Java 9,10,11

New Capabilities

About Myself

- I am Maruthi Janardhan
- Consulting Chief Architect at Trilogy
  - I currently do cross company architectural and code reviews and make recommendations
- Been doing Java since 2000
- Been with ANZ, IBM and there after into my own startups
- I train sometimes

Java has done something right for sure

- Humungous enterprise projects are today running on java and new projects continue to be built with java
- Has high market share of the programming world: https://www.tiobe.com/tiobe-index/
- Maturity of libraries, tools, community, documentation and eco system in general is very hard to beat

- JVM is hard to beat even today:
  - Garbage collection
  - Optimisations across processors
  - Reliability
  - Jython, Kotlin, Clojure, Groovy, Scala and Jruby

- Enterprises like when a corporate stands in and says - hey I have your back for next n years
- Enterprises also like when the decision making process on future path is democratic and there is no vendor lock in (Ex: Amazon Corretto, OpenJdk)
- Having said all this - frontends and mobile platforms are a different story that Java has not fit in.

Modularity

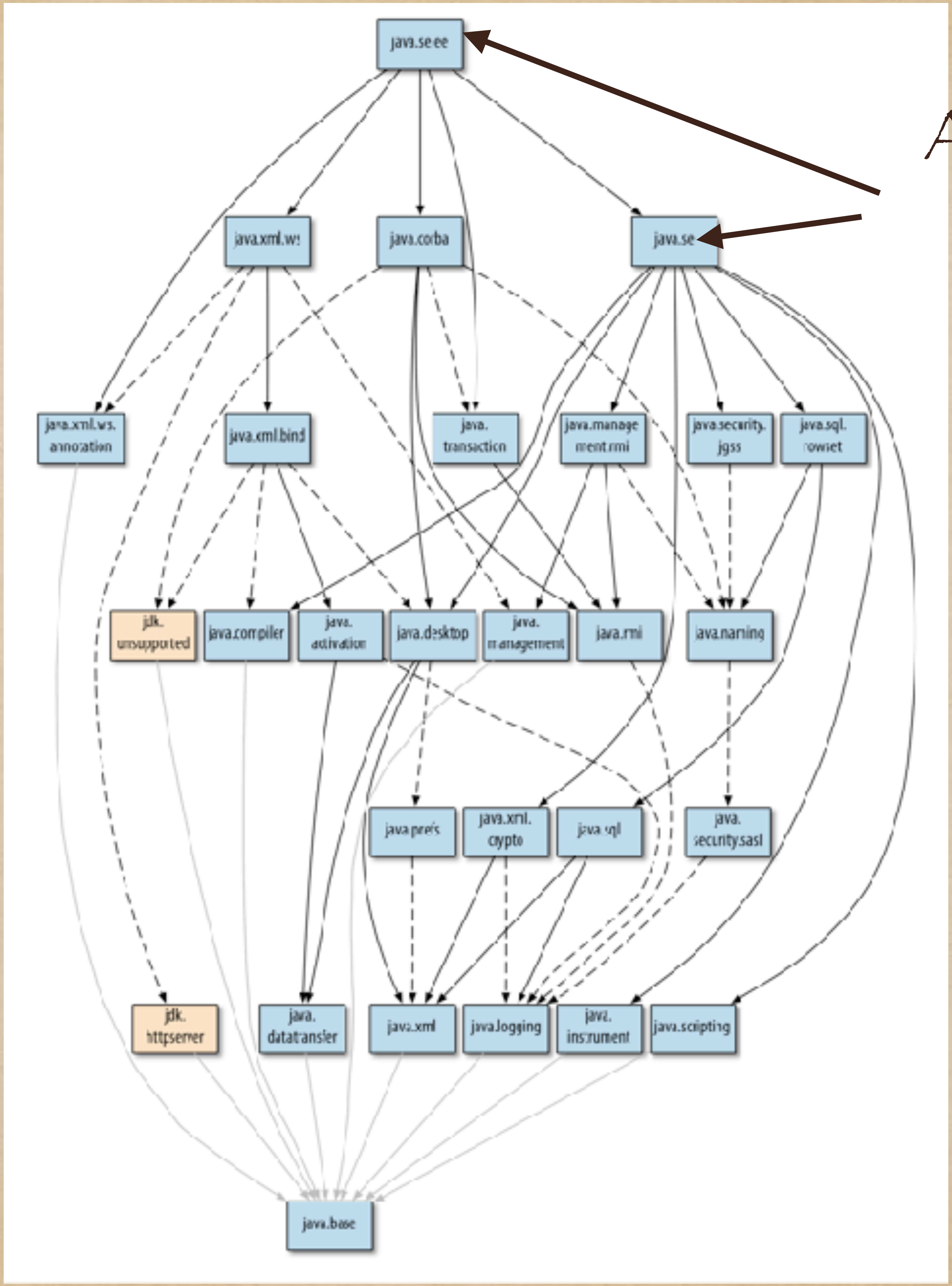
- Modularity is driven from ground up
    - Packages vs Namespaces
    - Jars
    - Maven, gradle, etc
    - Classpath hell is better than DLL hell

Platform Module System (JPMS)

- Java has accumulated a lot of legacy code as a platform.
- It has been a platform that works on IoT devices to large servers, swing, awt, legacy data structures, old security spec, etc. So it has everything that anybody needs in one monolith.
- JPMS was designed to solve this: module—a uniquely named, reusable group of related packages and resources

rt.jar

- its about 60 MB on jdk 8. JRE itself hits about 150MB
- Take CORBA: The classes supporting CORBA in the JDK are still present in rt.jar to this day.
- No matter whether you use CORBA or not, the classes are there.
- unnecessary use of disk space, memory, and CPU time.
- Cognitive overhead of obsolete classes showing up in IDE autocompletions and documentation during development.
- Same goes for AWS/Swing/JavaFx
    - Backward compatibility is one of the most important guiding principles for Java



Aggregator Modules

Optimisation

- Because the module system knows which modules belong together, including platform modules, no other code needs to be considered during JVM startup.
- Before modules, this was much harder, because explicit dependency information was not available and a class could reference any other class from the classpath.
- find jmods and use the jmod tool
  - `$JAVA_HOME/bin/jmod list java.net.http.jmod`

Lets create some libraries

- httplib: A library that exposes:

```
public String get(String url)
```

- Put this into an independent project
- Create another library: weather

```
public Weather getWeather(String city)
```

- weather depends on httplib. Get it running with a main() in weather project

Httplib get

```
public String get(String url) throws Exception {
    URL obj = new URL(url);
    HttpURLConnection con = (HttpURLConnection) obj.openConnection();

    con.setRequestProperty("User-Agent", "APIClient");

    int responseCode = con.getResponseCode();
    System.out.println("\nSending 'GET' request to URL : " + url);
    System.out.println("Response Code : " + responseCode);

    BufferedReader in = new BufferedReader(new InputStreamReader(con.getInputStream()));
    String inputLine;
    StringBuffer response = new StringBuffer();

    while ((inputLine = in.readLine()) != null) {
        response.append(inputLine);
    }
    in.close();

    return response.toString();
}
```

```
URL for Weather:
http://api.openweathermap.org/data/2.5/weather?q=CITY&APPID=cc3d457fb9707805588d131ce51804be

For Forecast:
http://api.openweathermap.org/data/2.5/forecast?q=CITY&APPID=cc3d457fb9707805588d131ce51804be


GSON Parsing:
Gson gson = new Gson();
HashMap<String, Object> dataMap = gson.fromJson(wjson, HashMap.class);
```

module-info.java

```
module Java11 {
    requires java.sql;

    exports com.mydomain;
}
```

- Readability
- Accessibility - No types from a non exported package can be used by other modules—even if types inside that package are public
- Setup module dependencies

Modules and Automatic Modules

- Setup module in Httplib

```
module httplib {
    exports com.mydomain.httplib;
}
```

- Setup module in weather. Add gson automatic module

```
module weatherlib {
    requires httplib;
    requires gson;

}
```

- Setup classpath dependencies and run in IDE

- Run on command line

```
java --module-path Weather.jar:HttpLib.jar:gson-2.8.5.jar -m weatherlib/
com.mydomain.weatherplus.WeatherService
```

- Transitive read relationships

```
      requires transitive java.logging;
```

- Requiring java.se.ee or java.se in application modules isn't a good idea. It means you're effectively replicating the pre-Java 9 behavior of having all of rt.jar accessible in your module.

- Qualified Exports

```
exports com.mydomain to com.yourdomain.util;
exports jdk.internal.jimage to jdk.jlink;
```

- Bad idea because this goes against fundamental premise of not knowing your users

Enhance with transitive

- Create another library weatherpluslib

```
public Weather getTomorrowsForecast(String city)
public Weather getWeather(String city)
```

- Uses weatherlib and httplib to make calls
- Run in IDE, run on command line by building jars

- Code compiled and loaded outside a module ends up in the unnamed module
  - The unnamed module is special: it reads all other modules
- In Java 8 and earlier, you could use unsupported internal APIs without repercussions. With Java 9, you cannot compile against encapsulated types in platform modules
- To aid migration, code compiled on earlier versions of Java using these internal APIs continues to run on the JDK 9 classpath for now

- You can still use identifiers called module, requires, etc in your other source files, because the module keyword is a restricted keyword.
    - It is treated as a keyword only inside module-info.java
- Java executable has a new —module, —show-module-resolution, —module-path options
- Using the explicit dependency information of module descriptors, module resolution ensures a working configuration of modules before running any code. (Unlike classnotfound exceptions)

Linking Modules

- An optional linking phase is introduced with Java 9, between the compilation and run-time phases
- jlink command and its options

```
/Library/Java/JavaVirtualMachines/jdk-11.0.1.jdk/Contents/Home/bin/jlink --module-
path WeatherPlus.jar:Weather.jar:HttpLib.jar:gson-m-2.8.5.jar --add-modules
weatherpluslib --output linkedfiles
```

- Jlink creates a package

DI with service providers

- One module can declare the API
- There can be many modules providing implementations for the API
- Module declarations can declare API and implementation. Scanning happens automatically

Providers for Pre init check

- Module that declares the API

```
module WeatherApi {
    exports com.mydomain.weather.weatherapi;
    uses WeatherApi;
}
```

- Module that implements

```
module weatherpluslib {
   requires WeatherApi;
   provides WeatherApi with WeatherServicePlus;
}
```

- Module that uses

```
module WeatherClient {
    requires WeatherApi;
}
```

- If A uses B, C provides B and requires D, A cant start till D is available in module path though it has no knowledge of C
- JLink however does not care about providers and hence does no service binding!
    - reason to not do automatic service binding in jlink is that java.base has an enormous number of uses clauses. The providers for all these service types are in various other platform modules. Binding all these services by default would result in a much larger minimum image size
    - Use —add-modules to include the providers in jlink