Java upgraden

Van Java 1.8 naar Java 9+

Als je vorig jaar naar een Javaconferentie ging en aan de zaal werd gevraagd wie er al op Java 9 of hoger in productie gedeployed had, gingen maar een paar handen de lucht in. 2019 lijkt echter het jaar te zijn waarin bedrijven toch echt serieus de migratie naar een nieuwere versie zijn gaan overwegen of er zelfs ermee bezig zijn. Dat in januari 2019 door Oracle de laatste publieke update geleverd is voor Java SE 1.8 kan wellicht een belangrijke reden zijn[1][2].

Ook niet te vergeten zijn de vele nieuwe features die geïmplementeerd zijn tussen Java 1.8 en de huidige Java 13. Waar moet je nu allemaal op letten bij het upgraden van Java 1.8 naar een hogere versie? In dit artikel zullen een aantal belangrijke aandachtspunten, waarop gelet moet worden bij upgraden, de revue passeren.

Breaking changes

Java 9 is een major versie upgrade en heeft een aantal breaking changes geïntroduceerd. Eén van de belangrijkste ervan is dat door de introductie van Project Jigsaw, interne API's en jars private zijn geworden. Het kan dus gebeuren dat projecten die prima op Java 1.8 werkten niet meer functioneren onder een nieuwere versie. Dit is de eerste keer in de geschiedenis van Java dat dit gebeurt.
Een van de redenen waarom veel bedrijven lang gewacht hebben met upgraden. Op het moment van schrijven van dit artikel is Java 11 de LTS (Long Term Support) versie en Java 13 de laatst gereleasde versie. Naar welke versie moet nu geüpgraded worden? Is dat Java 9, 11 of zelfs 13? Het antwoord daarop ligt gedeeltelijk bij een aantal factoren, zoals:

- Volwassenheid (Testen/voortbrengingsproces)
- De code
- Verplichtingen/Keuzes/Contracten/Licenties/JDK/etc.)
- Scope
- Toekomstvisie

Volwassenheid

Hoe makkelijk er geüpgraded kan worden is niet alleen een technische aangelegenheid. Het is ook een kwestie van volwassenheid. Volwassenheid van de testen, het voortbrengingsproces en van de codebase.

Testen

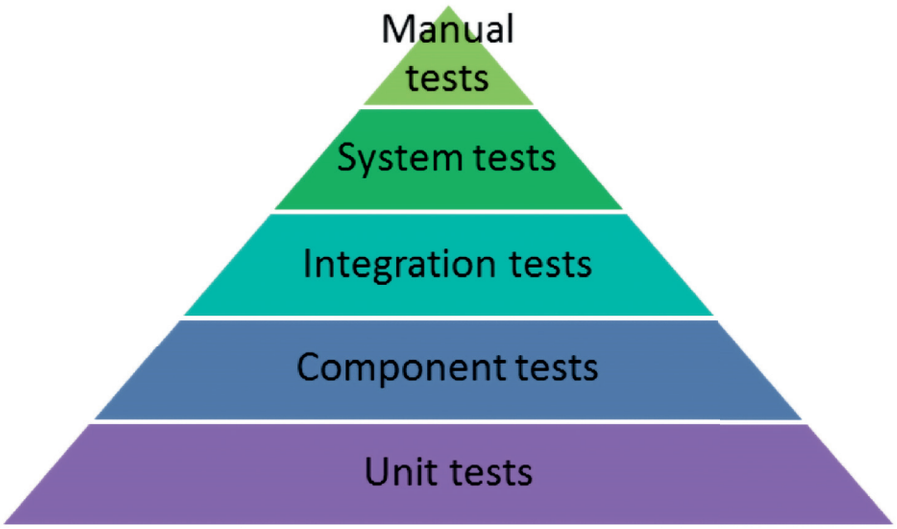
Hoe goed en hoe uitgebreid zijn de tests? Hoe zeker kan je aantonen dat alles nog blijft werken? Hoe snel worden fouten ontdekt, bij refactoren, libraries upgraden of Java zelf? De testpiramide van Martin Fowler[3] (zie afbeelding 1) wordt vaak gehanteerd bij het kijken naar volwassenheid van testen. Hoe beter de kwaliteit en de dekkingsgraad van de testen op elke laag van de piramide, hoe zekerder men kan zijn dat er bij veranderingen, zoals upgraden, alles goed blijft werken of dat fouten snel ontdekt worden. Naast de bovengenoemde testen, kan ook nog gekeken worden naar de volgende soorten testen:

- Load en stress testen
- Security en Penetration testen
- Statische Code quality testen

Voortbrengingsproces

In hoeverre wordt er geautomatiseerd naar productie gegaan, en wat houdt dit dan in? Zijn er veel handmatige stappen of is er een gestandaardiseerde omgeving waarop productie versies gebouwd worden? Door migratiestappen vroegtijdig in het proces in te bouwen, worden er eerder fouten ontdekt die door de introductie van bijvoorbeeld een nieuwe JDK ontstaan. Alles wat vaak gedaan moet worden, wordt op den duur saai en als iets saai wordt, worden fouten gemaakt. Het is dus belangrijk om dit soort repeterende taken zoveel mogelijk te automatiseren, waardoor het ontwikkel-, test-, deployproces zelf

Ivo Woltring is Software Architect en Codesmith bij Ordina JTech en houdt zich graag bezig met nieuwe ontwikkelingen in de software wereld.



Afbeelding 1.

kan groeien en waardoor vertrouwen in het product groeit. In een Continuous Integration / Continuous Deployment (CI/CD) straat kan men veel statische kwaliteitstesten inbouwen die extra inzicht kunnen geven in de kwaliteit van de code. Enkele voorbeelden hiervan zijn:

- Quality Assurance testen:
  - SonarQube
  - Nexus IQ (voorheen Sonatype CLM)
  - HP Fortify / Checkmarx

En andere tools als:

- PMD
- Checkstyle
- SpotBugs (voorheen FindBugs)

De code

Met de migratie van JDK 1.8 naar een nieuwere versie zijn een aantal grote en kleine veranderingen geïntroduceerd, waar we in code rekening mee moeten houden. De belangrijkste hiervan zijn:

- Gebruik van interne API's
- Afhankelijkheid van Java EE modules
- Split packages
- Runtime
- Versionering
- Classloader

Gebruik van interne API's

Door de komst van het Java Platform Module System (JPMS)[4], wat voorheen Project Jigsaw genoemd werd, is het gebruik van de interne Java API niet meer mogelijk. De belangrijkste hiervan zijn de `com.sun.*` en `sun.*` packages[7] (voorbeeld Listing 1, 2 en 3).
Listing 1 zal bij compileren met Java 1.8 warnings geven maar wel compileren.
In Java 9 en hoger zal het niet meer compileren met de boodschap uit listing 3.
In dit soort gevallen zal de code moeten worden aangepast om de migratie naar een nieuwe versie te laten slagen. Het is ook belangrijk 3rd party libraries op dit gebruik te controleren.

Afhankelijkheid van Java EE modules

In Java SE zat heel wat code die eigenlijk bedoeld was voor de Java EE wereld. Deze zijn ondertussen verwijderd of gemarkeerd voor verwijdering (deprecated)[6]:

- `java.xml.ws` (JAX-WS, Plus SAAJ and Web Services Metadata)
- `java.xml.bind` (JAXB)
- `java.activation` (JAF)
- `java.xml.ws.annotation` (Common Annotations)

```
import sun.misc.BASE64Encoder;
public class ComSunBase64 {
   private static void getBase64EncodedString(String inputString) {
       String encodedString = new BASE64Encoder().encode(inputString.getBytes());
       System.out.println("encodedString = " + encodedString);
   }
   public static void main(String[] args) {
       getBase64EncodedString("IvoNet.nl");
   }
}
```

Listing 1.

JAVA 9 IS EEN MAJOR VERSIE UPGRADE EN HEEFT EEN AANTAL BREAKING CHANGES GEÏNTRODUCEERD

```
Java 1.8> javac nl/ivonet/ComSunBase64.java
nl/ivonet/ComSunBase64.java:19: warning: BASE64Encoder is internal proprietary API and may be removed in a future
release
import sun.misc.BASE64Encoder;
               ^
nl/ivonet/ComSunBase64.java:27: warning: BASE64Encoder is internal proprietary API and may be removed in a future
release
        String encodedString = new BASE64Encoder().encode(inputString.getBytes());
                                   ^
2 warnings
Java 1.8> java nl.ivonet.ComSunBase64
encodedString = SXZvTmV0Lm5s
```

Listing 2.

```
Java 11> javac nl/ivonet/App.java
nl/ivonet/App.java:19: error: cannot find symbol
import sun.misc.BASE64Encoder;
               ^
  symbol:   class BASE64Encoder
  location: package sun.misc
nl/ivonet/App.java:27: error: cannot find symbol
        String encodedString = new BASE64Encoder().encode(inputString.getBytes());
                                   ^
  symbol:   class BASE64Encoder
  location: class App
2 errors
```

Listing 3.

- `java.corba` (CORBA)
- `java.transaction` (JTA)
- `java.se.ee` (Aggregator module for de 6 modules hierboven)
- `jdk.xml.ws` (Tools for JAX-WS)
- `jdk.xml.bind` (Tools for JAXB)

Waar je met de --add-modules optie in Java 9 modules nog kon activeren zijn deze in Java 11 echt verwijderd. Ze staan wel in maven central, zodat ze als losse dependencies opgenomen kunnen worden.

Split packages

Een 'Split Package' is als er twee (of meer) packages bestaan met dezelfde naam, maar geplaatst in verschillende libraries. Vanaf Java 9 is een module niet meer in staat uit eenzelfde package te lezen uit twee verschillende modules. Sterker nog: geen twee modules mogen deze packages hebben, of ze nu geëxporteerd zijn of niet. Echter, omdat je code op de class path staat, wordt deze toegekend aan de zogenaamde "unnamed" module en deze wordt niet gecontroleerd op module gerelateerde zaken. Als er dus packages gespleten (split) zijn tussen een module en de classpath, dan zal het package op de classpath niet gezien worden. De compiler zal dan klagen over missende classes ook al staan ze effectief op het classpath. Codewijzigingen zijn hier meestal nodig, wat upgraden naar een nieuwe versie van Java uitdagender maakt.

Runtime

Bestanden als rt.jar, tools.jar, en dt.jar zijn verdwenen toen de JDK gemodulariseerd werd. Dit zal voor de meeste software geen problemen opleveren, maar is wel iets waar rekening mee gehouden moet worden door bijvoorbeeld bouwers van IDE's.

Versionering

Vanaf Java 9 wordt de 1.x losgelaten en zal Java verder volle versies uitbrengen zoals Java 11. Dit betekent echter niet dat al deze versies zogenaamde "Major versions" zijn. Java 9 was de laatste major versie. Alle versies erna zijn versies die elk half jaar uitkomen met nieuwe features. Eens in de drie jaar zal een LTS versie uitgebracht worden. Vaak wordt in CI/CD pipelines gecontroleerd op welke javaversie gebruikt wordt door gebruik te maken van de versiestring. Het formaat is veranderd dus de parser zal aangepast moeten worden.. Denk hierbij aan: `java.version`, `java.runtime.version`, `java.vm.version`, `java.specification.version`, `java.vm.specification.version`.

Classloader

Weinig projecten zullen last van deze verandering hebben, tenzij je een libraryontwikkelaar bent, maar toch is het de moeite waard om te noemen. De classloader is van een andere type geworden waardoor je niet meer kunt casten naar bijvoorbeeld java.net.URLClassLoader. Echter de compiler zal niet klagen. Pas tijdens het runnen van de applicatie zal deze naar voren komen[7] (Listing 4):

```
Exception in thread "main" java.lang.ClassCastException: java.base/jdk.internal.loader.
ClassLoaders$AppClassLoader cannot be cast to java.base/java.net.URLClassLoader
        at nl.ivonet.JavaMagazineUrlClassLoader.main(JavaMagazineUrlClassLoader.java:27)
```

Listing 4.

Verplichtingen/keuzes

Contracten met leveranciers, lopende licenties en dergelijke kunnen gevolgen hebben voor de te volgen migratie stappen. Hoe vrij ben je als organisatie om bijvoorbeeld de JDK te kiezen, of adviseert een Vendor een specifieke JDK/JVM? Zijn er specifieke redenen waarom de huidige JDK gekozen is en wat zijn deze dan?

Vanaf Java 9 is Oracle overgestapt op een 6-maandelijkse releasecyclus met elke drie jaar een LTS versie. Java 11 is een LTS versie en de volgende zal Java 17 zijn. Voor deze LTS versies zal Oracle updates leveren en met de juiste licensing voor langere tijd. Voor bedrijven die stabiliteit hoog in het vaandel hebben is het wellicht handig om op de LTS versies te blijven. Bedrijven die van verandering houden en de nieuwe features in de taal belangrijk vinden, kunnen met de tussentijdse versies meebewegen. De support van een tussentijdse versie vervalt op het moment dat een nieuwe versie wordt uitgebracht. Een voordeel van meebewegen per versie is dat migreren een 'no brainer' kan gaan worden omdat het vaak gedaan wordt en de veranderingen per versie klein zullen zijn ten opzichte van migreren per drie jaar.

Een nadeel van de halfjaarlijkse releases kan fragmentatie zijn. Binnen één bedrijf zou je vele versies van Java kunnen hebben. Is dit wenselijk? Containerisatie zou hierbij kunnen helpen, omdat containers voor de benodigde isolatie zorgen. Bij upgraden van Java 1.8 naar Java 11 of hoger zijn dit belangrijke zaken om een standpunt op in te nemen.

Scope

Een belangrijke factor bij het upgraden van Java is de scope van de migratie. Moet de hele code base worden gemigreerd, of zijn er een aantal key applicaties die (eerst) moeten? Zijn er applicaties die niet gemigreerd gaan worden? Is er uniformiteit in de inrichting van de verschillende omgevingen? Wat zijn de aantallen waar rekening mee moet worden gehouden?

Het upgraden van één project onderhouden door één team met een test- en productieomgeving is aanzienlijk minder werk dan honderden applicaties over verschillende teams met verschillende test- en productieomgevingen. Hoe zien de verschillende omgevingen er uit? Dubbel uitgevoerd, virtueel, cloud, clustered of een combinatie hiervan? Welke Java platform(en) worden er gebruikt? Denk hierbij aan bijvoorbeeld Java SE / JavaEE / JakartaEE? Wordt er gebruik gemaakt van applicatieserver(s) zoals bijvoorbeeld Websphere, Wildfly/JBoss, Payara en van welke versie van Java zijn deze afhankelijk?

Toekomstvisie

Is dit een eenmalige migratie of wordt er verwacht dat regelmatig gemigreerd gaat worden naar nieuwere versies? (Evolutie versus revolutie) Ofwel: is er een JDK onderhoudsplan?

Conclusie

Dit artikel geeft geen antwoord op alle vragen die erin gesteld zijn en dat kan ook niet, omdat elke upgrade zowel organisatorische als technische aspecten in zich heeft. Dit maakt het niet mogelijk om overal een eenduidig antwoord op te geven (maatwerk). Bij de upgrade van Java 1.8 naar hoger is het wel verstandig om minimaal naar de volgende LTS te upgraden. Dit is niet Java 9, maar Java 11! In dit artikel is een poging gedaan om een compleet beeld te schetsen van waar rekening mee moet worden gehouden bij de upgrade. Maar er zullen vast taken aan toegevoegd kunnen worden die situatieafhankelijk zijn. Succes met de upgrade naar Java 11 of hoger en hopelijk geeft dit document de nodige steun daarbij. ■

EEN BELANGRIJKE FACTOR BIJ HET UPGRADEN VAN JAVA IS DE SCOPE VAN DE MIGRATIE

REFERENTIES

[1] Oracle Java SE Support Roadmap: **https://www.oracle.com/technetwork/java/java-se-support-roadmap.html**

[2] End of Public Updates: **https://blogs.oracle.com/java-platform-group/end-of-public-updates-is-a-process%2c-not-an-event**

[3] Fowlers' Test Piramid: **https://martinfowler.com/articles/practical-test-pyramid.html**

[4] JSR-376 - Java Platform Module System: **https://jcp.org/en/jsr/detail?id=376 / https://en.wikipedia.org/wiki/Java_Platform_Module_System**

[5] Java 9 removed features: **https://www.oracle.com/technetwork/java/javase/9-removed-features-3745614.html**

[6] JEP 320 - Remove Java EE and Corba modules: **http://openjdk.java.net/jeps/320**

[7] Companion Code: **http://ivo2u.nl/ZM**