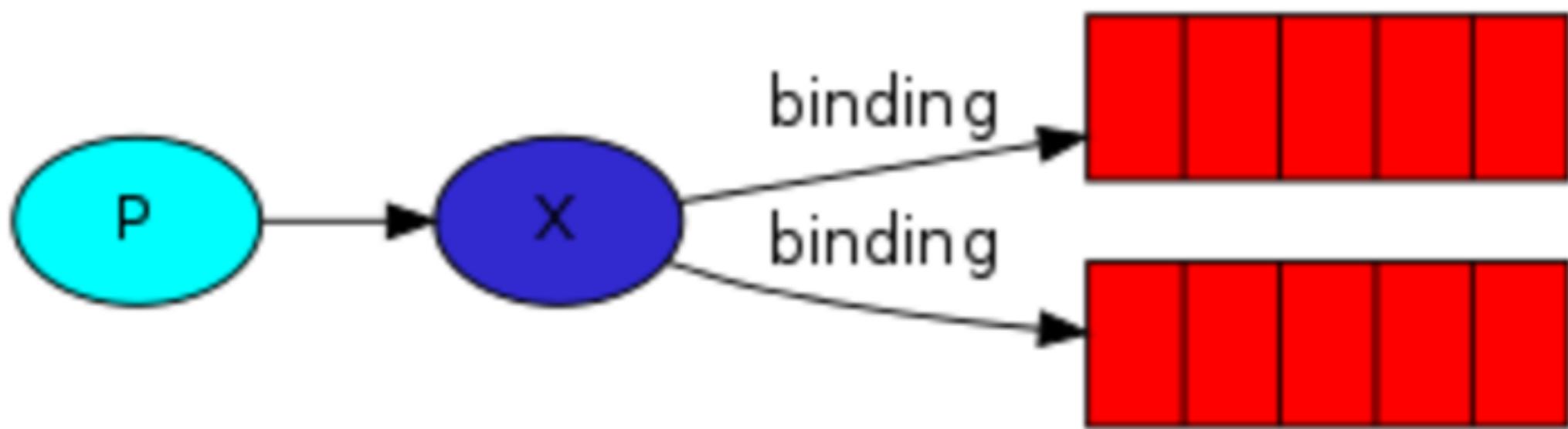
# RabbitMQ - Model de transmitere a mesajelor

- Ideea de bază a modelului RabbitMQ este că producătorul NU trimite mesajul direct unei cozi.
- Producătorul nu știe nici dacă mesajul chiar va fi distribuit unei cozi.
- Producătorul poate doar să trimită mesaje unui *exchange*.
- Un *exchange* primește mesajele de la producători și le trimită spre cozile corespunzătoare.
- Un *exchange* trebuie să știe exact ce să facă cu un mesaj în momentul primirii:
  - Trebuie adăugat la o anumită coadă?
  - Trebuie adăugat la mai multe cozile?
  - Trebuie șters (eng. *discarded*)?
- Regulile folosite pentru luarea deciziei sunt definite de ***tipul de exchange***:
  - *direct*,
  - *topic*,
  - *headers*
  - *fanout*.



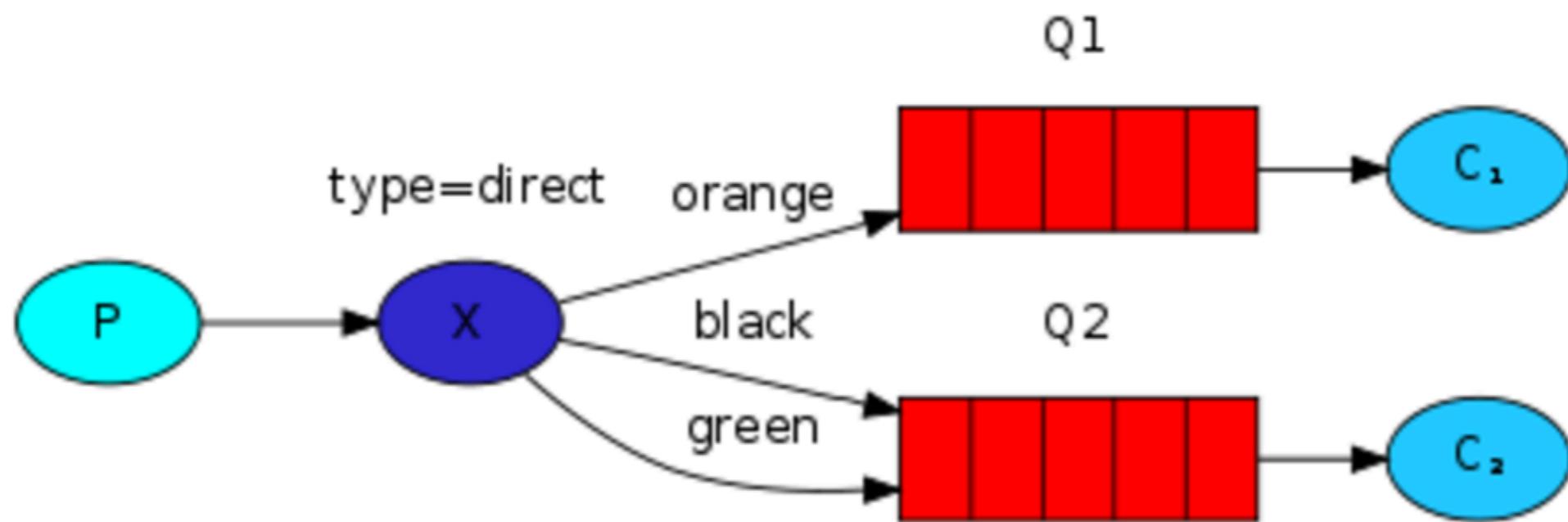
# RabbitMQ - Fanout exchange

- **Fanout exchange** - distribuie mesajele tuturor cozilor cunoscute.
- Asocieri (eng. *bindings*): Relația dintre un exchange și o coadă este numită **asociere** (binding). Coada este interesată să primească mesaje de la exchange.
- Asocierile pot avea un extra parametru, numit **cheie de asociere** (eng. *binding key*)
- Semnificația cheii de asociere depinde de tipul de exchange.
- Fanout exchanges ignoră această valoare.



# RabbitMQ - Direct exchange

- **Direct exchange**: - mesajul va fi distribuit cozilor care au asociate ca și chei exact cheia de rutare a mesajului.
- Exemplu:
  - direct exchange X cu două cozi asociate (Q1, Q2). Prima coadă are cheia de asociere *orange*, a doua coadă are asociate cheile: *black* și *green*.
  - Un mesaj cu cheia de rutare *orange* va fi distribuit cozii Q1.
  - Mesajele cu cheile de rutare *black* sau *green* vor fi distribuite cozii Q2.
  - Mesajele cu alte chei de rutare (ex. *white*, *red*, etc.) se vor pierde.

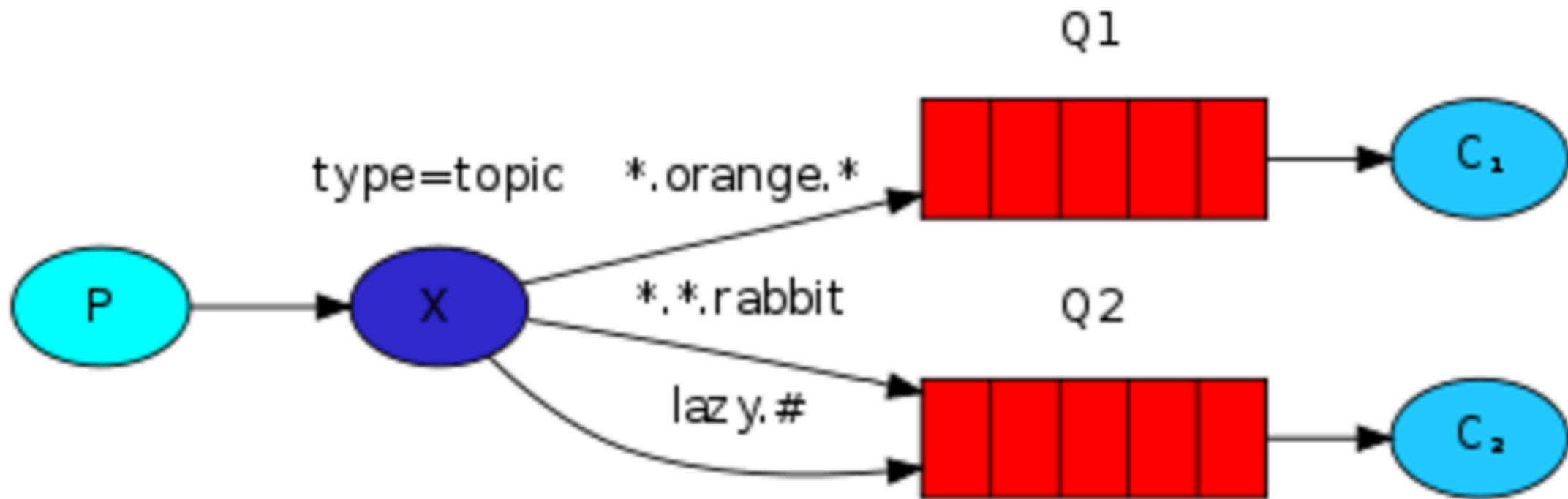


# RabbitMQ - Topic exchange

- Mesajele trimise unui **topic exchange** nu pot avea o cheie de rutare arbitrară.
- Cheia de rutare trebuie să fie o listă de cuvinte separate prin punct. Cuvintele sunt de obicei caracteristici ale mesajelor.
- Exemple valide: "stock.usd.nyse", "nyse.vmw", "quick.orange.rabbit".
- În *cheia de rutare* pot fi oricâte cuvinte, dar lungimea nu trebuie să depășească 255 bytes.
- Cheia folosită pentru specificarea asocierilor are aceeași formă.
- Un mesaj având o anumită cheie de rutare va fi trimis tuturor cozilor asociate cu o cheie de asociere care se potrivește.
- Cazuri speciale pentru cheile folosite la specificarea asocierilor:
  - Caracterul \* (star) poate înlocui un singur cuvânt.
  - Caracterul # (hash) poate înlocui zero sau mai multe cuvinte.
- Topic exchange se poate comporta ca și celelalte tipuri de exchange:
  - Când o coadă are asociată cheia "#" (hash) - va primi toate mesajele indiferent de cheia de rutare folosită (*fanout exchange*).
  - Dacă nu se folosesc caracterele speciale "\*" și "#" la asocierile cozilor, topic exchange se va comporta ca și un *direct exchange*.

# RabbitMQ -Exemplu (1)

- Trimiterea unor mesaje care descriu animale.
- Cheia de rutare este formată din 3 cuvinte: "<viteză>.<culoare>.<animal>".
  - S-au creat 3 asocieri (eng. *bindings*): coada **Q1** este asociată cu cheia de asociere "**\*.orange.\***", iar coada **Q2** cu cheile "**\*.\*.rabbit**" și "**lazy.#**".
  - Sumar asocieri:
    - **Q1** este interesată de toate animalele având culoarea *orange*.
    - **Q2** este interesată de toți iepurii și de animale - speciile leneșe.



# RabbitMQ - Exemplu (2)

Cheia de rutare - Mesaj	Coadă
quick.orange.rabbit	Q1, Q2
lazy.orange.elephant	Q1, Q2
quick.orange.fox	Q1
lazy.brown.fox	Q2
lazy.pink.rabbit	Q2 (o dată)
quick.brown.fox	Nici una
orange	Nici una
quick.orange.male.rabbit	Nici una
lazy.orange.male.rabbit	Q2

# RabbitMQ

- **Headers Exchange:**

- Este proiectat pentru rutarea folosind mai multe atribute, care sunt păstrate în antetul (header-ul) mesajului, și nu ca o cheie de rutare.
- **Cheia de rutare este ignorată.** Atributele folosite pentru rutare sunt luate din antet.
- Un mesaj este o *potrivire*, dacă valoarea din antet este egală cu valoarea precizată la asociere.
- Este posibil să se facă asocierea dintre o coadă și un exchange folosind mai multe antete.
- În acest caz, broker-ul are nevoie de informații adiționale:
  - Ar trebui să ia în considerare mesajele care conțin cel puțin un antet sau toate antetele?
  - Se poate configura opțiunea folosind argumentul *x-match* de la binding:
    - *any* - este de ajuns ca un singur antet să apară
    - *all* - toate antetele trebuie să apară.

# Referințe

- RabbitMQ:  
<https://www.rabbitmq.com/getstarted.html>
- Craig Walls, Spring in Action, Fourth Edition, Ed. Manning, 2015
- ActiveMQ:  
<http://activemq.apache.org/hello-world.html>

# Medii de proiectare și programare

2022-2023

Curs 13

# Continut

- WebSockets
- Autentificare REST
  - JWT
- Examen

# HTTP

## Limitările HTTP:

- *Stateless*:
  - Browserul deschide o conexiune pe portul 80.
  - Trimit o cerere HTTP serverului web.
  - Aplicația server decide ce va face cu cererea, procesează datele, generează răspuns HTML și îl trimit serverului web.
  - Serverul web adaugă antetele HTTP corespunzătoare răspunsului, îl trimit browserului și închide conexiunea.
- Site-urile web trebuie să păstreze informațiile despre utilizatori (ex. cookies).
  - Informația este transmisă între client și serverul web la fiecare cerere.
  - Informațiile adiționale măresc dimensiunea pachetelor transmise și pot face aplicațiile web vulnerabile (securitatea datelor).
- Comunicarea este întotdeauna inițiată de client și fiecare pereche cerere/răspuns este independentă de celelalte cereri/răspunsuri.

# HTTP

- Aplicații *real-time*: burse de valori, vânzarea biletelor, trafic, citirea aparatelor medicale
- Metode:
  - *Polling*, browserul *trimite regulat* cereri HTTP și primește răspunsul imediat. Este o soluție bună dacă se cunosc intervalele la care mesajele devin disponibile, deoarece se pot sincroniza cererile clientilor cu momentele când informațiile sunt disponibile pe server.
  - *Long-polling*, browserul trimite o cerere la server și serverul păstrează cererea deschisă pentru o perioadă prestabilită de *temp*. Dacă în acea perioadă se primește o notificare, serverul trimită clientului un răspuns care conține și notificarea. Dacă nu se primește nici o notificare, serverul trimită un răspuns pentru a încheia cererea clientului.
  - *Streaming*, browserul trimite o cerere completă dar serverul trimită și păstrează un răspuns incomplet ce este actualizat continuu și păstrat deschis pe termen nelimitat (sau o perioadă prestabilită de *temp*). Răspunsul este actualizat de fiecare dată când trebuie trimisă o notificare, dar serverul nu semnalează terminarea răspunsului, păstrând astfel conexiunea deschisă pentru mesaje ulterioare. Pot să apară probleme din cauza păstrării într-un buffer a răspunsurilor.

# WebSockets

- WebSocket-urile sunt o **conexiune persistentă, bi-directională, full-duplex** de la un browser web la un server.
- După ce conexiunea a fost stabilită, ea rămâne deschisă până când clientul sau serverul decide să o închidă.
- Cât timp conexiunea este deschisă, clientul și serverul își pot trimite mesaje unul altuia în orice moment de timp.
- Programarea web devine astfel bazată pe eveniment, și nu doar inițiată de utilizator. Este *stateful*.
- O singură aplicație server care rulează știe de toate conexiunile, permitând comunicarea cu orice număr de conexiuni deschise în orice moment de timp.

# Protocolul WebSocket

- În 2011, IETF a standardizat protocolul WebSocket sub denumirea de RFC 6455.
- De atunci, majoritatea browserelor Web implementează cod API pe partea de client care suportă protocolul WebSocket.
- S-au dezvoltat biblioteci în Java, .NET, Ruby, Objective C, JavaScript, etc. care implementează protocolul WebSocket.
- Browsere care suportă WebSockets: nativ în Chrome, Firefox, Opera și Safari (inclusiv Safari pentru dispozitive mobile), Internet Explorer.
- Orice browser care nu suportă WebSockets poate folosi soluția *polyfill Flash*.

# Protocolul WebSocket - Handshake

Pentru stabilirea unei conexiuni WebSocket, clientul trimite o cerere HTTP pentru un *handshake* WebSocket:



```
GET /echo HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGh1IHNhbXBsZSSub25jZQ==
Sec-WebSocket-Version: 13
Origin: http://example.com
```

# Protocolul WebSocket - Handshake

- Dacă serverul acceptă cererea de actualizare a protocolului, va returna un răspuns **HTTP 101 Switching Protocols**:



**HTTP/1.1 101 Switching Protocols**

**Upgrade: websocket**

**Connection: Upgrade**

**Sec-WebSocket-Accept: HSmrc0sM1YUkAGmm5OPpG2HaGWk=**

**Sec-WebSocket-Protocol: echo**

- După ce serverul returnează răspunsul 101, protocolul la nivel de aplicație se va schimba din **HTTP** în **WebSockets**, care va folosi conexiunea TCP stabilită anterior. Din acest moment, ambii participanți pot trimite/primi mesaje în orice moment de timp.



# WebSocket - URI

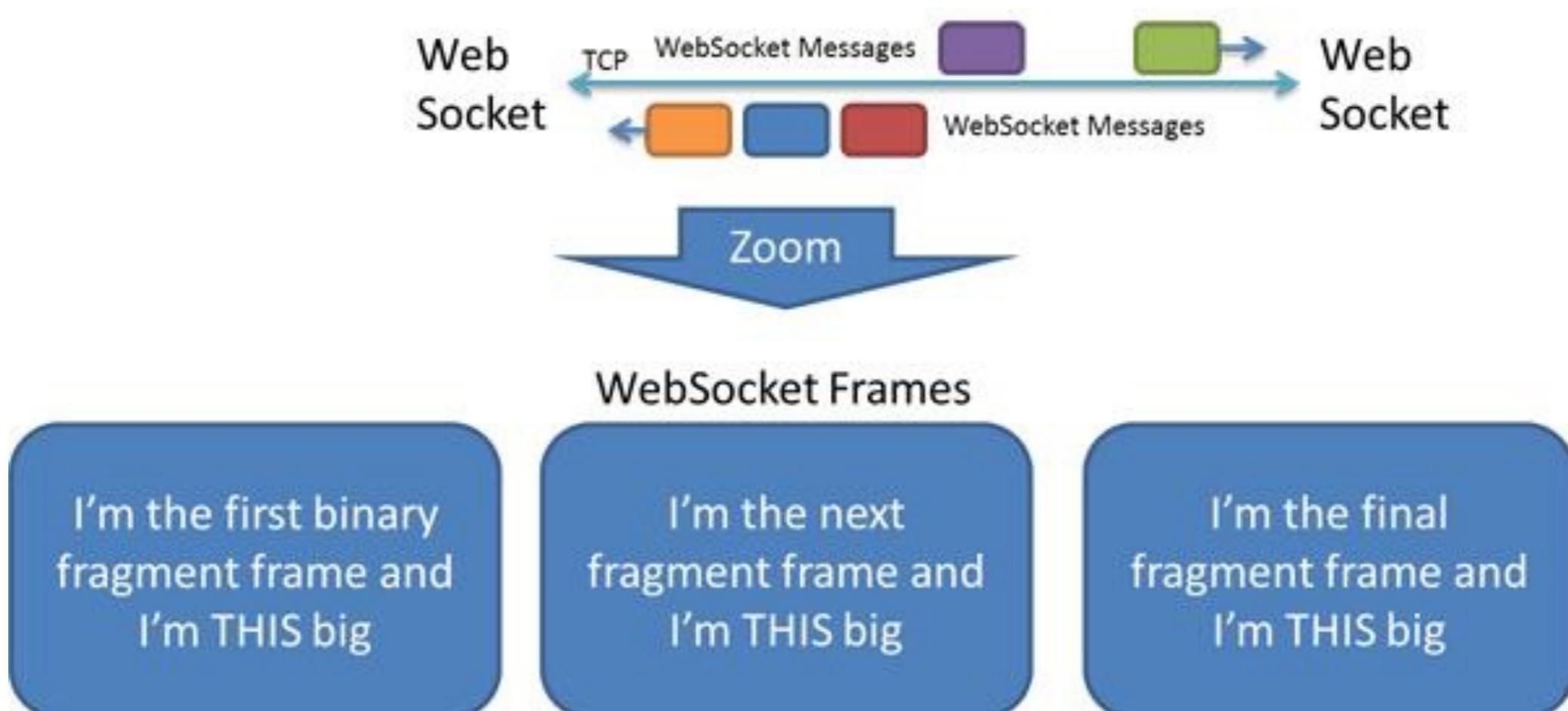
- Protocolul WebSocket definește două formate URI noi, similare cu formatele HTTP:
  - "ws://" // host [ ":" port ] path [ "?" query ] modelată după formatul "http:"
    - Portul implicit este 80.
    - Este folosit pentru **conexiuni nesecurizate** (necriptate).
  - "wss://" // host [ ":" port ] path [ "?" query ] modelată după formatul "https:"
    - Portul implicit este 443.
    - Este folosit pentru **conexiuni securizate** peste Transport Layer Security (TLS).

`ws://example.com`

`ws://example.com:8080/echo`

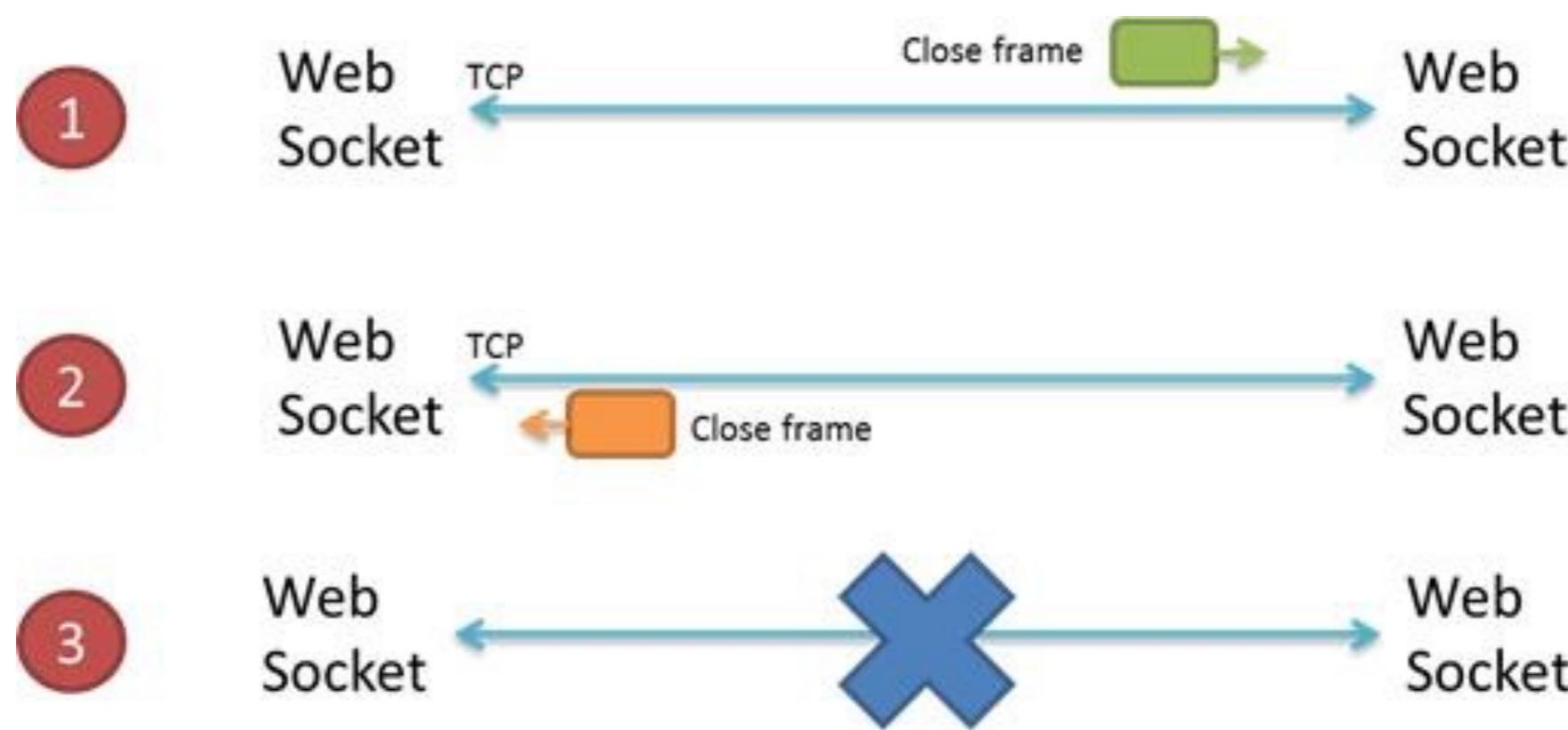
# Mesaje WebSocket

- După o înțelegere (handshake) cu succes, aplicația și serverul web pot inter schimba mesaje WebSocket.
- Un mesaj este compus dintr-o secvență de unul sau mai multe fragmente de mesaj/date numite “*frames*”. Fiecare frame conține informații cum ar fi:
  - Dimensiunea/lungimea framelui.
  - Primul frame din mesaj va conține tipul mesajului (text sau binar).
  - Un flag (*FIN*) care indică dacă acesta este ultimul frame din mesaj.



# Închiderea unui WebSocket

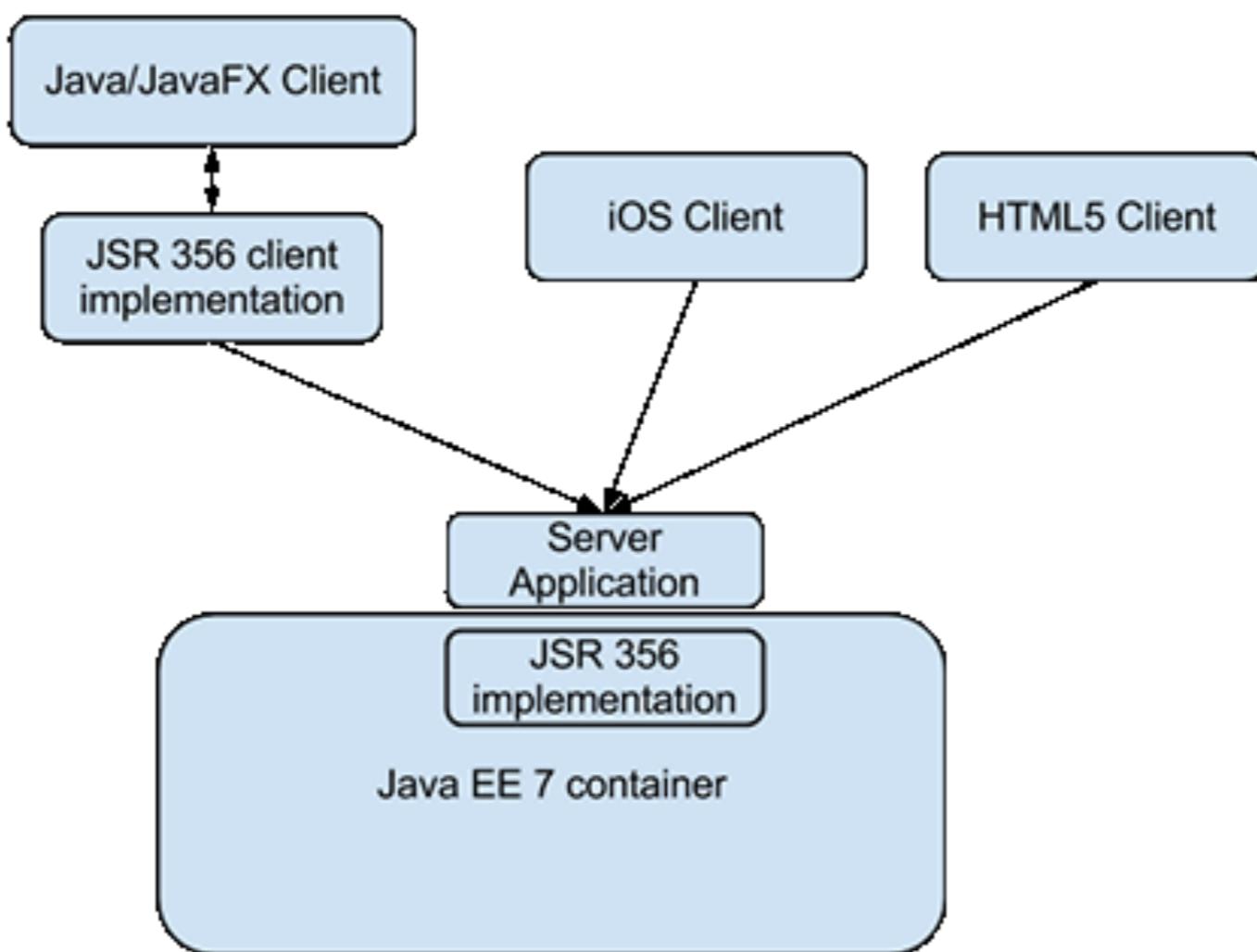
- Fiecare participant la conexiune (clientul sau serverul) poate iniția o cerere de terminare a înțeleggerii (*handshake*).
- Un frame special – un frame *close* – este trimis celuilalt participant. Frame-ul close poate să conțină opțional un motiv pentru terminare și un status code.
- Protocolul definește un set de valori ce pot fi folosite pentru status code.
- Cel care trimitе frame-ul close, nu trebuie să mai trimită alte mesaje după trimiterea acestuia.
- Când celălalt participant primește un frame close, el va răspunde cu propriul frame close. Poate să trimită mesajele netrimise anterior (din buffer) înainte de a trimit frameul close.



# WebSockets API - Java

- **JSR 356**, Java API for WebSocket, specifică API-ul care poate fi folosit de dezvoltatorii Java când vor să integreze/folosească WebSocket-uri în aplicațiile lor, atât pe partea de server cât și pe partea de client Java.
- Fiecare implementare a protocolului WebSocket care pretinde să respecte JSR 356 trebuie să implementeze acest API.
- Dezvoltatorii pot dezvolta aplicațiile lor bazate pe WebSocket independent de implementarea WebSocket.
- JSR 356 face parte din standardul Java EE 7.
- Toate serverele de aplicații care respectă Java EE 7 vor avea/contine o implementare a protocolului WebSocket care respectă standardul JSR 356.

# WebSockets API - Java



- Un client Java poate folosi o bibliotecă client care respectă JSR 356 pentru tratarea problemelor (eng. *issues*) specifice protocolului WebSocket.
- Alți clienți (iOS, HTML5) pot folosi alte implementări (care nu sunt bazate pe Java) dar care respectă specificația RFC 6455 pentru a comunica cu aplicația server.

# WebSockets API - Java

- JSR 356 folosește adnotări și injectare.
- Sunt suportate **două modele de programare**:
  - *Bazate pe adnotări*. Se folosesc clase POJO anotate, cu ajutorul cărora dezvoltatorii pot trata evenimentele ce apar în ciclul de viață al unui WebSocket.
  - *Bazate pe interfețe*. Dezvoltatorii pot trata evenimentele prin implementarea interfeței *Endpoint* care conține metodele corespunzătoare evenimentelor ce pot să apară.
- **Evenimentele din ciclul de viață**:
  - Un client inițiază conexiunea trimînd o cerere de tip HTTP handshake.
  - Serverul răspunde cu un handshake.
  - Conexiunea este stabilită. Din acest moment conexiunea este simetrică, bidirectională.
  - Ambii participanți (clientul și serverul) pot să trimită și să primească mesaje.
  - Unul dintre participanți încide conexiunea.
- Majoritatea evenimentelor din ciclul de viață pot fi mapate la metode Java, indiferent de abordarea folosită (adnotări sau interfețe).

# Java WebSockets API - Adnotări

- Un endpoint care acceptă cereri Websocket (serverul) poate fi orice POJO (Plain Old Java Object) anotat cu `@ServerEndpoint`.
- Această anotare specifică containerului (web) că clasa anotată ar trebui considerată un endpoint WebSocket.
- Elementul adnotării specifică calea către endpoint-ul WebSocket.

```
@ServerEndpoint("/hello")  
public class MyEndpoint { }
```

```
@ServerEndpoint("/hello/{userid}")  
public class MyEndpoint { }
```

- valoarea variabilei `{userid}` poate fi obținută în metodele corespunzătoare ciclului de viață folosind anotarea `@PathParam`.

# Java WebSockets API - Adnotări

- Un endpoint care inițiază o conexiune WebSocket (clientul) poate fi un POJO adnotat cu `@ClientEndpoint`.
- `ClientEndpoint` nu acceptă variabile de cale (ex. {userid}) deoarece nu așteaptă cereri.

`@ClientEndpoint`

```
public class MyClientEndpoint {}
```

- Inițierea unei conexiuni WebSocket, folosind abordarea bazată pe adnotări:

```
javax.websocket.WebSocketContainer container =  
javax.websocket.ContainerProvider.getWebSocketContainer();
```

```
container.connectToServer(MyClientEndpoint.class,  
new URI("ws://localhost:8080/hello"));
```

- Clasele anotate cu `@ServerEndpoint` sau `@ClientEndpoint` sunt numite endpoint-uri anotate.

# Java WebSockets API - Adnotări

- După ce a fost stabilită o conexiune WebSocket se creează un obiect de tip **Session** și metoda adnotată cu **@OnOpen** va fi apelată.
- Metoda poate avea diferiți parametrii:
  - Un parametru de tip **javax.websocket.Session** specificând sesiunea (obiectul Session) creată.
  - Un obiect de tip **EndpointConfig** care conține informații despre configurația endpoint-ului.
  - Zero sau mai mulți parametrii de tip String adnotați cu **@PathParam**, care referă variabilele de cale din calea endpoint-ului(server).

**@OnOpen**

```
public void myOnOpen (Session session) {  
    System.out.println ("WebSocket opened: "+session.getId());  
}
```

# Java WebSockets API - Adnotări

- O instanță de tip `Session` este validă atât timp cât conexiunea WebSocket nu este închisă.
- Clasa `Session` conține metode care permit dezvoltatorilor să obțină mai multe informații despre conexiune.
- Conține metoda `getUserProperties()` care returnează un dicționar `Map<String, Object>` ce păstrează date specifice aplicației.
- Permite dezvoltatorilor să păstreze informații specifice în obiectul de tip `Session`, informații care ar trebui păstrate între diferite apele la metode.

# Java WebSockets API - Adnotări

- Când unul dintre participanți primește un mesaj, este apelată metoda adnotată cu `@OnMessage`.
- Metoda poate avea parametri:
  - Un parametru de tip `javax.websocket.Session`.
  - Zero sau mai mulți parametrii de tip string adnotati cu `@PathParam`, care referă valorile variabilelor de cale (server).
  - Mesajul care poate fi de tip text, binar sau pong.
- Pentru fiecare tip de mesaj este permisă câte o metodă adnotată cu `@OnMessage`. Parametrii permisi pentru specificarea conținutului depind de tipul mesajului primit.

## `@OnMessage`

```
public void myOnMessage (String txt) {  
    System.out.println ("WebSocket received message: "+txt);  
}
```

# Java WebSockets API - Adnotări

- Dacă metoda adnotată cu `@OnMessage` returnează ceva, valoarea returnată va fi transmisă celuilalt participant la conexiune.

`@OnMessage`

```
public String myOnMessage (String txt) {  
    return txt.toUpperCase();  
}
```

- Metodă alternativă de a trimite mesaje folosind o conexiune WebSocket:

```
RemoteEndpoint.Basic other = session.getBasicRemote();  
other.sendText ("Hello, world");
```

- Metoda `getBasicRemote()` din clasa `Session` returnează o reprezentare a celuilalt participant la conexiunea WebSocket, un `RemoteEndpoint`. Acea instanță `RemoteEndpoint` poate fi folosită pentru trimiterea diferitor tipuri de mesaje (text, binar, pong).
- Obiectul de tip `Session` poate fi obținut/păstrat în metodele corespunzătoare ciclului de viață. (ex. metoda adnotată cu `@onOpen`)

# Java WebSockets API - Adnotări

- Metoda `@onclose` este apelată când se închide conexiunea WebSocket.
- Poate avea parametrii:
  - Un obiect de tip `javax.websocket.Session`. Acest parametru nu mai poate fi folosit după ce conexiunea WebSocket este închisă (imediat după terminarea execuției metodei).
  - Un obiect de tip `javax.websocket.CloseReason` care descrie motivul închiderii conexiunii WebSocket (ex. închidere normală, eroare de protocol, serviciu supraîncărcat, etc.).
  - Zero sau mai mulți parametrii de tip String adnotăți cu `@PathParam`, referind variabilele de cale din URL asociat endpoint-ului (server).

## `@OnClose`

```
public void myOnClose (CloseReason reason) {  
    System.out.println ("Closing a WebSocket due to  
"+reason.getReasonPhrase());  
}
```

- În cazul apariției unei erori, va fi apelată metoda adnotată cu `@OnError`.

# Java WebSockets API - Mesaje

- Orice obiect Java poate fi transmis sau primit ca și mesaj WebSocket.
- Trei tipuri diferite de mesaje:
  - *Text*
  - *Binare*
  - *Pong* (specifice conexiunii WebSocket).
- Pentru mesaje de tip *text* sunt permisi următorii parametrii:
  - **String** care reprezintă tot mesajul
  - Un tip primitiv Java sau clasa asociată care reprezintă mesajul convertit la acel tip
  - **String** și o valoarea booleană care reprezintă o parte din mesaj (mesajul poate avea mai multe părți)
  - **Reader** pentru a primi întregul mesaj ca și un stream (blochează execuția până la terminarea citirii de pe stream)
  - orice obiect Java pentru care participantul la conexiune are un *text decoder* (**Decoder.Text** sau **Decoder.TextStream**).

# Java WebSockets API - Mesaje

- Parametrii permisi pentru **mesajele binare**:
  - **byte[]** sau **ByteBuffer** pentru a primi tot mesajul
  - perechea (**byte[]**, **boolean**) sau (**ByteBuffer**, **boolean**) pentru a primi mesajul în părți.
  - **InputStream** pentru a citi tot mesajul dintr-un stream (*blocking*).
  - orice obiect pentru care participantul are un *binary decoder* (**Decoder.Binary** sau **Decoder.BinaryStream**).
- Parametrii permisi pentru **mesaje pong**:
  - **PongMessage** pentru tratarea mesajelor de tip pong.
- Orice obiect Java poate fi convertit într-un mesaj de tip text sau binar folosind un convertor (eng. *encoder*).
- Mesajul în format text/binar va fi transmis celuilalt participant, unde va fi reconvertis (eng. *decoded*) într-un obiect Java.
- Pentru transmiterea mesajelor WebSocket se folosește adesea formatul **XML** sau **JSON**. În acest caz convertirea/reconvertirea se reduce la transformarea unui obiect Java într-un XML/JSON și invers.

# Java WebSockets API - Mesaje

- Un convertor este o clasă care implementează interfața `javax.websocket.Encoder`, iar un reconvertor este o clasă care implementează interfața `javax.websocket.Decoder`.
- Participanții la conexiunea Websocket trebuie să știe posibilitățile convertorilor/reconvertorilor.
- Lista convertorilor/reconvertorilor poate fi transmisă ca și elemente ale adnotărilor `@ClientEndpoint` și `@ServerEndpoint`.

```
@ServerEndpoint(value="/endpoint", encoders = MessageEncoder.class,
decoders= MessageDecoder.class)

public class MyEndpoint {

...
}
```

# Java WebSockets API - Messages

```
class MessageEncoder implements Encoder.Text<MyJavaObject> {  
    @Override  
    public String encode (MyJavaObject obj) throws EncodingException {  
        ...  
    }  
}  
  
class MessageDecoder implements Decoder.Text<MyJavaObject> {  
    @Override  
    public MyJavaObject decode (String src) throws DecodeException {  
        ...  
    }  
  
    @Override  
    public boolean willDecode (String src) {  
        // return true if we want to decode this String into  
        // a MyJavaObject instance  
    }  
}
```

# Java WebSockets API - Messages

- Interfață **Encoder** are următoarele subinterfețe:
  - **Encoder.Text** - convertirea obiectelor Java în mesaje de tip text
  - **Encoder.TextStream** - adăugarea obiectelor Java la un *character stream*
  - **Encoder.Binary** - convertirea obiectelor Java în mesaje binare
  - **Encoder.BinaryStream** - adăugarea obiectelor Java la un *binary stream*
- Interfață **Decoder** are 4 subinterfețe:
  - **Decoder.Text** - convertirea mesajelor text într-un obiect Java
  - **Decoder.TextStream** - citirea unui obiect Java dintr-un *character stream*
  - **Decoder.Binary** - convertirea unei mesaj binar într-un obiect Java
  - **Decoder.BinaryStream** - citirea unui obiect Java dintr-un *binary stream*

# Abordarea bazată pe interfețe

- Clasa `javax.websocket.Endpoint`:

- se redefinesc metodele `onOpen`, `onClose`, și `onError`:

```
public class MyOwnEndpoint extends javax.websocket.Endpoint {  
    public void onOpen(Session session, EndpointConfig config) {...}  
    public void onClose(Session session, CloseReason closeReason) {...}  
    public void onError (Session session, Throwable throwable) {...}  
}
```

- Pentru interceptarea mesajelor trebuie înregistrat un obiect de tip `javax.websocket.MessageHandler` în definirea metodei `onOpen`:

```
public void onOpen (Session session, EndpointConfig config) {  
    session.addMessageHandler (new MessageHandler() {...});  
}
```

# Abordarea bazată pe interfețe

- Interfața **MessageHandler** are două subinterfețe:
  - **MessageHandler.Partial** - apariția unui mesaj parțial.
  - **MessageHandler.Whole** - apariția unui mesaj complet.

```
public void onOpen (Session session, EndpointConfig config) {  
    final RemoteEndpoint.Basic remote = session.getBasicRemote ();  
    session.addMessageHandler (new MessageHandler.Whole<String> () {  
        public void onMessage (String text) {  
            try {  
                remote.sendString (text.toUpperCase ()) ;  
            } catch (IOException ioe) {  
                // handle send failure here  
            }  
        }  
    }) ;  
}
```

# WebSockets

- Exemple:
  - Broadcast
  - Whiteboard
- Referințe
  - RFC 6455
  - <https://tools.ietf.org/html/rfc6455>
  - Johan Vos, *JSR 356, Java API for WebSocket*,
  - <http://www.oracle.com/technetwork/articles/java/jsr356-1937161.html>
  - Brian Raymor, *WebSockets: Stable and Ready for Developers*,
  - <https://msdn.microsoft.com/en-us/hh969243.aspx>

# Autentificare REST

- *Autentificarea* (eng. *Authentication*): verificarea credențialelor trimise de client (utilizator, parolă)
- *Autorizarea* (eng. *Authorization*): verificarea dreptului de a accesa anumite resurse
- Antetul HTTP *Authorization*
- Constrângerea REST *stateless*: la fiecare cerere HTTP trebuie transmise informațiile necesare
- Abordări:
  - HTTP Authentication (RFC 2617)
    - Basic Authentication
    - Digest Authentication
  - OAuth 2.0

# Autentificare REST

- **HTTP Basic Authentication**

- Folosirea antetului HTTP Authorization în care se transmit username-ul și parola codificate în base64 la fiecare cerere:

**GET / HTTP/1.1**

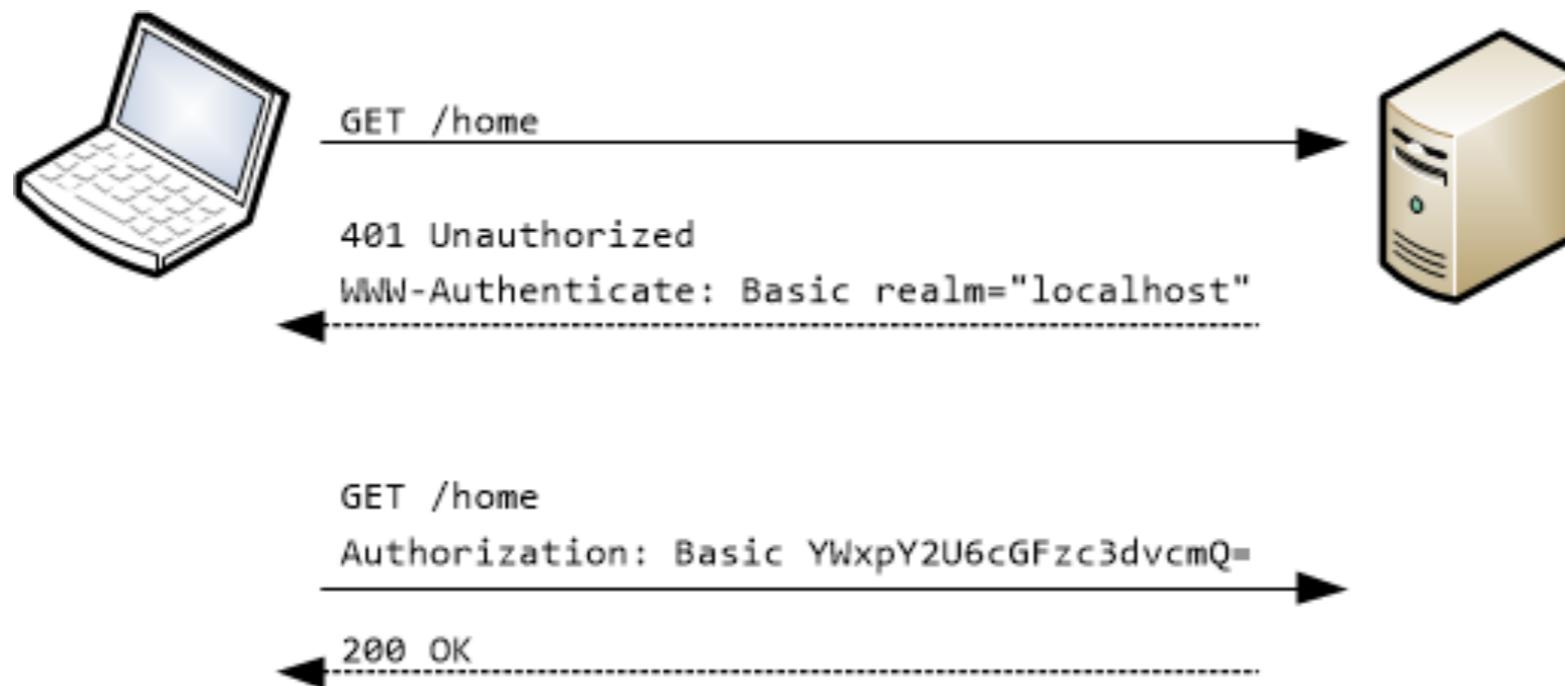
**Host: example.org**

**Authorization: Basic Zm9vOmJhcg==**

- Credențialele sunt codificate, dar **nu sunt criptate!**

- Informațiile pot fi obținute ușor.

- Se recomandă folosirea doar cu SSL/TLS



# Autentificare REST

## • **HTTP Digest Authentication**

- Folosirea antetului HTTP Authorization în care se transmit username-ul, parola și alte informații codificate.
- HMAC (hash based message authentication)
- Se transmite o versiune criptată a parolei
- Se codifică și alte informații despre resursa dorită

Exemplu: utilizator “**ion**”, parola “**ionpass**”,

**GET /users/ion/account**

```
digest = base64encode(hmac("sha256", "ionpass",
                            "GET+/users/ion/account"))
```

**GET /users/ion/account HTTP/1.1**

**Host: example.org**

**Authorization: hmac ion: [digest]**

- Parola nu poate fi criptată pe server, pentru a putea reconstrui digest
- Se recomandă folosirea unui “secret” cunoscut de client și server

# Autentificare REST

## • HTTP Digest Authentication

- Avantaj: un hacker nu poate modifica cererea (ar trebui să modifice valoarea *digest* și nu cunoaște “secret”-ul)
- Dezavantaj: un hacker poate executa cererea de oricâte ori
  - Soluție: se transmit mai multe informații în *digest*:
    - Data curentă
    - Nonce (Number used *once*). La cereri subsecvente valoarea nonce este diferită.

```
digest = base64encode(hmac("sha256", "secret",
"GET+users/ion/account+30may201912:59:24+123456"))
```

GET /users/ion/account HTTP/1.1

Host: example.org

Authorization: hmac ion:123456: [digest]

Date: 30 may 2019 12:59:24

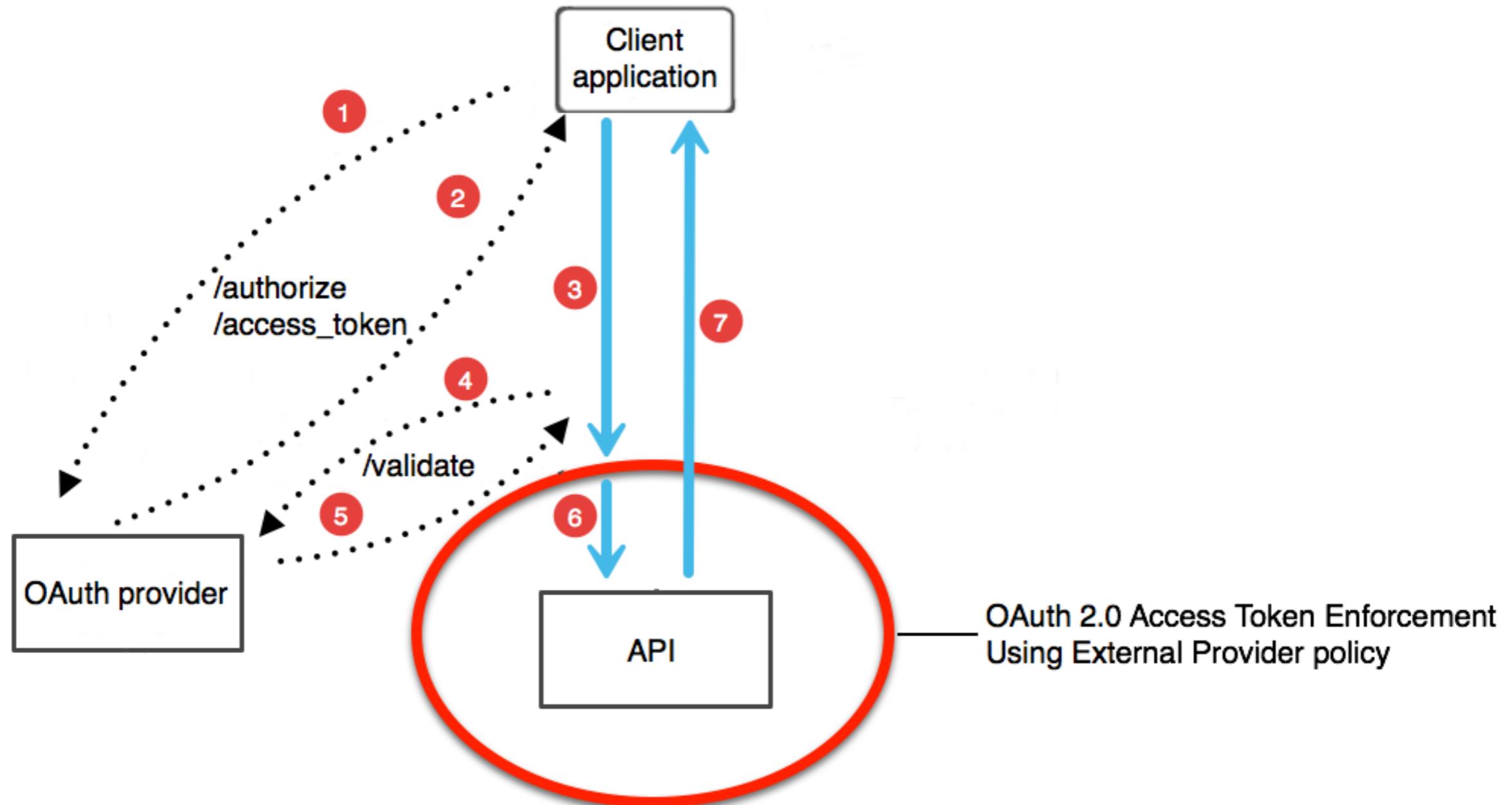
# Autentificare REST

- **OAuth**

- OAuth 1.0 decembrie 2007
  - Sigur, dar dificil de implementat (criptografie, interoperabilitate)
- OAuth 2.0 octombrie 2012
  - Mai ușor de implementat
  - Compromisuri la nivel de securitate
  - Cel mai folosit
- **Flux:**
  - Aplicația client se înregistrează la un **furnizor**
  - Furnizorul transmite clientului un “secret” unic
  - Clientul include “secret”-ul la fiecare cerere
  - Dacă cererea nu este formată corect, are date lipsă sau nu conține secretul corect, ea va fi respinsă.

# Autenticare REST - OAuth

## OAuth Dance



# JSON Web Token (JWT)

- **JSON Web Token** este un standard (RFC 7519) care definește un mod *compact* și *independent* de a transmite informații între participanți ca și un object JSON.
  - *Compact*: are dimensiuni reduse.
    - poate fi transmis prin URL, parametrii POST, antet HTTP
    - este transmis rapid
  - *Independent*: un token JWT conține toată informația necesară despre o entitate pentru a evita interogări multiple ale bazei de date.
    - cine primește tokenul nu trebuie să facă alte cereri la server pentru a-l valida.
- Informația conținută într-un token JWT poate fi verificată și se poate avea încredere în ea pentru că este *semnată digital*.
- JWT poate fi semnat folosind:
  - un *secret* (folosind HMAC)
  - cheie publică/privată folosind RSA, ECDSA, etc

# JWT

- Aplicabilitate JWT
  - Autentificare/Autorizare
  - Schimb de informații între participanți (client/server)
    - Securizat
- **Structura unui token JWT**
  - 3 părți separate prin '.': **aaa . bbb . ccc**
  - Header (**aaa**)
  - Payload (**bbb**)
  - Semnătura digitală (**ccc**)

# Header JWT

- Conține informații despre tipul de token și algoritmii de criptare folosiți pentru conținut.
- Are **două părți**:
  - tipul tokenului (JWT)
  - algoritmul de criptare (ex. HMAC SHA256, RSA)

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

# Payload JWT

- Conține informații despre entitate ce pot fi verificate (identitatea utilizatorului, permisiunile, etc), eng. *claims*
- 7 *claims* rezervate și recomandate (dar nu obligatorii):
  - *iss* (issuer), *sub* (subject) -utilizatorul, *aud* (audience) - pentru cine a fost creat, *exp* (expiration time), *nbf* (not before time), *iat* (issued at time), *jti* (JWT ID) -identificator unic, poate fi folosit o singură dată

```
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "admin": true  
}
```

- Poate conține și alte informații adiționale (eng. *custom claims*). Poate fi folosit orice nume, care nu este rezervat prin specificație.

# Semnătura digitală JWT

- Este folosită pentru a verifica expeditorul și pentru a verifica dacă mesajul a fost modificat pe parcurs.
- Se codifică Base64 header-ul și payload-ul și se cripteză folosind algoritmul specificat în header (împreună cu secretul/cheie publică-privată)

**HMACSHA256 (**

```
base64UrlEncode(header) + "." +  
base64UrlEncode(payload) ,  
secret)
```

- **Înainte de folosirea unui token JWT trebuie verificată semnătura lui.**

# JWT.io

ALGORITHM HS256 ▾

## Encoded

PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ik1hcmluZXNjdSBWYXNpbGUiLCJpYXQiOjE1MTYyMzkwMjJ9.J3H8MqVValMLWo1ycE0UBNax3g1XeURR2XwU5Z9F9Zc
```

## Decoded

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

PAYOUT: DATA

```
{  
  "sub": "1234567890",  
  "name": "Marinescu Vasile",  
  "iat": 1516239022  
}
```

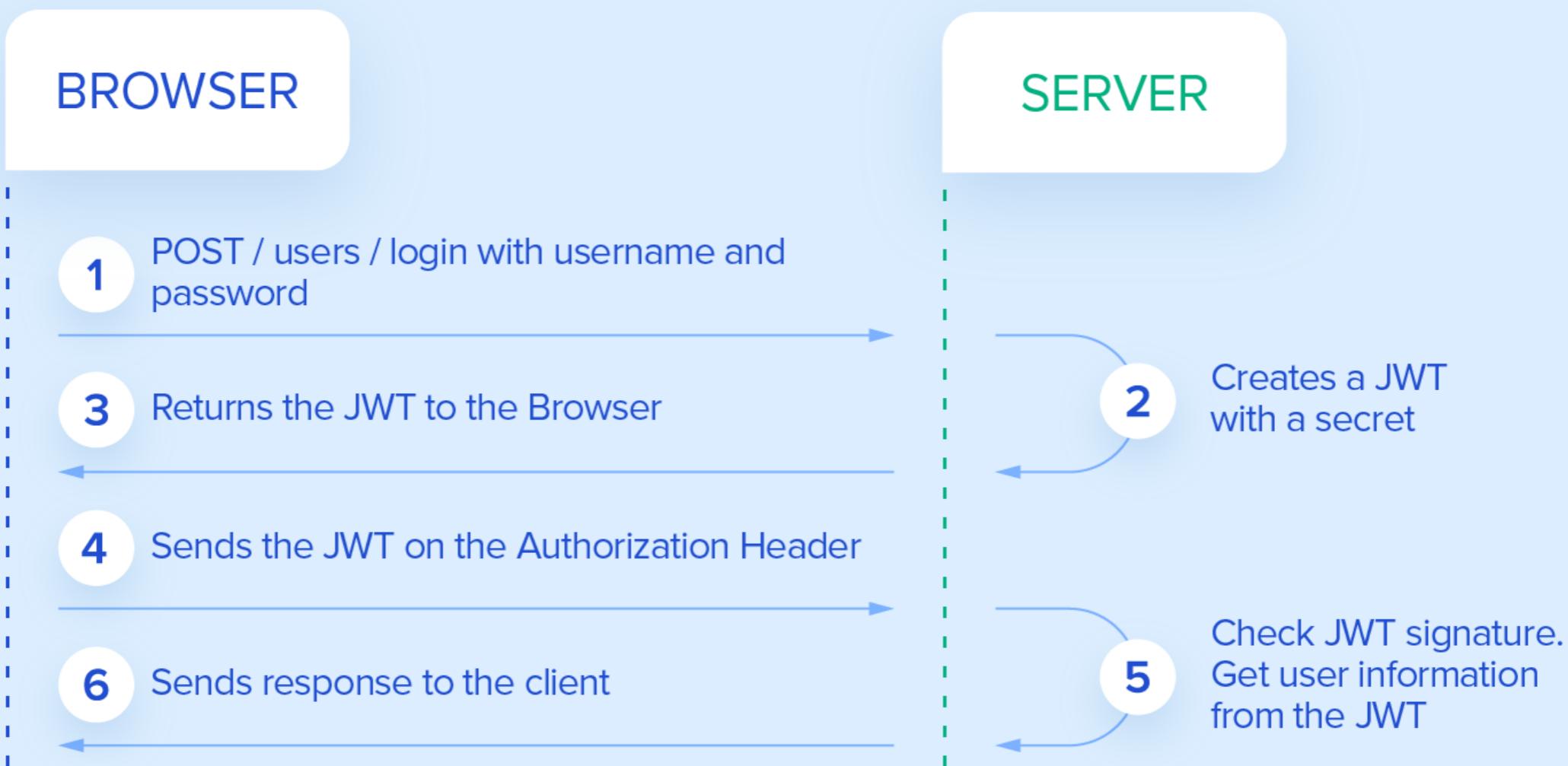
VERIFY SIGNATURE

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  secretul meu  
)  secret base64 encoded
```

✓ Signature Verified

SHARE JWT

# JWT



# Referințe

- RFC 7235 - Access Authentication Framework  
<https://tools.ietf.org/html/rfc7235#section-2>
- RFC 2617 - HTTP Authentication: Basic and Digest Access Authentication  
<https://tools.ietf.org/html/rfc2617>
- RFC 6749 - OAuth2 standard  
<https://tools.ietf.org/html/rfc6749>
- Guy Levin, *RESTful API Authentication Basics*,  
<https://blog.restcase.com/restful-api-authentication-basics/>
- Dejan Milosevic, *REST Security with JWT using Java and Spring Security*  
<https://www.toptal.com/java/rest-security-with-jwt-spring-security-and-java>
- Bruno Krebs, *Implementing JWT Authentication on Spring Boot APIs*  
<https://auth0.com/blog/implementing-jwt-authentication-on-spring-boot/>

# Examen

- **Calcul media finală:**

- **10%** Teme lab (4 teme)- TL
- **30%** Media lab - Lab ( $\text{Lab} \geq 4.5$  - examen in sesiunea normala)
- **10%** Quizzuri - MQ=(Q1+Q2)/2
- **50%** Nota examen, ( $\text{NE} \geq 4.5$ )

**Media finală=0.1\*TL+0.3\*Lab+0.5\*NE+MQ**

**Promovat: Media finală  $\geq 4.5$ !**

# Examen - Structura

- **2 întrebări teoretice - 15 min - 2p**
- **Dezvoltarea unei aplicații client server - 2 h - 8p**
  - Puteți reutiliza codul sursă (teme laborator, alte exemple, etc).
  - Puteți folosi resurse de pe Internet (pentru erori, alte probleme).
  - **NU AVEȚI VOIE SĂ DISCUTAȚI CU ALTE PERSOANE (viu grai, chat, telefon, email, social media, etc)**
  - **Denumirile interfețelor, claselor, atributelor, etc. trebuie modificate conform enunțului problemei!**
  - **Conțează arhitectura și proiectarea (interfete)!**
  - **Jurnalizare, fisiere de configurare, conectare la baza de date, ORM**
  - **2 servicii REST (respectarea regulilor de formare a URL-ului)**
  - **Observer distribuit (notificare)**
  - **Pentru a fi evaluată, soluția dezvoltată trebuie să compileze!!!**
  - **Se punctează doar funcționalitățile care se execută!**