CODESPRING
OUR CODE YOUR SUCCESS
20 years

Codespring mentorprogram
Enterprise szoftverfejlesztési módszerek (MP II)

Let's win the race together!

Automatizált build és függőségkezelés

Gradle és Maven

Simon Károly, Sulyok Csaba
simon.karoly@codespring.ro, sulyok.csaba@codespring.ro

1. rész

Bevezető

Build rendszerek szükségessége

- a build parancsok legtöbb programozási nyelvnél repetitívek, hibaérzékenyek
- `src/hello/HelloWorld.java`:

```
package hello;
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

- futtatható JAR állomány készítése és futtatása:

```
# compile
javac -sourcepath src -d build/classes src/hello/HelloWorld.java
# próba: class állomány futtatása
java -cp build/classes hello.HelloWorld
# MANIFEST írása
echo "Main-Class: hello.HelloWorld" > MANIFEST.mf
# JAR állomány készítése
jar cvfm build/jar/HelloWorld.jar MANIFEST.mf -C build/classes/ .
# JAR állomány futtatása
java -jar build/jar/HelloWorld.jar
```

Build rendszerek szükségessége

- ezen folyamatok automatizálása már elérhető a Make eszköz óta
- a meghívható parancsok valamilyen **leíró (deszkriptor) állományban** találhatóak
  - Make esetén `Makefile`
- a leíró állomány verziókövetőre feltölthető, így egy projekten dolgozó összes csapattag garantáltan ugyanazt a folyamatot hívja
- egy projekt gyökerébe kerül, általában elkülönítve a forráskódtól
- további taszkok, melyek repetitívek, s segíthet az automatizálásuk:
  - futtatás
  - tesztek futtatása
  - kitelepítés külső szerverre

Függőségkezelés

CODESPRING
OUR CODE YOUR SUCCESS
20 years

- vállalati szintű rendszerek esetén legtöbbször nem elegendő a nyelvbe beépített függvények sokasága
- külső (*third-party*) könyvtárak és keretrendszerek beépítése szükséges
- kompilálás és más aktivitások automatizálása (linkelés, összecsomagolás, tesztelés, futtatás konzolról, stb.)
- pontos verziók leszedése minden alkalommal
- **nem** akarjuk verziókövetőn tárolni a függőségeket
- ugyanazon **deszkriptorban** rögzítjük, hogy melyek a függőségek (ez az állomány bekerül verziókövetés alá)
- minden nagyobb nyelv rendelkezik (legalább egy) ilyen eszközzel

Build és függőségkezelési eszközök

CODESPRING
OUR CODE YOUR SUCCESS
20 years

- Legtöbb programozási nyelv nyújt egyet, pl.:
  - Java: Maven és Gradle
  - node.js: npm, yarn
  - Python: pip
  - .NET/C#: NuGet
- Egyes eszközök nem nyújtanak függőségkezelést, csak build automatizálást (pl. Make, Ant)
- Nem kompilált nyelvek esetén gyakran a függőségkezelési aspektus kerül előtérbe (pl. npm)
- Kompilált nyelvek esetén a modern eszközök mindkét feltételnek eleget tesznek (pl. Maven, Gradle)

2. rész

Groovy, a gradle szkriptek nyelve

Groovy

- A Groovy egy Java alapú interpretált szkriptnyelv, amely egyszerűbben tanulható nyelvekből inspirálódik (pl. Python)
- Standard Java kód direkten értelmezhető Groovy-ban, de egyszerűsíthető
- Egyszerűsítések:
  - Nem szükséges egy `main` függvény, egy szkript belsejében levő kód enkapszulálódik egy `main` függvénybe.
  - Elhagyhatóak a pontosvesszők.
  - A `System.out` stream metódusai prefix nélkül meghívhatóak.
  - Függvényhívások zárójelei hanyagolhatóak.
  - Getter és setter metódusok kezelhetőek mint adattag.
  - A `return` kulcsszó mellőzhető metódusok utolsó sorában.
  - Szimpla idézőjelek használhatóak, dupla idézőjelekbe bash-stílusú változóneveket tehetünk.
  - Nem erősen típusos (`def` szócskával adhatunk meg változókat).
  - `List` és `Map` példányok megadhatóak inline zárójelekkel.

Groovy összehasonlítás

CODESPRING
OUR CODE YOUR SUCCESS
20 years

`test.groovy` – Java-kompatibilis kód

```
public class MyClass {

    private String myAttribute;

    public MyClass(String myAttribute) {
       this.myAttribute = myAttribute;
    }

    public String getMyAttribute() {
        System.out.println("Retrieving attribute");
        return myAttribute;
    }

    public static void main(String[] args) {
        System.out.println("Hello, World!");
        MyClass myObject = new MyClass("myAttributeValue");
        System.out.println("Retrieved attribute is " + myObject.getMyAttribute());
    }
}
```

Groovy összehasonlítás

CODESPRING
OUR CODE YOUR SUCCESS
20 years

`test.groovy` – Groovy egyszerűsített kód

```
class MyClass {

    def myAttribute

    def getMyAttribute() {
        println "Retrieving attribute"
        myAttribute
    }
}

println 'Hello, World!'
def myObject = new MyClass(myAttribute: "myAttributeValue");
println "Retrieved attribute is ${myObject.myAttribute}"
```

Groovy closures

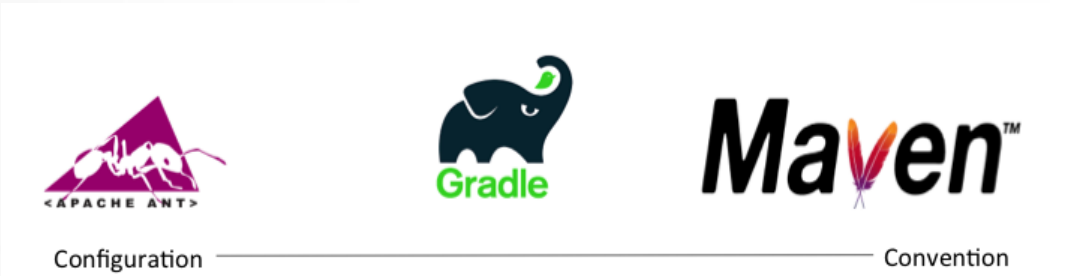
- **Closure** = *lambda* kifejezés = egy névtelen kódblokk, mely különböző kontextusokban újrahasználható
- Fogadhat tetszőleges számú paramétert

```
// defining closures
def c1 = { println "hello from closure" }
def c2 = { param1 ->
    println "hello from closure, param1 = ${param1}"
}

// executing closures
c1(); c2(42)

// using closures as arguments
println "iterating through 1 to 10"
(1..10).forEach { it -> println it }
```

3. rész

Gradle (és Maven)

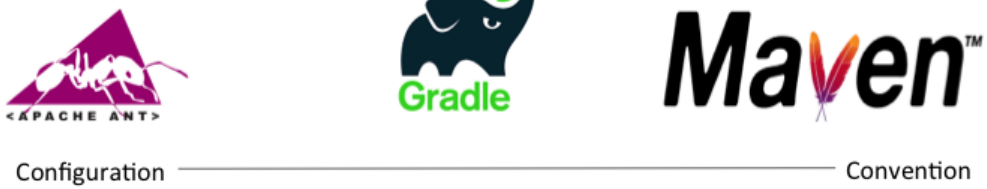
Gradle (és Maven)



Mindkettő

- Automatizált build, függőség- és projektmenedzsment eszközök
- Nyílt forráskódú
- Több projektet menedzselhetünk (multimodul) egyszerre
- Konvenciót diktál folderek és állományok elhelyezésére, nevére és strukturájára
- Számos külső kiegészítő plugin

Gradle (és Maven)



Gradle

- `https://github.com/gradle/gradle`
- deszkriptor: *build.gradle* (Groovy nyelv) vagy *build.gradle.kts* (Kotlin)
- célpont mappa: *build*
- kiterjeszthető
- nincs saját függőségi rendszere, támogatja a Maven és Ivy-t
- sok másik programozási nyelvhez nyújt plugint, illetve elterjedt az Android fejlesztők körében

Maven

- `https://github.com/apache/maven`
- deszkriptor: *pom.xml* (XML formátumú)
- célpont mappa: *target*
- convention over configuration – rigid szabványok, nehezen terjeszthetőek, de legtöbb helyzetre megoldást nyújtanak
- saját függőségi rendszer
- kifejezetten Java projektekhez

Gradle taskok

- A `gradle` egy command-line eszköz, amely taskok futtatását teszi lehetővé.
- `gradle tasks` – egy task mely kiírja a lehetséges taskokat (a `tasks` is egy task)
- `gradle <taskName>` - lefuttatja az adott nevű taskot.
- A taskot a `task` kulcsszóval adjuk hozzá a projekthez, s a benne definiált `doLast` kulcsszóval adjuk meg a teendőjét.
- A taskok között függőségek lehetnek (`dependsOn` kulcsszóval).
- A taskok leírását két kulcsszón keresztül állítjuk be: `group` és `description`.

Gyakorlat

Készítsünk egy Gradle projektet amely 2 taskot tartalmaz: `"a"` és `"b"`. Mindkettő fő teendője, hogy írjon egy-egy üzenetet a konzolra. A `"b"` task függjön az `"a"` tasktól. Mindkettőnek értelmezzük a kategóriáját és leírását. Hívjuk meg a `"b"` taskot, illetve listázzuk a lehetséges taskokat.

Példa: Gradle taskok

CODESPRING
OUR CODE YOUR SUCCESS
20 years

```
description = '''
    Példaprojekt Gradle taszkokkal
'''

// létrehozunk egy "a" nevű taszkot
task a {
    group 'Example' // a taszk kategóriája
    description 'Example task' // leírása

    doLast { // a taszk tényleges kódja
        println 'Hello from task a'
    }
}

// alternatív írásmód taszk létrehozására
task('b', group: 'Example', description: 'Another example task',
        dependsOn: ['a']) {
    doLast {
        println 'Hello from task b'
    }
}

// a b taszkot állítjuk alapértelmezettnek
// gradle meghívása paraméterek nélkül egyenértékű a "gradle b"-vel
defaultTasks 'b'
```

Gradle pluginok

- Egy plugin létrehoz új taszkokat, megjelöl forráskódszetteket, vagy elvégez más projektkonfigurációt.
- Példa: **`java`**, **`cpp`**, **`npm`**, **`application`**, **`maven-publish`**, etc.
- Hivatalosan támogatott pluginok:
  `https://docs.gradle.org/current/userguide/standard_plugins.html`
- Plugin érvénybe léptetése:
  - az **`apply`** metódussal:

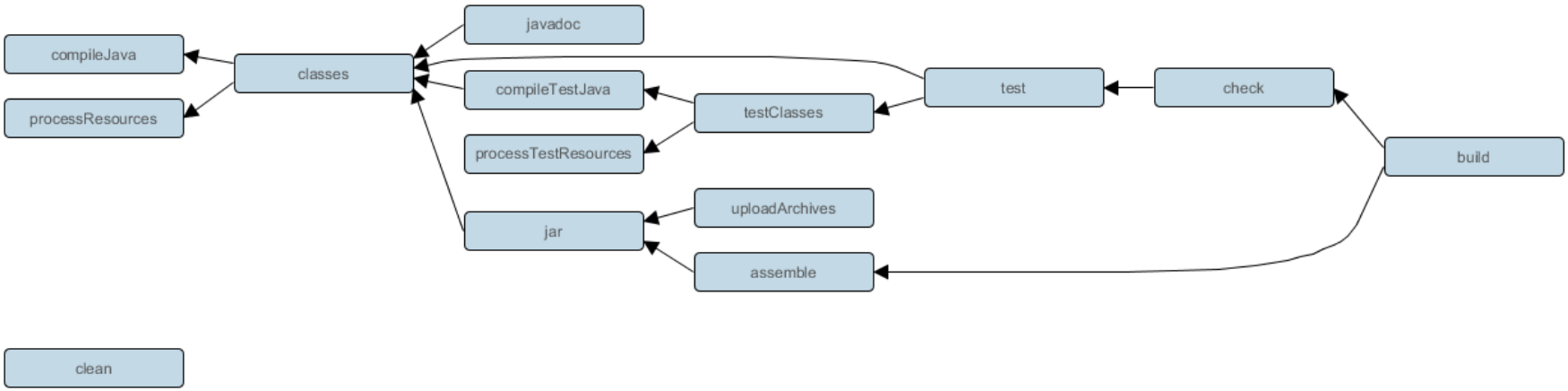
```
apply plugin: 'java'
```

  - a **`plugins`** closure segítségével (újabb, megadható verzió is a pluginoknak):

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

Példa: gradle `java` plugin

- Megjelöli a Maven konvenciók által diktált forráskódmappákat (`src/main/java`, `src/main/resources`, `src/test/java`, `src/test/resources`)
- Készít taszkokat buildelésre, tesztelésre, takarításra, stb., amelyek között jól követhető függőségi gráfot alakít ki (pl. a `jar` csomagoló task függ a `classes`-től, amely burkolja a forráskódok kompilálását – `compileJava` – és erőforrások másolását – `processResources`)
- Listázhatjuk a meghívott taskokat a `gradle --console=verbose taskname` flaggel

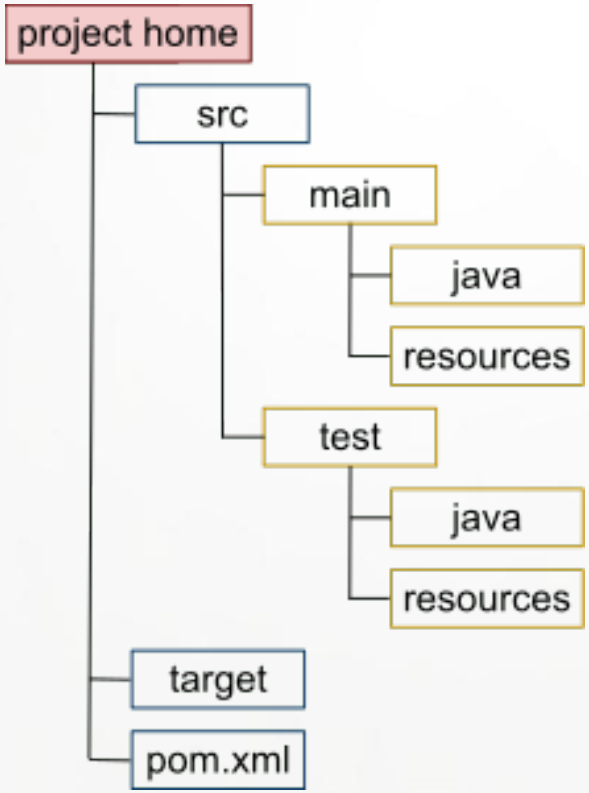


Példa: gradle **`application`** plugin

- **`application`** plugin: létrehoz taszkokat, amelyekkel:
  - **`run`** - futtathatjuk a projektet egy **`main`** metódust tartalmazó osztály megadásával
  - **`installDist`**, **`distZip`** - összecsomagolhatjuk a projektet s az összes függőséget egy kiadható **`distributable`**-be
- további információ

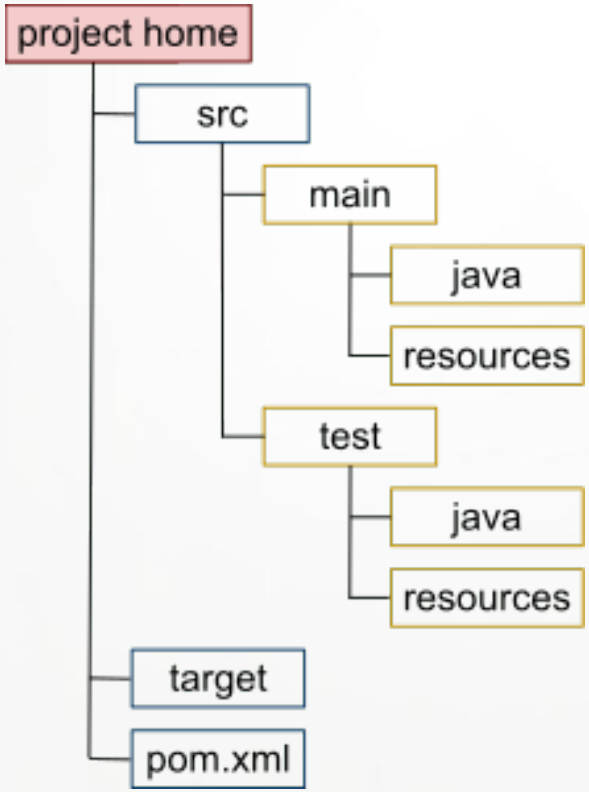
```
plugins {
    id 'java'
    id 'application'
}

application {
    // a main metódust tartalmazó osztály megjelölése
    mainClass = 'edu.codespring.gradleex.application.HelloWorld'
}
```

Konvencionális projektstruktúra



- konvencionális projektstruktúra
  - `build.gradle` (Gradle esetén) vagy `pom.xml` (Maven esetén) - jelen projekt telepítés-deszkriptora;
  - `build` (Gradle esetén) vagy `target` (Maven esetén) - minden build eredmény, köztes állományok, reportok, stb.; **nem** kerül be verzió-követés alá;
  - `src/main/java` - kompilálandó forráskód; bekerül a végső termékbe;
  - `src/main/resources` - extra erőforrások (konfigurációs fájlok, képek, stb.); nem kompiláljuk; bekerül a végső termékbe;
  - `src/test/java` - unit tesztek forráskódjai; lefutnak buildeléskor, de nem kerülnek be a végső termékbe;
  - `src/test/resources` - unit teszteknél használatos erőforrások; nem kerülnek be a végső termékbe;

Konvencionális projektstruktúra

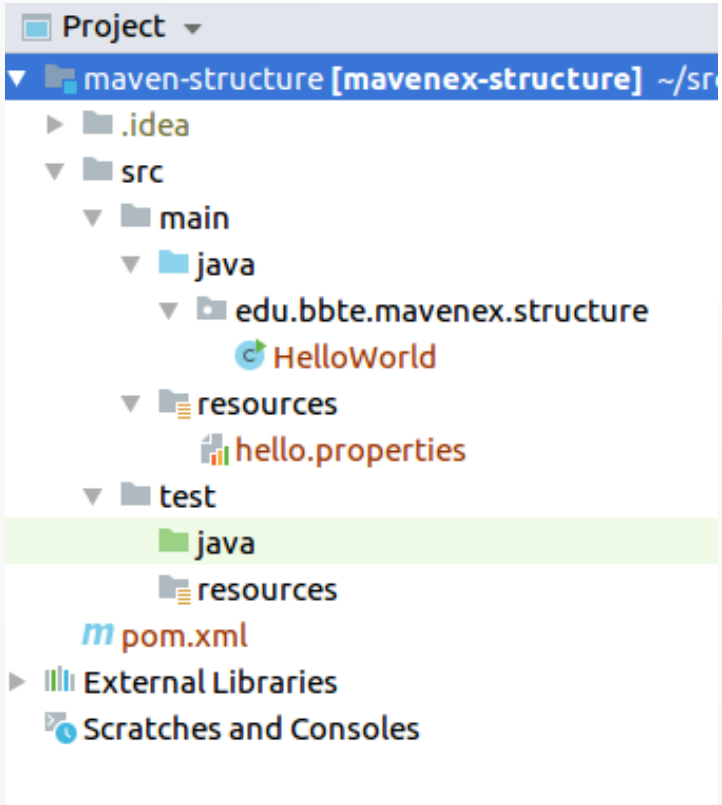


- a `src/test/java`-n belüli osztályok látják a `src/main/java`-n belüli osztályokat, illetve a `src/main/resources`-en belüli erőforrásokat
- fordítva nem érvényes: a `src/main/java`-n belüli osztályok **nem** látják a `src/test/java`-n belüli osztályokat, illetve a `src/test/resources`-en belüli erőforrásokat

Konvencionális csomagstruktúra

- Java forráskódmappák struktúrájára vonatkozó konvenciók (pl. `src/main/java`-n belül)
- mélyebb csomaghierarchia alkalmazott $\Rightarrow$ garantált, hogy nincs átfedés a függőségekben megjelenő osztálynevekkel
- egyezik a hálózati domainnevek hierarchiájában, de a domainnevek fordított sorrendben alkalmazzák ezt
- a csomagok sorrendben tartalmazzák a következő információt:
  1. a projekt típusa
     - kereskedelmi (commercial–`com`)
     - tanítási jellegű (educational–`edu`)
     - non-profit (organizational–`org`)
  2. a tulajdonos intézmény neve (cég vagy egyetem neve)
  3. a projekt neve
  4. további finomított szétbontás a projekten belül, általában osztályok célja szerint (több részlet később)
- példák teljes osztálynevekre:
  - `edu.codespring.blog.views.BlogPostView`
  - `com.google.calendar.controllers.MainController`

Maven projekt példa



HelloWorld.java:

```java
package edu.codespring.mavenex.structure;

import java.util.Properties;
import java.io.InputStream;
import java.io.IOException;

/**
 * példakód egyszerű állományolvasásra classpath-ből
 */
public class HelloWorld {
    public static void main(String[] args) throws IOException {
        // nyitunk egy streamet a classpathből
        // nem feltétlenül egyezik a lokális
        // fájlrendszer tartalmával
        InputStream propsStream = HelloWorld
            .class.getResourceAsStream("/hello.properties");

        // betöltjük a kulcs-érték párokat
        Properties props = new Properties();
        props.load(propsStream);

        // kulcs szerinti olvasás
        System.out.println("Hello " + props.get("name"));
    }
}
```

Gradle függőségek

- **függőség**: jelen modultól független modul, amelyben definiált osztályokra szükségünk van a folyamat valamely szakaszában (pl. kompiláláskor, futtatáskor, teszteléskor)
- a Gradle nem nyújt saját függőségi mechanizmust, csak támogatja a Maven vagy Ivy stílusú tárolókból való behúzást
- függőségek használatához szükséges a `java` plugin aktiválása
- Maven modulokra történő függőségek esetén szükséges ismerni a csomag Maven ID-ját, melynek komponensei:
  - `groupId` - alap csoport csomagnév–konvencionálisan egyezik a korábban említett csomaghierarchia első tagjaival
  - `artifactId` - a projekt/almodul neve
  - `version` - a modul verziószáma
  - `packaging` (*opcionális*) - a modul végtermékének típusa, alapértelmezetten `jar` (alternatív értékek: `zip`, `war`, `ear`, stb.)
  - `classifier` (*opcionális*) - ha több végterméket készítünk, ez az utolsó string különbséget tesz közöttük

Függőségek és tárolók

CODESPRING
OUR CODE YOUR SUCCESS
20 years

- a függőségeknek megfelelő csomagokat a Gradle és Maven először egy lokális tárolóban keresi (alapértelmezett a home folderen belül), majd a deklarált külső repokban
- a **repositories** szekcióban deklarálunk Maven vagy Ivy stílusú tárolókat, ahonnan függőségeket kereshetünk – pár gyorsan használható metódus:
  - `mavenCentral()` => `https://repo.maven.apache.org/maven2/`
  - `google()` => `https://maven.google.com/`
- vizuális keresés a Central repóban: `https://mvnrepository.com/` vagy `https://search.maven.org/`
- első letöltés után a csomagok cache-elődnek a lokális tárolóban ⇒ első használat lassú lehet
- más tárolókat is megadhatunk URL alapján:

```
repositories {
    mavenCentral()
    maven {
        url "http://jcenter.bintray.com/"
    }
}
```

Függőség hatókörök

- A projektéletciklus melyik fázisában van szükség egy-egy függőségre?
- A Gradle `java` pluginja külön `configuration`-eket definiál aszerint, hogy mikor van szükségünk függőségekre. Ez a lista kiterjeszthető.
- Fontosabb értékek:
  - *implementation* - mind kompiláláskor, mind futáskor szükséges, s mindkét source setben
  - *runtimeOnly* - kompiláláskor nem szükséges, de futáskor igen
  - *compileOnly* - kompiláláskor szükséges, de futáskor elvárjuk egy külső konténertől, hogy szolgáltassa a függőséget (pl. webkonténer, EE szerver)
  - *testImplementation*, *testCompileOnly*, *testRuntimeOnly* - csak teszteléskor szükséges (csak a `src/test/java`-beli osztályok látják)

Gradle függőségek

CODESPRING
OUR CODE YOUR SUCCESS
20 years

- a **dependencies** szekcióban deklaráljuk a függőségeket
- két lehetséges szintaxis:
  - single String, "**:**" karakter elválasztással:

```
configuration 'group:artifact:version'
```

  - kulcs-érték párosok használata

```
configuration group: 'group', name: 'name', version: 'version'
```

Gradle függőségek példa


```
plugins {
    id 'java'
}

repositories {
    // define Maven Central as main repository
    mavenCentral()
}

dependencies {
    // API for logging necessary for compilation
    implementation group: 'org.slf4j', name: 'slf4j-api', version: '1.7.25'
    // implementation for SLF4J logging only necessary when running the application
    runtimeOnly group: 'ch.qos.logback', name:'logback-classic', version:'1.2.3'
    // JUnit only necessary in testing
    testImplementation group: 'junit', name: 'junit', version: '4.12'
}
```

Gradle függőségek példa

CODESPRING
OUR CODE YOUR SUCCESS
20 years

- függőségek vizsgálata a gradle **`dependencies`** taszkkal:

```
$ gradle dependencies

runtimeClasspath - Runtime classpath of source set 'main'.
+--- org.slf4j:slf4j-api:1.7.25
\--- ch.qos.logback:logback-classic:1.2.3
    +--- ch.qos.logback:logback-core:1.2.3
    \--- org.slf4j:slf4j-api:1.7.25

testRuntimeClasspath - Runtime classpath of source set 'test'.
+--- org.slf4j:slf4j-api:1.7.25
+--- ch.qos.logback:logback-classic:1.2.3
|    +--- ch.qos.logback:logback-core:1.2.3
|    \--- org.slf4j:slf4j-api:1.7.25
\--- junit:junit:4.12
    \--- org.hamcrest:hamcrest-core:1.3
```

Példa: függőségek Gradle-ben

src/main/java/.../HelloWorld.java:

```
public class HelloWorld {

    // logger betöltése
    // ez az osztály csak akkor látszik, ha van az slf4j-api-ra implementation hatáskörű függőség
    public static final Logger LOG = LoggerFactory.getLogger(HelloWorld.class);

    public void logHello() throws IOException {
        InputStream propsStream = HelloWorld.class.getResourceAsStream("/hello.properties");
        Properties props = new Properties();
        props.load(propsStream);

        // logger használata
        LOG.info("Hello " + props.get("name"));
    }

    public static void main(String[] args) throws IOException {
        new HelloWorld().logHello();
    }
}
```

Példa: függőségek Gradle-ben

CODESPRING
OUR CODE YOUR SUCCESS
20 years

src/test/java/.../HelloWorldTest.java:

```
/**
 * Unit teszt példa
 */
@RunWith(JUnit4.class)
public class HelloWorldTest {

    private HelloWorld helloWorld = new HelloWorld();

    @Test
    public void testLogHello() throws IOException {
        // meghívjuk a tesztelendő kódot
        helloWorld.logHello();
        // tesztelünk valamit vele kapcsolatban
        // assert(...);
    }
}
```

Példa: gradle `maven-publish` plugin

- **nem** szükséges ahhoz, hogy Maven stílusú függőségeket használjunk a projektünkben–az natívan beépített
- cserébe generál egy Mavennel kompatibilis csomagot a mi projektünkből, hogy más Maven-alapú projektek hivatkozhassanak rá
- generál egy `pom.xml`-t a `build.gradle`-ben található információ alapján, majd a már meglévő JAR-csomagoló taszkot kiegészíti a POM becsomagolásával
- a publikálandó eredmények Maven stílusú ID-ját a `publishing` szekcióban adjuk meg
- taszkok:
  - `publishToMavenLocal` - feltölti a lokális Maven cache-be az eredményezett erőforrást (alapértelmezetten `~/.m2/repository`)
  - `publish` - feltölti az összes külső Maven tárolóba az erőforrást
- további információ

Példa: gradle `maven-publish` plugin

`build.gradle`:

```
plugins {
    id 'java'
    id 'maven-publish' // megadja a 'publish*' taskokat, pl. `publishToMavenLocal`
}

publishing {
    publications {
        // a generált pom információja itt módosítható
        maven(MavenPublication) {
            groupId = 'edu.codespring.gradleex.mavenplugin'
            artifactId = 'gradleex-mavenplugin'
            version = '1.0.0-SNAPSHOT'

            from components.java
        }
    }
}
```

Minőségbiztosítás gradle-lel

A `java` plugin magával hoz két fontos taszkot:

1. `test`
   - lefuttatja az összes JUnit tesztet, melyet felfedez a `src/test/java` forrásmappában
   - olvasható reportot generál a `build/reports` kimeneti mappában
2. `check`
   - általános burkoló taszk, mely függőségek révén lefuttat minden minőségbiztosítási taszkot
   - a `build` függ a `checktől`
   - a `check` alapértelmezetten csak a `test`től függ, de új QA-s taszkokkal bővíthetjük a projekt életciklusát

Statikus kódelemzés gradle-lel

Statikus kódelemző eszközök beépíthetőek a projektbe pluginok által. Ismertebb Java eszközök:

```
apply plugin: 'checkstyle'
apply plugin: 'pmd'
apply plugin: 'spotbugs'
```

- Mindegyik eszköz raportokat generál a **`build/reports`** kimeneti mappába
- Alapértelmezetten bármely eszközben észlelt hiba a build teljes eséséhez vezet. Ez a viselkedés egy-egy flag beállításával konfigurálható.
- További konfigurációs opciók: kimeneti formátum (HTML, XML), ruleset-ek be- és kikapcsolása, mappák vagy osztályok átugrása, stb.
- Az általános **`check`** tasktól automatikusan függőség kerül egy-egy eszközt futtató taszkra.
- Ezen eszközök használhatóak a fejlesztői környezetekben is, közös beállításokkal.

Multimodul projektek

- Több alprojektre bonthatjuk egy-egy nagy projektünket. Előnyök:
  - Segít a modularizálásban
  - Segít a függőségek szétválasztásában (egy-egy modul csak attól függ, amit ténylegesen használ)
  - Modulok függhetnek egymástól, de nem körkörösen, így kényszerítjük a tiszta kódolást
  - Egyszerre adhatunk ki karbantartási parancsokat, s ez minden alprojektre lefut
- A Gradle natívan támogatja a multimodul projekteket.
- Egy gyökérmodul alprojekteket (modulokat) importál a **`settings.gradle`** állományon keresztül
- Beállításokat globálisan lehet végezni minden projekten az **`allprojects`** closure-rel, vagy csak az alprojekteken a **`subprojects`**-szel
- A Gradle **`projects`** taszkjával információt kaphatunk a projekt hierarchiáról
- Ha a multimodulon belül másik modulra szeretnénk függőséget tenni, hivatkozhatunk a multimodulon belüli nevére:

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

Példa: Multimodul projektek gradle-lel

CODESPRING
OUR CODE YOUR SUCCESS
20 years

`settings.gradle`:

```
// almodulok definiciója relatív elérési útvonal szerint
include 'module1', 'module2'
```

gyökér `build.gradle`:

```
// ez a kódrészlet minden projektre érvényes
allprojects {
    group = 'edu.codespring.gradleex'
    version = '1.0.0-SNAPSHOT'
}

// ez a kódrészlet minden alprojektre érvényes
subprojects {
    apply plugin: 'java'

    sourceCompatibility = 1.8
    targetCompatibility = 1.8

    repositories {
        mavenCentral()
    }
}
```

Példa: Multimodul projektek gradle-lel

CODESPRING
OUR CODE YOUR SUCCESS
20 years

`module1/build.gradle`:

```
// csak a függőségeket szükséges megadni, a többi beállítás a szülő deszkriptorban található
dependencies {
    implementation group: 'org.slf4j', name: 'slf4j-api', version: '1.7.25'
    runtimeOnly group: 'ch.qos.logback', name:'logback-classic', version:'1.2.3'
}
```

`module2/build.gradle`:

```
plugins {
    id 'application'
}

dependencies {
    implementation group: 'org.slf4j', name: 'slf4j-api', version: '1.7.25'
    runtimeOnly group: 'ch.qos.logback', name:'logback-classic', version:'1.2.3'

    // belső projekt
    implementation project(':module1')
}

application {
    mainClass = 'edu.codespring.gradleex.frontend.HelloWorld'
}
```

Gyakorlatok

1. Készítsünk egy “Hello, World!” gradle alkalmazást. Tartsuk tiszteletben az elhelyezési konvenciókat. Tanulmányozzuk és próbáljuk ki a lehetséges Gradle taszkokat. Próbáljuk ki a projekt importálását IntelliJ-be.
2. Módosítsuk a projektet, hogy a “Hello, World!” üzenetet egy naplózási rendszer segítségével végezze. A függőséget konfiguráljuk a Gradle deszkriptorban.
3. Alkalmazzuk az `application` plugint, s teszteljük a `run` és `installDist` taszkokat.

Gyakorlatok

4. **Otthoni:** A korábban elkészített git tárolót módosítsuk, hogy legyen egy használható Gradle projekt:
   - Készítsünk neki egy **`build.gradle`** deszkriptort.
   - Mozgassuk el a meglévő állományokat a konvencionális folderstruktúrájukba.
   - Terjesszük ki a **`.gitignore`** állományt releváns részekkel (pl. a **`build`** mappa).
   - Szinkronizáljunk minden változást a GitLab szerverre.