

# Curs Java pentru testare automata



# Cuprins curs:

- ▶ Maven
- ▶ Log4j



# MAVEN

- ▶ Un build tool este un instrument care automatizează tot ceea ce ține de construirea proiectului software.
- ▶ Prin construirea proiectului software ne referim, de obicei:
  - ▶ Generarea documentației din codul sursă
  - ▶ Compilarea codului sursă
  - ▶ Packaging codul sursă compilată (fișier jar, de exemplu)
  - ▶ Instalarea codului pe un server



# MAVEN

- ▶ Maven simplifică și standardizează procesul de construire a proiectului. Se ocupă de compilare, distribuție, documentare, colaborare în echipă și alte sarcini fără probleme.
- ▶ Maven crește reutilizarea și se ocupă de majoritatea sarcinilor legate de construcție.



# MAVEN -POM

- ▶ POM în contextul Maven înseamnă Project Object Model. Este o unitate fundamentală de lucru în Maven. Este un fișier XML care se află în directorul de bază al proiectului ca pom.xml.
- ▶ POM conține informații despre proiect și diverse detalii de configurare utilizate de Maven pentru a construi proiectul.
- ▶ Un fișier POM este o reprezentare XML a resurselor proiectului, cum ar fi codul sursă, codul de testare, dependențe (JAR-uri externe utilizate) etc.
- ▶ POM conține referințe la toate aceste resurse. Fișierul POM ar trebui să fie situat în directorul rădăcină al proiectului căruia îi aparține.



# MAVEN -POM

- ▶ Când executăm o comandă Maven, îi dam lui Maven un fișier POM pentru a executa comenzile.
- ▶ Maven va executa apoi comanda pe baza resurselor descrise în POM.

## POM.xml

Maven

Build life cycle

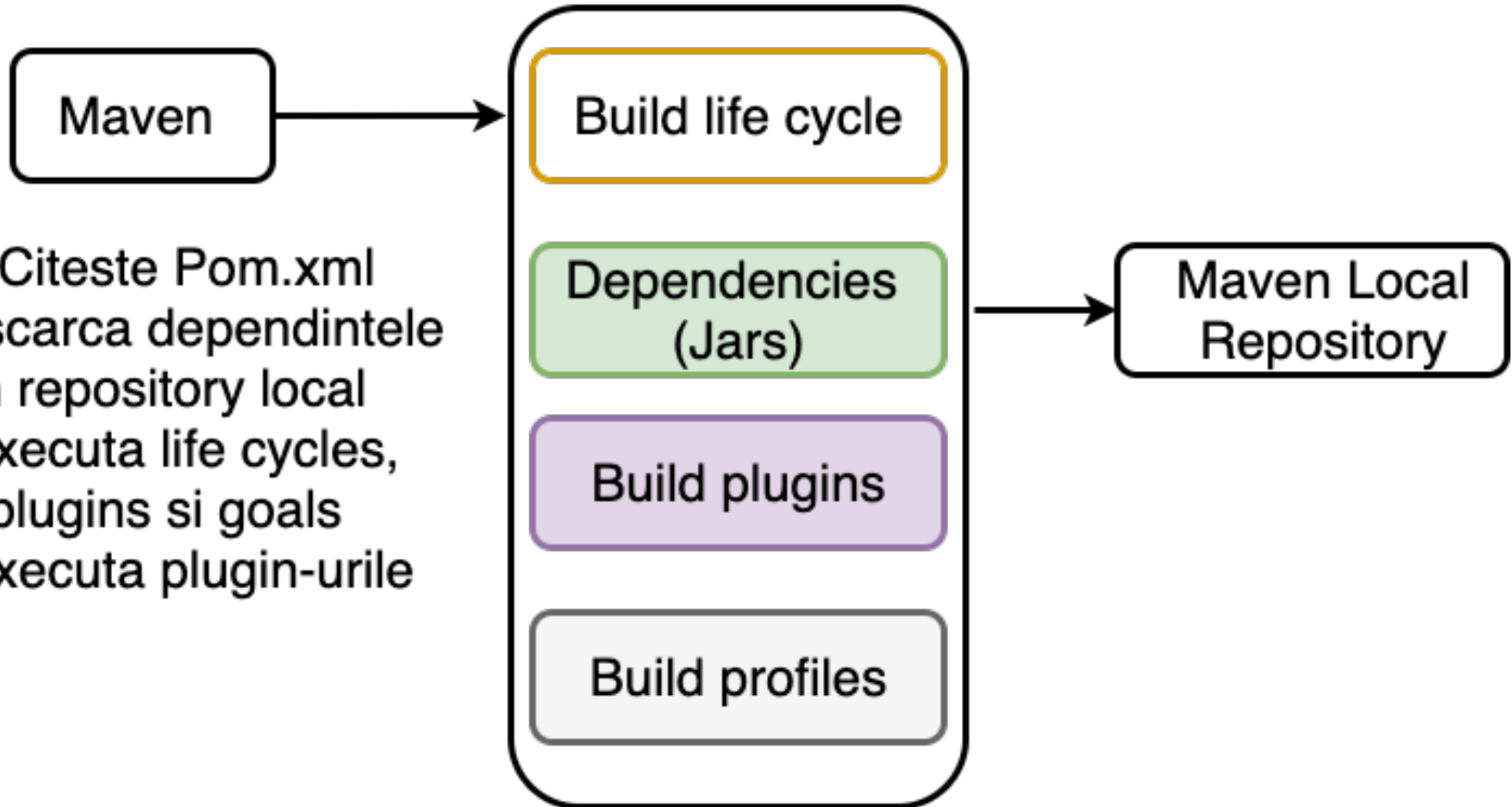
Dependencies  
(Jars)

Build plugins

Build profiles

Maven Local  
Repository

1. Citeste Pom.xml
2. Descarca dependintele in repository local
3. Executa life cycles, plugins si goals
4. Executa plugin-urile





# Maven build lifecycle

Procesul de build din Maven este împărțit în build lifecycle, phases și goals.

Build lifecycle constă dintr-o succesiune de build phases și fiecare build phase constă dintr-o succesiune de build goals.


Când rulăm Maven, îi transmitem o comandă lui Maven.

Această comandă este numele unui build lifecycle, phases sau goal.

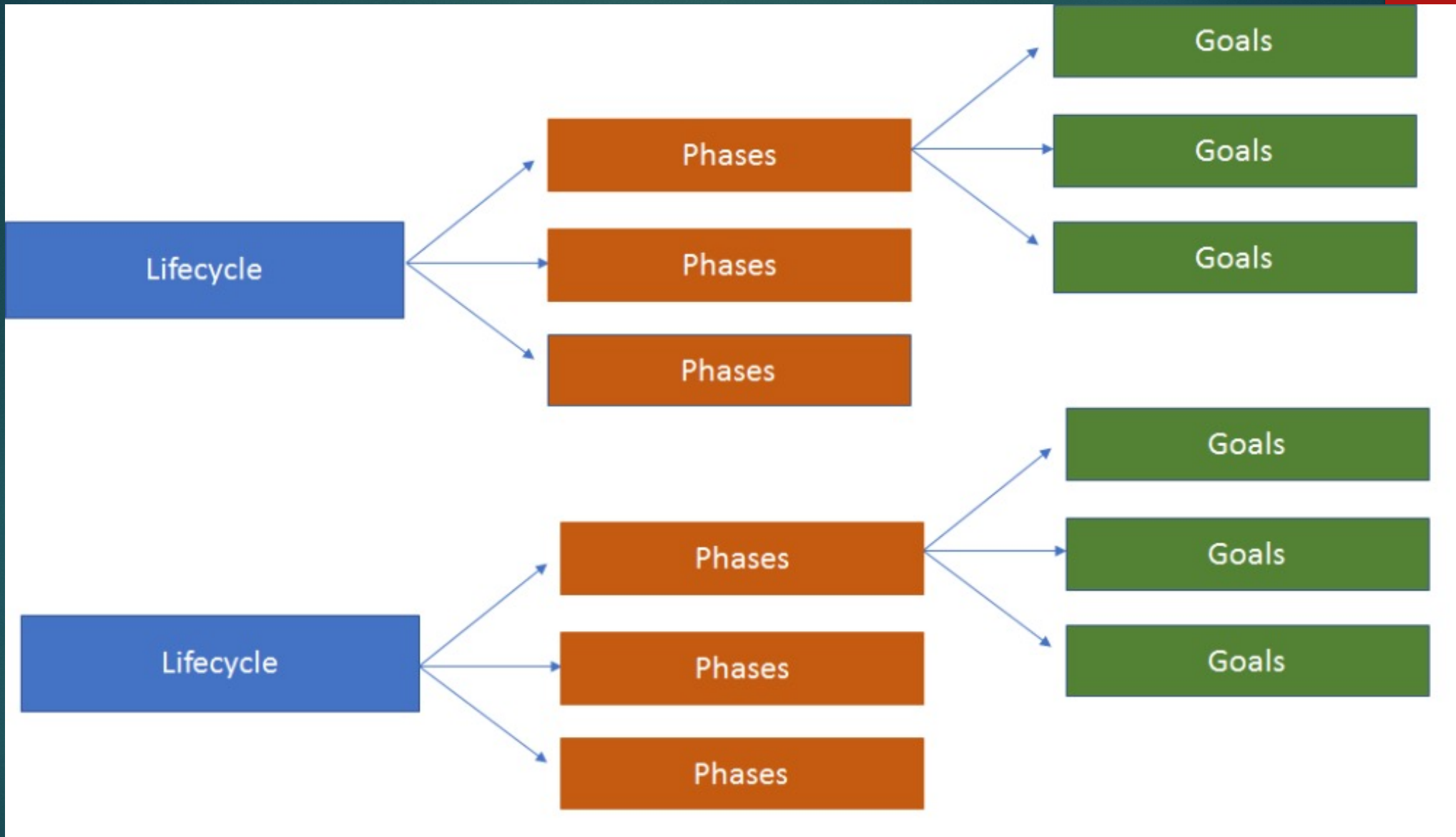


# Maven build lifecycle

Cand se solicită executarea unui Build lifecycle toate build phases din acel ciclu de viață sunt executate.



Dacă se solicită executarea unui build phase, se execută și toate build phase anterioare acesteia în secvența predefinită.





# Maven lifecycle

- ▶ **Default** (gestionează tot ceea ce ține de compilarea și packaging-ul proiectului)
- ▶ **Clean** (gestionează tot ceea ce privește eliminarea fișierelor temporare din directorul de ieșire, inclusiv fișiere sursă generate, clase compilate, fișiere JAR anterioare etc)
- ▶ **Site** (gestionează tot ceea ce ține de generarea documentației )



# Default lifecycle

<code>validate</code>	validate the project is correct and all necessary information is available.
<code>initialize</code>	initialize build state, e.g. set properties or create directories.
<code>generate-sources</code>	generate any source code for inclusion in compilation.
<code>process-sources</code>	process the source code, for example to filter any values.
<code>generate-resources</code>	generate resources for inclusion in the package.
<code>process-resources</code>	copy and process the resources into the destination directory, ready for packaging.
<code>compile</code>	compile the source code of the project.
<code>process-classes</code>	post-process the generated files from compilation, for example to do bytecode enhancement on Java classes.
<code>generate-test-sources</code>	generate any test source code for inclusion in compilation.
<code>process-test-sources</code>	process the test source code, for example to filter any values.
<code>generate-test-resources</code>	create resources for testing.
<code>process-test-resources</code>	copy and process the resources into the test destination directory.



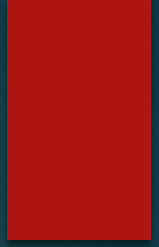


test-compile	compile the test source code into the test destination directory
process-test-classes	post-process the generated files from test compilation, for example to do bytecode enhancement on Java classes.
test	run tests using a suitable unit testing framework. These tests should not require the code be packaged or deployed.
prepare-package	perform any operations necessary to prepare a package before the actual packaging. This often results in an unpacked, processed version of the package.
package	take the compiled code and package it in its distributable format, such as a JAR.
pre-integration-test	perform actions required before integration tests are executed. This may involve things such as setting up the required environment.
integration-test	process and deploy the package if necessary into an environment where integration tests can be run.
post-integration-test	perform actions required after integration tests have been executed. This may include cleaning up the environment.
verify	run any checks to verify the package is valid and meets quality criteria.
install	install the package into the local repository, for use as a dependency in other projects locally.
deploy	package to the remote repository for sharing with other developers and projects.

# Default lifecycle



# Clean lifecycle



pre-clean	execute processes needed prior to the actual project cleaning
clean	remove all files generated by the previous build
post-clean	execute processes needed to finalize the project cleaning



# Site lifecycle

pre-site	execute processes needed prior to the actual project site generation
site	generate the project's site documentation
post-site	execute processes needed to finalize the site generation, and to prepare for site deployment
site-deploy	deploy the generated site documentation to the specified web server

# Default lifecycle

Phase	plugin:goal
<code>generate-resources</code>	<code>plugin:descriptor</code>
<code>process-resources</code>	<code>resources:resources</code>
<code>compile</code>	<code>compiler:compile</code>
<code>process-test-resources</code>	<code>resources:testResources</code>
<code>test-compile</code>	<code>compiler:testCompile</code>
<code>test</code>	<code>surefire:test</code>
<code>package</code>	<code>jar:jar</code> and <code>plugin:addPluginArtifactMetadata</code>
<code>install</code>	<code>install:install</code>
<code>deploy</code>	<code>deploy:deploy</code>



# Maven dependencies

- ▶ Unul dintre primele obiective pe care Maven le execută, este de a verifica dependențele necesare proiectului.
- ▶ Dependențele sunt fișiere JAR externe (biblioteci Java) pe care le folosește proiectul.
- ▶ Dacă dependențele nu se găsesc în depozitul local Maven, Maven le descarcă dintr-un depozit central Maven și le pune în depozitul local.
- ▶ Depozitul local este doar un director de pe hard disk-ul computerului.



# Maven build plugins

- ▶ Build plugins sunt folosite pentru a insera obiective suplimentare într-un build phase
- ▶ Dacă trebuie să efectuam un set de acțiuni pentru proiectul nostru care nu sunt acoperite de fazele și obiectivele standard de construcție Maven, puteți adăuga un plugin în fișierul POM.
- ▶ Maven are câteva plugin-uri standard pe care le putem folosi și putem chiar să le implementem și în Java, dacă avem nevoie.



# Maven build profiles

- ▶ Build profile sunt utilizate dacă avem nevoie să ne construim proiectul în moduri diferite.
- ▶ De exemplu, poate fi necesar să ne construim proiectul pentru computerul local, pentru mediul de dezvoltare și mediul de testare.
- ▶ Și poate că va trebui să-l construim pentru implementare în mediul de producție. Aceste versiuni pot fi diferite.
- ▶ Pentru a activa diferite versiuni, putem adăuga diferite build profiles în fișierele POM.
- ▶ Când executăm Maven, putem spune ce build profile să utilizăm.



# Log4J

- ▶ log4j este un API de logare, fiabil, rapid și flexibil scris în Java, care este distribuit sub licența software Apache.
- ▶ log4j a fost portat în limbile C, C ++, C #, Perl, Python, Ruby și Eiffel.
- ▶ log4j este extrem de configurabil prin fișiere de configurare externe în timpul rulării.
- ▶ Vizualizează procesul de înregistrare în termeni de niveluri de priorități și oferă mecanisme pentru a direcționa informațiile de înregistrare către o mare varietate de destinații, cum ar fi o bază de date, fișier, consolă, UNIX Syslog etc.



# Log4J

- ▶ log4j are trei componente principale:
- ▶ **loggers**: Responsabil de captarea informațiilor de înregistrare.
- ▶ **appenders**: Responsabil pentru publicarea informațiilor de înregistrare în diferite destinații preferate.
- ▶ **layouts** : responsabil pentru formatarea informațiilor de înregistrare în diferite stiluri



# Log4J

- ▶ Este thread safe.
- ▶ Este optimizat pentru viteză.
- ▶ Se bazează pe o ierarhie denumită logger. Acesta acceptă mai mulți anexe de ieșire per logger.
- ▶ Nu este limitat la un set predefinit de facilități.
- ▶ Comportamentul de înregistrare poate fi setat în timpul rulării utilizând un fișier de configurare.
- ▶ Este conceput pentru a gestiona excepțiile Java de la început.
- ▶ Folosește mai multe niveluri, și anume ALL, TRACE, DEBUG, INFO, WARN, ERROR și FATAL.



# Log4J levels

- ▶ ALL Toate nivelurile, inclusiv nivelurile personalizate.
- ▶ DEBUG Desemnează evenimente informaționale cu granulație fină, care sunt cele mai utile pentru depanarea unei aplicații.
- ▶ INFO Desemnează mesaje informaționale care evidențiază progresul aplicației la nivel cu granulație grosieră.
- ▶ WARN Desemnează situații potențial dăunătoare.
- ▶ ERROR Desemnează evenimente de eroare care ar putea permite aplicației să ruleze în continuare.
- ▶ FATAL Desemnează evenimente de eroare foarte severe care probabil vor duce aplicația la avort.
- ▶ STOP Cel mai înalt rang posibil și este destinat să dezactiveze înregistrarea.
- ▶ TRACE Desemnează evenimente informaționale mai fine decât DEBUG.



			x: Visible				
	FATAL	ERROR	WARN	INFO	DEBUG	TRACE	ALL
OFF							
FATAL	x						
ERROR	x	x					
WARN	x	x	x				
INFO	x	x	x	x			
DEBUG	x	x	x	x	x		
TRACE	x	x	x	x	x	x	
ALL	x	x	x	x	x	x	x

# Log4J levels