JAVA 11

Rony Keren

JOHN BRYCE
Leading in IT Education
a matrix company

Topics – Java 9

API & code

- Reactive programming with flow
- Collection Factories
- Stream API new features
- Private interface methods
- HTTP/2 API
- Stack walking

Environmental

- Modular Java & Jlink
- Jshell
- Multi version jars
- G1 made default

Topics – Java 10

API & code

- Local vars
- Custom GC API

Environmental

- App CDS
- Full Parallel GC
- Heap Allocation

Topics – Java 11

Removed API's & Tools

API & code

- Vars for LAMBDAs
- java.lang.String
- Predicate.not()
- Incubator java.net
- Reference.clone()

Environmental

- AppCDS modules support
- Single File Launch
- Lazy Allocated Compiling Threads
- G1 Update
- ZGC

OpenJDK & Oracle

- For JDK11 - Identical in Java support
- Oracle will not update OpenJDK anymore
- OracleJDK usage in production requires licensing
  - Free for developing and testing
- Note:
  - OpenJDK got no commercial features
    - -XX:+UnlockCommercialFeatures results with an error
  - Advanced Management Console – available on OracleJDK only

Release History & Roadmap

- Java 9 - July 27, 2017
- Java 10 - March 2018
- Java 11 - September 2018
  - Long Term Support
    - Oracle – N/A
    - OpenJDK - September 2022
- Java 12 - March 2019
    - Oracle – N/A
    - OpenJDK - September 2019

JAVA 9

API & CODE

Reactive programming with Flow

Reactive programming enhancements
made in latest versions

Reactive Programming

“Reactive programming is an asynchronous programming paradigm concerned with data streams and the propagation of change” Wiki

Why do we need this?

- Relevant for asynchronous messaging only
- When facing unknown amounts of requests we usually go asynchronous
- When lots of requests are published we might face back-pressure
  - need lots of processing threads….
  - need unbounded Queues….
  - But in most cases we are forced to restrict both Threads & Queues

Reactive Programming

Goal is to:

- Reduce blocking back pressure
  - Done by splitting requests into small phases
  - Each phase can be forked separately in the execution path
  - Use a strong mechanism that simplifies all this

- Good reactive API should encapsulate thread management & communication complexity

Reactive Programming

- So, we need: Stream API, dynamic invoker, thread pools & reactive infrastructure
- Java 8 provides
  - Stream API & Parallel streams backed by Fork-Join
  - Static and dynamic programming support
- Java 9 adds the core infrastructure for reactive programming
  - Flow – unit that processes events and encapsulates concurrency
  - Subscriber – event endpoint
  - Publisher – generates events and publishes to registered subscribers
  - Processors – subscribing interceptors (for creating subscription chain)

Reactive Programming

How do we do it?

- Create a Flow.Publisher
- Register Flow.Subscribers via Publisher.subscribe()
- Implement Subscriber to handle events:
  - onSubscribe()
  - onNext()
  - onError()
  - onComplete()
- Use Publisher to generate events

- Flow acts like a Pipe here – passing events from publisher to the ‘Sink’ side – the Consumer
- BUT –unlike pipes - it uses Executor, the daemon common pool (ForkJoin)

Reactive Programming

- When creating Publisher
  - Default constructor uses common pool (Fork-Join daemon pool)
  - Alternative executors may be used instead
  - This is how all thread complexity remains hidden

- Multiple subscribers may be registered to a single Publisher
  - Use publisher.subscribe()

```
//Create Publisher (works with common-pool)
SubmissionPublisher<String> publisher = new SubmissionPublisher<>();

// Create Publisher with dedicated pool
Executor e=Executors.newFixedThreadPool(3);
SubmissionPublisher<String> publisher = new SubmissionPublisher<>(e);
```

Subscriber

```java
  public class MySubscriber<T> implements Subscriber<T> {
       private Subscription subscription;
       @Override
       public void onSubscribe(Subscription subscription) {
         this.subscription = subscription;
      }
       @Override
       public void onNext(T item) {
         subscription.request(1); //Long.MAX_VALUE may be considered as unbounded
       }
       @Override
       public void onError(Throwable t) {
         t.printStackTrace();
      }
       @Override
       public void onComplete() {
         System.out.println("Done");
      }
}
```

Reactive Programming

Publishing messages

- Use publisher.submit(T)
- Each registered Subscriber get its own instance of Subscription

```
//Create Publisher
SubmissionPublisher<String> publisher = new SubmissionPublisher<>();

//Register Subscriber
MySubscriber<String> subscriber = new MySubscriber<>();
publisher.subscribe(subscriber);

//Publish messages
String[] items = {"msg1", "msg2", "msg3"};
Arrays.asList(items).stream().forEach(publisher::submit);
```

Processors

- Enhanced subscribers
- While subscribers acts as endpoints, processors delegates messages
- Processor uses Function<T,R> to process messages
  - Incoming message is T
  - Outgoing message is R (which might be T as well..)

Reactive Programming

Processors *example*

```
public class Processor1<T,R> extends SubmissionPublisher<R> implements Processor<T,
R> {

  private Function<? super T, ? extends R> function;
  private Subscription subscription;

  public MyTransformProcessor(Function<? super T, ? extends R> function) {
    super();
    this.function = function;
 }
  @Override
  public void onSubscribe(Subscription subscription) {
    this.subscription = subscription;
  }
  @Override
  public void onNext(T item) {
    submit((R) function.apply(item));
    subscription.request(1);
  }…
```

Reactive Programming

Processors

- Now we can define a subscription chain
- This is haw we split asynchronous tasks while using thread pools

```
SubmissionPublisher<String> publisher = new SubmissionPublisher<>();
MySubscriber<Integer> subscriber = new MySubscriber<>();

//Creating Midpoints Processors
Processor1<String, String> p1 =
    new Processor1<>(s -> {if(s.equals("msg1"))return “100"; return “200";});
Processor1<String, Integer> p2 =
    new Processor1<>(s -> Integer.parseInt(s));

//Configuring subscription chain
publisher.subscribe(p1);
p1.subscribe(p2);
p2.subscribe(subscriber);
```

Collection Factories

Easy to use and remember – new factories for creating unmodifiable collections

Collection Factories

There are many different ways to create collections:

```
List<Integer> numbers = new ArrayList<>();
for(int i=0;i<100;i++){
    numbers.add(i);
}
```

```
List<Integer> numbers = Arrays.asList(1,2,3…)
```

```
List<Integer> n=Collections.unmodifiableList(new ArrayList() {
 {add(1);add(2);add(3);}
});
```

```
List<Integer> numbers = Stream.of(1,2,3…).collect(Collectors.toList());
```

- Java 9 provides a much straight forward methods

Collection Factories

Java 9 collection factory methods:

  - List.of()
  - Set.of()
  - Map.of()

- Generates unmodifiable collections
  - Updates causes UnsupportedOperationException

- List.of() & Set.of() of methods takes var-args for 10 elements or more
- BUT - in order to save array allocations, all got 10 different ‘of()’ methods:
  - Set/List: of(T t1), of(T t1, T t2), of(T t1, T t2, T t3), ……
  - Map: of(K k, V v), of(K k1, V v1, K k2, V v2) ….
  - Means that from 0 to 9 elements – no arrays are allocated

Collection Factories

Example

```
List<String> wordsList = List.of("a","b","c","d","a");

Set<String> wordsSet = Set.of("a","b","c","d");

Map<Integer,String> wordsMap = Map.of(1,"a",2,"b",3,"c",4,"d",5,"a");
```

Streams new features

Cool APIs can be even cooler..

Streams new features

Method improvements

- Java 8 iterate() method cannot have any stop condition but limit()
  - Iterate(T seed, UnaryOperator<T>)

```
//Generates a Stream<Integer> with values: 1, 2, 4, 8
Stream.iterate(1, n -> n*2).limit(4);
```

  - If we don’t call limit() - iterate() never returns…

- Java 9 provides another version for iterate() for saving this unwanted pause:
  - Iterate(T seed, Predicate<T>, UnaryOperator<T>)
  - Iteration continues as long as test returns ‘true’

```
//Generates a Stream<Integer> with values: 1, 2, 4, 8
Stream.iterate(1, n -> n<=8 ,n -> n*2);
```

Streams new features

New methods:

- takeWhile(Predicate<T>) – passes elements as long as test returns ‘true’
- dropWhile(Predicate<T>) – drops elements as long as test returns ‘true’

```
//Generates a Stream<Integer> with values: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Stream<Integer> nums=Stream.iterate(1, n -> n<=10, n -> n+1);
List<Integer> low =nums.takeWhile(n -> n<=5).collect(Collectors.toList());
System.out.println(low);

//Output:
[1, 2, 3, 4, 5]
```

```
//Generates a Stream<Integer> with values: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Stream<Integer> nums=Stream.iterate(1, n -> n<=10, n -> n+1);
List<Integer> high=nums.dropWhile(n -> n<=5).collect(Collectors.toList());
System.out.println(high);

//Output:
[6, 7, 8, 9, 10]
```

Streams new features

ofNullable(T)

- If T is null – returns an empty stream
- If T is an object – returns a Stream<T> with a single object in it
- Mostly relevant when getting flat-maps from an object that might be null

```
//lib & lib.getBooks() might be NULL

Library lib = loadLibrary(id);
Stream<Book> books= Stream.ofNullable(lib)
                          .flatMap(l-> Stream.ofNullable(l.getBooks()));

//we end up with a Book stream of an empty stream (in case lib was NULL)
//in addition, if library returns NULL
```

Interface Private Methods

What happens when two or more default methods share code?

Interface Private Methods

Java 8 provides interface default & static method support

- But what if 2 or more default / static method share code ?

Java 9 provides interface private methods

- These methods are available to other private / default / static methods
- Are not inherited or visible for implementing classes

⚠ NOTE

Private methods are counted.

So, @FunctionalInterface cannot have both abstract & private methods…

Interface Private Methods

Example

```
public interface Recorder<T> {
        default void startRecord() {
                System.out.println("recording started at: ");//+timestamp());
        }
        void record(T data);
        default void endRecord() {
                System.out.println("recording stopped at: ");//+timestamp());
        }
        private String timestamp() {
                return DateFormat.getTimeInstance().format(new Date());
        }
}
```

```
RecorderImpl<String> r=new RecorderImpl<>();
r.startRecord();
r.record("Hello World!");
r.endRecord();
```

HTTP/2 Support

Simple Http client API with HTTP/2 support

HTTP/2 Support

HTTP/2

- Uses multiplexing
  - Can send multiple parallel requests over single TCP connection
  - HTTP/1 is limited to only 6 at a time
- Headers are packed and compressed
  - Saves bandwidth
- Push support
  - After first request – server can push data asynchronously
- Binary protocol

HTTP/2 Support

Java 9 provides a simple, standalone HTTP/2 based API

- Package: `jdk.incubator.http.*`
- HttpClient
  - Responsible for connection configuration (SSL support)
  - Configuring client for handling requests and web-sockets
- HttpRequest
  - Encapsulates all Http-Request information
  - Uses builders to configure request headers and content
- HttpResponse
  - Encapsulates all Http-Response information
  - Uses handlers to parse response body

HTTP/2 Support

HttpClient

- authenticator() : Optional< Authenticator>
- cookieManager() : Optional<CookieManager>
- followRedirects() : Redirect
  - Redirect ENUM stands for
    - ALWAYS – Always redirect
    - NEVER – Never redirect
    - SAME_PROTOCOL – redirect only to the same protocol
      - http to http
      - https to https
      - http to https and vice versa are not redirected
- executor() - Returns the default executor for this client
  - Each client uses a new dedicated executor

HTTP/2 Support

HttpClient

- newHttpClient() – static method to instantiate default client
- newWebSocketBuilder(URI uri, Listener listener)
  - WebSocket.Listener
    - onOpen()
    - onText(), onBinary(), onPing(), onPong()
    - onClose(), onError()
- In order to send HttpRequests:
  - send(…) – blocking operation
  - sendAsynchronously – non-blocking – using executor
    - Results with CompletableFuture

```
//creating new Http Client
HttpClient httpClient=HttpClient.newHttpClient();
```

HTTP/2 Support

HttpRequest

- Uses HttpRequest.Builder inner class to construct and define requests
  - newBuilder()
- HttpRequest.Builder
  - Provides pipelined methods for generating HttpRequest
  - uri(URI)
  - GET(), POST(), PUT(), DELETE()
  - setHeader(…), setHeaders(…)

```
//creating new Http Request
HttpRequest httpReq=HttpRequest.newBuilder()
                               .uri(new URI("http://google.com"))
                               .GET().build();
```

HTTP/2 Support

HttpResponse

- Wraps the HTTP result sent by the server
- Main operations:
  - body() : T – body content is handled by BodyHandler (later)
  - headers()
  - request() – each result holds a reference to the origin request
  - statusCode() : int – return HTTP result status

HTTP/2 Support

HttpResponse

- HttpResponse.BodyHandler
  - Parses HTTP response body into T body of HttpResponse
  - Is attached to a request on HttpClient submit
  - Main methods:
    - asByteArray(), asByteArrayConsumer(Consumer<byte[]>)
    - asFile(Path path)
    - asString(), asString(Charset charset)

HTTP/2 Support

Example

```
HttpClient httpClient=HttpClient.newHttpClient();
HttpRequest httpReq=HttpRequest.newBuilder().uri(new URI("http://google.com"))
                                                  .GET().build();
// Request Data:
System.out.println(httpClient.version());
System.out.println(httpReq.uri());
System.out.println(httpReq.method());

// Response Data:
HttpResponse<String> httpRes=httpClient.send(httpReq,
                                 HttpResponse.BodyHandler.asString());
System.out.println(httpClient.version());
System.out.println(httpRes.statusCode());
System.out.println(httpRes.headers());
System.out.println(httpRes.body());
```

HTTP/2 Support

Launching

- Strange thing here is that while everything compiles, when you launch you end up with:

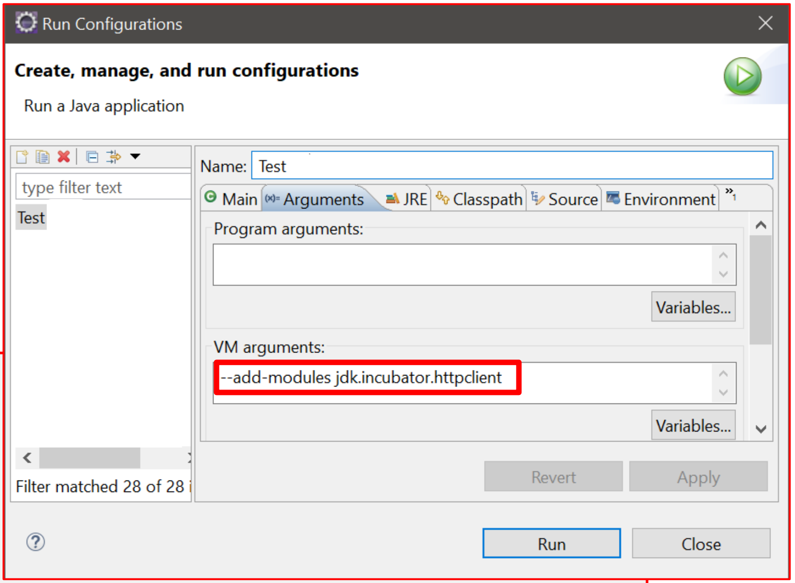
```
Exception in thread "main" java.lang.NoClassDefFoundError: jdk/incubator/http/HttpClient
           at web.http2.Test.main(Test.java:14)
Caused by: java.lang.ClassNotFoundException: jdk.incubator.http.HttpClient
           at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClassLoader.java:582)
           at java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(ClassLoaders.java:185)
           at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:496)
           ... 1 more
```

- Java 9 is modular…we'll discuss it later
- It happens since httpclient module is not part of java_base modules
- In order to launch correctly we must add httpclient module:
  - Done via –add-modules
  - Httpclient module name is: jdk.incubator

HTTP/2 Support

launching

Doing it right:



Create, manage, and run configurations

Run a Java application

Name: Test

Program arguments:

VM arguments:
--add-modules jdk.incubator.httpclient

Filter matched 28 of 28 i

```
WARNING: Using incubator modules: jdk.incubator.httpclient
Request Data:
HTTP_2
http://google.com
GET
Response Data:
HTTP_2
302
jdk.incubator.http.ResponseHeaders@4d339552
<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
<A HREF="http://www.google.co.il/?gfe_rd=cr&amp;dcr=0&amp;ei=n8nlWev9AdDb8Afl46v4BQ">here</A>.
</BODY></HTML>
```

Stack Walking

Easy way to walk through stack trace and enjoy streams while doing it!

Stack Walking

What’s wrong with StackTrace & StackTraceElements?

- Not easily accessible
- Eagerly generates full stack trace
  - Not efficient when you are looking for recent invocations in the stack..
- Heavy

StackWalker

- Easy to use
- Thread-safe (threads may share same stack trace information)
- Generates StackFrame **stream**
- Streams are lazily executed by nature

Stack Walking

StackWalker main operations:

- getInstance() methods for static allocations
  - Options can be assigned as a Set
    - StackWalker.Option.RETAIN_CLASS_REFERENCE
    - StackWalker.Option.SHOW_HIDDEN_FRAMES,
    - StackWalker.Option.SHOW_REFLECT_FRAMES
- forEach(Consumer<StackFrame> consumer)
  - Allows to eagerly iterate and manipulate StackFrames
- walk(Function<Stream<StackFrame>,?> function)
  - Returns a stream of StackFrames
  - Since stream are lazily executed – partial stack info can be loaded via limit()
  - Can be invoked multiple times to obtain new streams

Stack Walking

Example

```
public class Example {
        public void a() {
                b();
        }
        public void b() {
                c();
        }
        public void c() {
                StackWalker sw=StackWalker.getInstance();
                sw.forEach(System.out::println);
        }
}
```

```
Output:
stackwalking.Example.c(Example.java:15)
stackwalking.Example.b(Example.java:10)
stackwalking.Example.a(Example.java:6)
stackwalking.Test.main(Test.java:8)
```

Stack Walking

Example

```
StackWalker sw=StackWalker.getInstance();


//count elements
long size=sw.walk(frames->frames.count());


//obtain full stack trace
List<StackFrame> all = sw.walk(frames->frames.collect(Collectors.toList()));


//obtain 3 last stack trace elements
List<StackFrame> last3 = sw.walk(frames->frames.limit(3)
                                               .collect(Collectors.toList()));
```

ENVIRONMENTAL

Modular Java & Jlink

Eliminate JAR hell by creating modules & defining dependencies

Modular Java & Jlink

Jar & Classpath Hell

- We usually add jars to the classpath and hope for the best…
  - Worst scenario is using unwanted classes
  - Common problem is ClassDefNotFoundException

- We have no runtime information regarding jars containing which class

- The JRE handles all jars just as a single collection of classes

- No further meta-data to
  - Create more focused images (with only classes we need)
  - Reuse code without risking in classpath collisions

Modular Java & Jlink

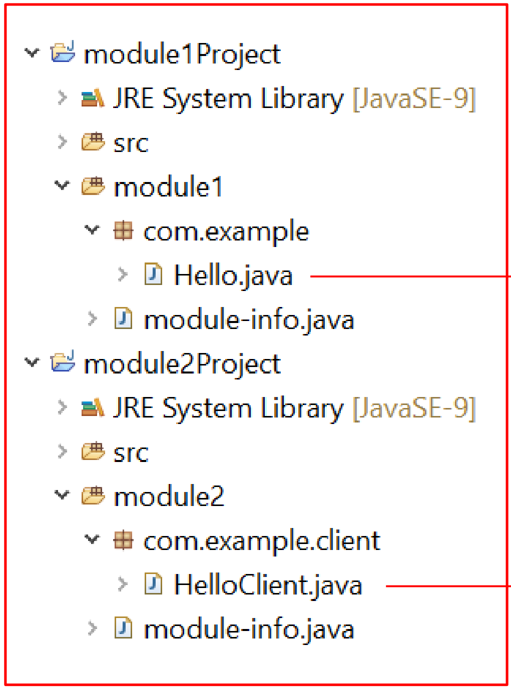
Modules

- ✓ Provides this meta-data layer
- ✓ Specifies exactly what is exposed to other modules
- ✓ Specifies module dependencies – so it can be checked along development
- ✓ Improves maintainability of large systems
- ✓ Allows creating focused, standalone images (via Jlink - later)

Modular Java & Jlink

Defining modules:

Project structure:



```
module1Project
    JRE System Library [JavaSE-9]
    src
    module1
        com.example
            Hello.java
        module-info.java
module2Project
    JRE System Library [JavaSE-9]
    src
    module2
        com.example.client
            HelloClient.java
        module-info.java
```

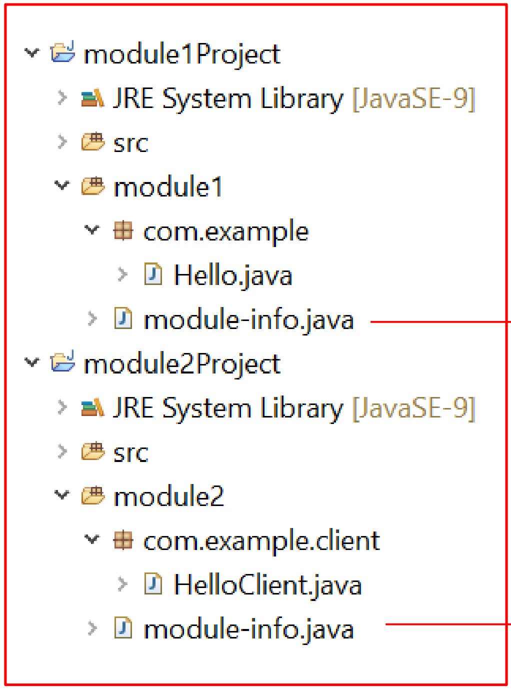
```
package com.example;

public class Hello {
        public void sayHello() {
                System.out.println("Hello!");
            }
}
```

```
package com.example.client;
import com.example.Hello;

public class HelloClient {
        public static void main(String[] args) {
                Hello h=new Hello();
                h.sayHello();
        }
}
```

Modular Java & Jlink

Defining modules:

Project structure:



```
module1Project
  JRE System Library [JavaSE-9]
  src
  module1
    com.example
      Hello.java
    module-info.java
module2Project
  JRE System Library [JavaSE-9]
  src
  module2
    com.example.client
      HelloClient.java
    module-info.java
```

```
module module1{
        exports com.example;
}
```

```
module module2{
        requires module1;
}
```

Modular Java & Jlink

Compiling modules:

- Basically done by IDE, but let’s see javac & java module support:

- For this example
  - we’ll use C:/temp directory as base directory
  - Sources are copies to base directory:
    - *C:/temp/module1/com/example/Hello.java*
    - *C:/temp/module1/module-info.java*
    - *C:/temp/module2/com/example/client/HelloClient.java*
    - *C:/temp/module2/module-info.java*

Modular Java & Jlink

JOHN BRYCE
Leading in IT Education
a matrix company

Compiling modules:

- Compiling module1 - C:/temp/mods/module1:

```
javac -d mods/module1 module1/module-info.java
module1/com/example/Hello.java
```

- Compiling module2 - C:/temp/mods/module2:
  - *--module-path* – specifies modules location
  - Modules can be packed as .mod files or remain expanded
  - Here we must specify module1 location since module2 depends on it

```
javac --module-path mods/module1 -d mods/module2
module2/module-info.java module2/com/example/client/HelloClient.java
```

Modular Java & Jlink

Running main class with modules:

- Running module2/com.example.client.HelloClient

  --*module-path* – specifies modules root location – C:/temp/mods

  -*m* specifies extended fully qualified class name to launch

```
java --module-path mods -m mods/module2/com.example.client.HelloClient


Output:
Hello!
```

Modular Java & Jlink

Jlink

- New utility that links different modules and creates a run-time image

- Run-time images
    - Includes java-base module
    - Contains only relevant modules and their dependencies
    - Highly relevant for
        - DevOps
        - Microservices

Modular Java & Jlink

Jlink

Parameters

--*module-path* – points to both java-base (jmods) & mods root directories

- default JAVA_HOME is: C:/Program Files/Java/jdk-9

--*add-modules* – included modules list

- dependencies are resolved automatically

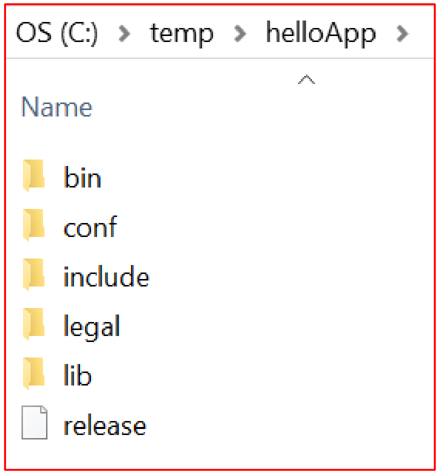
--*output* – generated image destination directory

```
jlink --module-path JAVA_HOME/jmods;mods --add-modules module2
      --output helloApp
```

Modular Java & Jlink

Jlink

- Generated image



OS (C:) › temp › helloApp ›

Name

- bin
- conf
- include
- legal
- lib
- release

- Running main class with image:

```
C:\temp\helloApp\bin>java -m module2/com.example.client.HelloClient

Output:
Hello!
```

Java REPL - Jshell

Use command line utility for rapid usage of Java code snippets

Java REPL - Jshell

Jshell utility

- Useful tool for prototyping and testing Java code snippets
- Each statement is evaluated and executed immediately
- Good for starting with Java
- Simplifies testing
- Is the first official Java REPL
  - Read – Eval – Print Loop
    - Read – accept expression from user
    - Eval – evaluating the expression as a var/method or read/invoke
    - Print – show the result of eval phase
    - Loop – wait for next user expression
  - A.K.A Interactive toplevel or Language shell

Java REPL - Jshell

Starting Jshell:

- Jshell is part of JDK9 installation and found under JAVA_HOME/bin directory
- To use it simply run the utility from the command:

```
C:\Program Files\Java\jdk-9\bin>jshell
|  Welcome to JShell -- Version 9
|  For an introduction type: /help intro

jshell>
```

- You may use –verbose in order to get more detailed prints

```
C:\Program Files\Java\jdk-9\bin>jshell -v
|  Welcome to JShell -- Version 9
|  For an introduction type: /help intro

jshell>
```

Java REPL - Jshell

Define and show vars:

```
jshell> 3+3
$1 ==> 6

jshell> int x=5
x ==> 5

jshell> x
x ==> 5
```

List vars:

```
jshell> /vars
|    int $1 = 6
|    int x = 5
```

Java REPL - Jshell

Define and show vars:

```
jshell> public void showAllVars(){System.out.println(x+","+$1);}
|  created method showAllVars()
```

Call method:

```
jshell> showAllVars()
5,6
```

List methods:

```
jshell> /methods
|    void showAllVars()
```

Java REPL - Jshell

Show history :

Includes

- Expressions & invocations
- Variable declaration
- Methods declaration

```
jshell> /list

   1 : 3+3
   2 : int x=5;
   3 : public void showAllVars(){System.out.println(x+","+$1);}
   4 : showAllVars()
```

Java REPL - Jshell

Using modules:

- Jshell must be started with --add-modules
  - May need to pre-set --module-path
- Jshell can use only what module exports!

```
C:\Program Files\Java\jdk-9\bin>jshell --module-path c:/temp/mods
                                       --add-modules module1
|  Welcome to JShell -- Version 9
|  For an introduction type: /help intro

jshell> import com.example.Hello

jshell> Hello h=new Hello()
h ==> com.example.Hello@6ee52dcd

jshell> h.sayHello()
Hello!
```

Java REPL - Jshell

Executing files

Predefined files:

- DEFAULT – loads commonly used imports
- JAVASE – imports all JavaSE packages
- PRINTING – adds print, println & printf as Jshell methods

```
C:\Program Files\Java\jdk-9\bin>jshell –startup file
```

Using pre-defined files in Jshell – example:

```
C:\Program Files\Java\jdk-9\bin>jshell --startup PRINTING
|  Welcome to JShell -- Version 9
|  For an introduction type: /help intro

jshell> printf("There are %d melons on the %s",2,“ shelf")
There are 2 melons on the shelf
```

Java REPL - Jshell

Closing Jshell session

```
jshell> /exit
|  Goodbye
```

Multi-version JARS

Generate JARs that can detect JSE version and choose the correct class accordingly

Multi-version JARS

Multi-versioning in Java

Goal here is to be able to generate JARS that

- Contains duplicated classes
- Each class has it own version-sensitive implementation
- Java runtime will choose the correct class according to the current hosting runtime

Multi-version JARS

Example:

```
public class VersionDependant {
        public List<String> getList(){
                System.out.println("Using List.of() - Java 9");
                return List.of("Hello","To","Multiversion","Jars");
        }
}
```

```
public class VersionDependant {
        public List<String> getList(){
                System.out.println("Using Arrays.asList() - Java 8");
                return Arrays.asList("Hello","To","Multiversion","Jars");
        }
}
```

```
public class Main {
        public static void main(String[] args) {
                VersionDependant v=new VersionDependant();
                System.out.println(v.getList());
        }
}
```

Multi-version JARS

Example:

First, we have to compile the project for each supported version:

- Compile with JDK8 compiler into lib\jdk8 directory
  - Assume we have placed source files in jdk8 directory

```
java -version
java version "1.8.0_71"
javac -d lib\jdk8 jdk8\*.java
```

- Compile with JDK9 compiler into lib\jdk9 directory
  - Assume we have placed source files in jdk9 directory

```
java -version
java version "9"
javac -d lib\jdk9 jdk9\*.java
```

Multi-version JARS

Example:

- Now, we use jar (JDK9) utility in order to create a multi-versioned JAR

  --*create* – generates a file

  --*file* – specifies generated jar location

  -*C* – points to classes location

  --*release* – specifies alternative version followed by –C

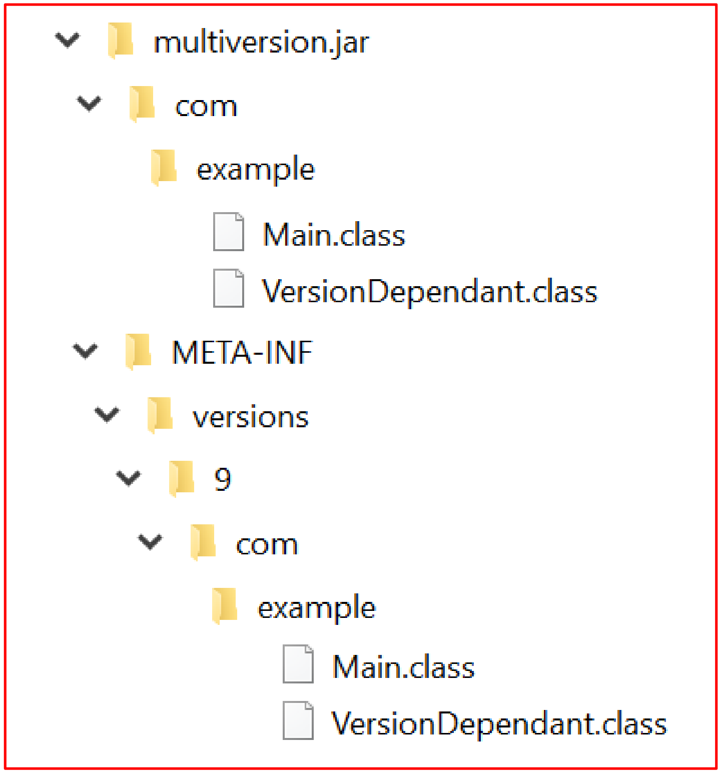
```
java -version
java version "9"

jar --create --file multiversion.jar -C lib\jdk8 . --release 9 -C lib\jdk9 .
Warning: entry META-INF/versions/9/com/example/Main.class contains a class
that is identical to an entry already in the jar
```

- Now we got **multiversion.jar** located at our working directory

Multi-version JARS

Example:

Multi-version JAR infrastructure:



```
multiversion.jar
  com
    example
      Main.class
      VersionDependant.class
  META-INF
    versions
      9
        com
          example
            Main.class
            VersionDependant.class
```

Multi-version JARS

Example:

- Launching with JRE8:

```
java -version
java version "1.8.0_71"

SET classpath=.\multiversion.jar

java com.example.Main
Using Arrays.asList() - Java 8
[Hello, To, Multiversion, Jars]
```

- Launching with JRE9:

```
java -version
java version "9"

SET classpath=.\multiversion.jar

java com.example.Main
Using List.of() - Java 9
[Hello, To, Multiversion, Jars]
```

G1 Made Default GC

Not in 7…

Not in 8…

It’s about time….

G1 Made Default GC

What are the defaults in earlier Hotspots ?

- New region
  - minor GC
  - name : Scavenge GC
  - parallel capable

- Old region
  - Full GC
  - name : CMS – Concurrent Mark Sweep
  - Parallel capable

- Recent minor versions of Java8 comes with G1 as default

G1 Made Default GC

What is G1?

According to Oracle:

- Targeted for multiprocessor machines with a large amount of memory
- Aims to provide the best balance between latency and throughput
- Main application related assumptions:
    - Heap sizes up to 10 GBs or larger
    - Rates of object allocation and promotion that can vary dramatically
    - pause-time target goals that aren’t longer than a few hundred millis

How does it work in general?

- Uses compaction
- Maintains areas with most phantom objects
- Collects in these areas first while leaving less to compact

G1 Made Default GC

More on G1

Instead of sweeping – compacts and defragments

Also generational (acts on NEW & OLD regions)

Based on concurrent / parallel behavior

Like CMS, a low-pause GC – but better

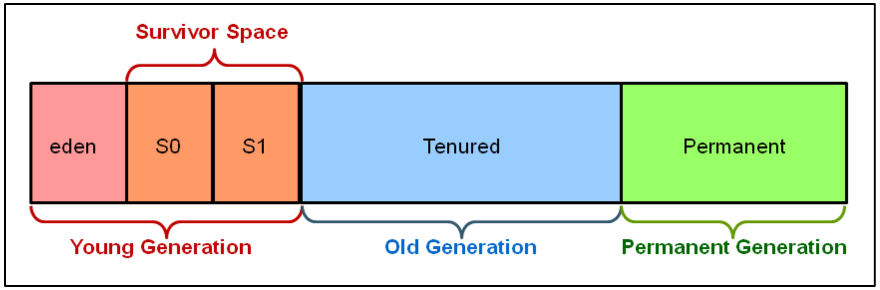
Usage: --XX: +UseG1GC

In Java 6 activate feature first with: -XX:+UnlockExperimentalVMOptions

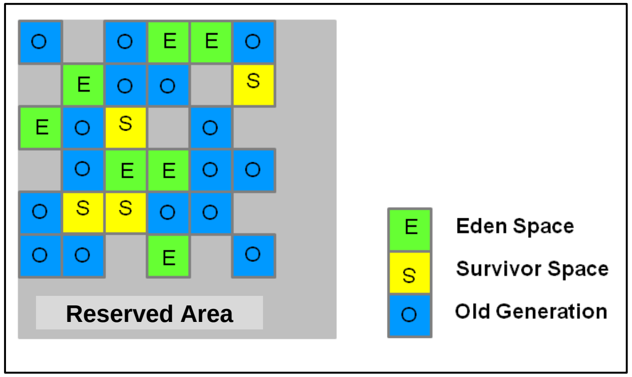
G1 Made Default GC

G1 – How does it work ?

Previous GCs (serial, parallel, CMS) manages objects in predefined memory regions:

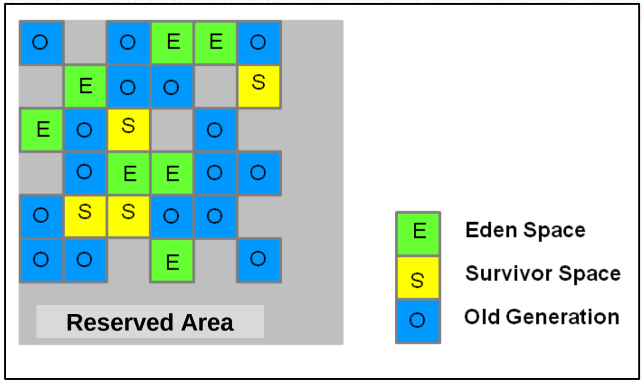


G1 manages objects like this:



G1 Made Default GC

G1 – How does it work ?



- heap is partitioned into a set of equal-sized heap regions
- Each partition has a role (Eden / Survivor Space / Old)
- Step1: G1 performs a concurrent global marking of unreachable objects
- Step 2: G1 tracks areas where most phantom objects exist
- Step 3: G1 collects in these areas FIRST(!)
- Step 4: G1 compacts the remaining object in these areas (not much left….)

Reserved Area

- for old region
- helpful in cases of huge allocations
- Size may be set as heap percent

JAVA 10

Java 10 – CODE & API

Local Variable Type Interface

Local vars

- Local assignments may now be expressed like this:

```
var map = new HashMap<String,Integer>();
```

- Var types are evaluated by the compiler – so no runtime effect, syntax only
- Correct usage is for more readable code:

```
var list = new ArrayList<String>();
var map = new HashMap<String,Integer>();
var number = 100;
```

- Abusing this feature will be something like this, which compiles…

```
var data = getData();
```

GC Interface

Custom GCs

- Until 10:
  - Implementing your GC requires handling all parts of memory (new, old, metaspace)
- After 10:
  - You may implement region specific GC and plug them with the available GC

Java 10 – ENVIRONMENTAL

Application Class Data Sharing

AppCDS

- Class are loaded natively on Metaspace
- When running an application you can
  - Specify classes to be archived – XX:DumpLoadedClassList
  - create an archive – Xshare:dump
- Other applications may use archive and save jars scanning…
  - Xshare:on
- In any case the -XX:+UseAppCDS flag must be specified

Application Class Data Sharing

AppCDS

- Step 1: Create class list to archive
  - generates classes.list file – which can be created manually as well:

```
$ java
    -XX:+UseAppCDS
    -XX:DumpLoadedClassList=classes.list
    -jar myApp.jar
```

```
java/lang/Object
java/lang/String
java/io/Serializable
java/lang/Comparable
java/lang/CharSequence
```

Application Class Data Sharing

AppCDS

- Step 2: Generate archive from a given class list
  - generates myApp-cds.jsa file which caches all classes specified in classes.list

```
$ java
    -XX:+UseAppCDS
    -Xshare:dump
    -XX:SharedClassListFile=classes.list
    -XX:SharedArchiveFile=myApp-cds.jsa
    --class-path myApp.jar
```

Application Class Data Sharing

AppCDS

- Step 3: use the archive in other JVM instances:

```
$ java
    -XX:+UseAppCDS
    -Xshare:on
    -XX:SharedArchiveFile=myApp-cds.jsa
    -jar app.jar
```

- You can log sharing activity to a file:

```
$ java -Xshare:on … -Xlog:class+load:file=cds.log -jar app.jar
```

  - In cds.log you can see class load activity and identify whether classes are loaded from the cache:

```
> [0.049s][info][class,load] org.java10.cds.HelloCDS source:
    shared objects file
```

Full Parallel GC G1

Parallel G1

- G1 works mostly in an ‘incremental’ way when performing full GC
  - Allows to clean on most reclaimable space blocks first
  - Means that GC never really performs full-GC…

- In some conditions where lots of objects must be cleaned
  - G1 switches to CMS (‘full’ mode) to perform a full GC
  - Before 10: G1 - CMS was single threaded – risk in long pause time
  - From 10: G1 – CMS is parallel (uses the same number of threads set with :GCThreads)

Alternative Heap Allocation

Heap Allocation

- Java uses DRAM by default
  - DRAM
    - Dynamic random-access memory
- There are strong alternative chips like NVDIMM
  - NVDIMM - non-volatile dual in-line memory module
- In order to specify alternative heap allocation in Java 10:
  - -XX:AllocateHeapAt=<path>

JAVA 11

Java 11 – REMOVED

Removed APIs & Tools

- The death of JEE – removal of all relevant APIs from 11 JDK:
  - java.activation (JAF)
  - java.xml.ws.annotation (Common Annotations)
  - java.corba (CORBA)
  - java.transaction (JTA)
  - java.se.ee (Aggregator module for the six modules above)
  - jdk.xml.ws (Tools for JAX-WS)
  - jdk.xml.bind (Tools for JAXB)
  - javax.mail

Removed APIs & Tools

- No more built-in support for XML based Web-Services (JAXP)
  - java.xml.ws in not included
  - JAX-WS tools are not included:
    - wsgen
    - wsimport

Removed APIs & Tools

- JAXB – Use Maven/Gradle dependencies in order to keep using in JDK11:

```
<dependency>
   <groupId>javax.xml.bind</groupId>
   <artifactId>jaxb-api</artifactId>
   <version>2.3.0</version>
</dependency>
<dependency>
   <groupId>com.sun.xml.bind</groupId>
   <artifactId>jaxb-core</artifactId>
   <version>2.3.0</version>
</dependency>
<dependency>
   <groupId>com.sun.xml.bind</groupId>
   <artifactId>jaxb-impl</artifactId>
   <version>2.3.0</version>
</dependency>
```

- No more JAXB utilities:
  - schemagen
  - xjc

Removed APIs & Tools

- No more built-in support for CORBA
  - java.xml.ws in not included
  - CORBA tools are not included:
    - idlj
    - orbd
    - servertool
    - Tnamesrv
  - rmic was updated and no longer supports –idlj & -iiop

Removed APIs & Tools

- Java WebStart was deprecated in 9 and removed in 11
- JRE Auto-update was removed
- Only JDK is shipped. No more JRE/Server JRE available
  - Jlink allows creating thin modules with minimal requirements
- Java Mission Control – not included in the JDK
  - Is now a separate download

Removed APIs & Tools

Scripting Engine

- Nashorn JavaScript Engine – deprecated
- jjs utility – deprecated as well

Java 11 – Environment

AppCDS support for modules

AppCDS supports Modules

CDS supports --**module-path** in order to specify classes included in modules rather than jars on the class-path

```
$ java
    -XX:+UseAppCDS
    -XX:DumpLoadedClassList=classes.list
    --module-path myModule
```

```
$ java
    -XX:+UseAppCDS
    -Xshare:dump
    -XX:SharedClassListFile=classes.list
    -XX:SharedArchiveFile=myApp-cds.jsa
    --module-path myModule
```

Single File Launch

Single File Launch

- Allows to launch application using main class source
- Limitation: If multiple classes are used – all must be on the same file
- May use modules via –add-modules

```
java HelloWorld.java
```

Compiling Threads – Lazy Allocation

Compiling Threads

- Since Java8 Tiered Compilation is done by default
- Number of compiling threads is traditionally set according to available CPUs
- In multiple CPU machines the number of compiling threads might be too high
- Idle threads still requires system resources and reduce performance

- JDK11 offers a new flag which allows lazy allocation of compiling threads

```
-XX:+UseDynamicNumberOfCompilerThreads
```

G1 Update

Parallel Reference Processing

- G1 pauses includes reference processing
- -XX:-ParallelRefProcEnabled allows to set parallel processing

- JDK11 set -XX:-ParallelRefProcEnabled to ‘true’ by default

ZGC – Experimental GC

ZGC

- Goals
  - Handle multi-terabyte heaps
  - Limit the GC pause time to no more than 10 milliseconds
  - Pause times do not increase with the heap or live-set size
  - Simplify tuning
- Characteristics:
  - Concurrent (for marking, re-allocation/compaction, reference processing)
  - Region-based
  - Compacting
  - NUMA-aware (memory location relative to the processor)
  - Using colored pointers
  - Using load barriers
  - Currently supported on Linux/x64

ZGC – Experimental GC

ZGC

- Classic phases:
  1. Reallocation/Completion phases
  2. Fixing pointers pointing into the reclaimed/reused regions
  3. Reuse memory

- ZGC approach:
  1. Reallocation/Completion phases while Reusing memory
  2. Fixing pointers pointing into the reclaimed/reused regions

  - Helps in keeping the overall heap overhead down
  - Eliminates the need in mark-compact algorithm to handle full GC

ZGC – Experimental GC

ZGC

- How can ZGC reuse memory while pointers haven't been fixed yet?

- ZGC Load Barriers uses Colored Pointers
  - Load Barriers
    - Manages Java threads access to pointers
    - ZGC uses small numbers of simple GC barriers to reduce overhead
  - Colored pointers
    - holds information regarding action needs to be taken before allowing Java threads to use the pointer
    - Currently information related to marking and relocating

- This is how pointers with deprecated addresses can be blocked by Load Barriers while their physical addresses are being reused
- This 'communication' between Load-Barrier and Colored Pointers can be extended

ZGC – Experimental GC

ZGC

- In order to use ZGC:

```
-XX:+UnlockExperimentalVMOptions -XX:+UseZGC -Xmx<size> -Xlog:gc
```

Java 11 – Syntax & APIs

Local Variable for LAMBDA

Extended Local vars support for LAMBDA

- Good reason to use ‘var’ is when setting annotations on LAMBDA variables:

```
list.stream().map((var s)->s.length()))...
```

```
list.stream().map((@NotNull var s)->s.length()))...
```

java.lang.String

String new methods

- isBlank() – returns ‘true’ if string is empty or contains whitespaces only
- lines() – returns a Stream<String> and streams the string according to line separator

```
httpRes.body().lines().forEach(System.out::println);
```

- repeat(int times) – returns a String, original value repeated according to ‘times’
- strip() – removes whitespaces from head & tail of the string
- stripLeading() – removes whitespaces at the head of the string
- stripTrailing – removes whitespaces at the tail of the string

Predicate.not()

Predicate static method not() – Predicate::not

- Useful since most test cases are ‘positively’ evaluated (isNull, isEmpty, ,isBlank, isAlive…)

```
list.stream().filter(Predicate.not(String::isBlank))...
```

```
//import static Predicate
list.stream().filter(not(String::isBlank))..
```

Incubator is now java.net

`jdk.incubator.http.*` is now **`java.net.*`**

- No need http module dependency for using HTTP2 client API

```
import java.net.*;
import java.net.http.*;
...
HttpClient httpClient=HttpClient.newHttpClient();
HttpRequest httpReq=HttpRequest.newBuilder().uri(new URI("http://google.com"))
                                                  .GET().build();
// Request Data:
System.out.println(httpClient.version());
System.out.println(httpReq.uri());
System.out.println(httpReq.method());

// Response Data:
HttpResponse<String> httpRes=httpClient.send(httpReq,
                                 HttpResponse.BodyHandler.asString());
System.out.println(httpClient.version());
System.out.println(httpRes.statusCode());
System.out.println(httpRes.headers());
System.out.println(httpRes.body());
```

Reference.clone()

Reference.clone()

- Untill 11 – inherits Object.clone()

  means that if your Reference implements Clonable – no exception is thrown

- JDK11 – Refrence.clone() always throws NotClonableException

Thank You!