1) Jshell(REPL):

REPL: REPL is an interactive shell into which we can enter commands and have these immediately executed and the results displayed.

Problem:

- To write simple HelloWorld program we need to write lots of things:

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

- New user might get confused with keywords such as Class, public and static.
- We need to write lots of code to test code snippets.

Solution:

- Use Jshell.
- No need to write above code to just print hello world.
- Easy to to test code snippets.

```
jshell>

jshell> System.out.println("Hello World");
Hello World
```

```
jshell> 10+5
$1 ==> 15

jshell> 10/5
$2 ==> 2

jshell> 10/3
```

```
jshell> 10+5
$1 ==> 15
```

```
jshell> $1
$1 ==> 15

jshell> $1=20
$1 ==> 20

jshell> System.out.print("$1 value now = "+$1)
$1 value now = 20
jshell>
```

```
jshell> /imports
|    import java.io.*
|    import java.math.*
|    import java.net.*
|    import java.nio.file.*
|    import java.util.*
|    import java.util.concurrent.*
|    import java.util.function.*
|    import java.util.prefs.*
|    import java.util.regex.*
|    import java.util.stream.*
jshell>
```

```
jshell> Math.max(5,10)
$11 ==> 10

jshell> Math.min(5,10)
$12 ==> 5
```

2) JPMS:

Problem: Jar Hell!

- When .class file is not found, entire application will stop (NoClassdefFounderror).
- jvm is throwing NoClassdefFounderror exception at runtime(after completing n statements), it is not giving error at starting.
- version conflicts(if same .class is present in two different jar files, jvm will pick (left to right) file, so it can produce version conflicts.)

Solution:

- At beginning(Compile time), jvm will check for module availability. it will throw no module found error at beginning of execution.
- No NoClassdefFounderror exception at run time.
- No version conflict(with the help of module config file).

JPMS:Java Platform Module System.

Java SE 9 there is a new structural element – modules.

- a class is a container of fields and methods
- a package is a container of classes and interfaces
- a module is a container of packages

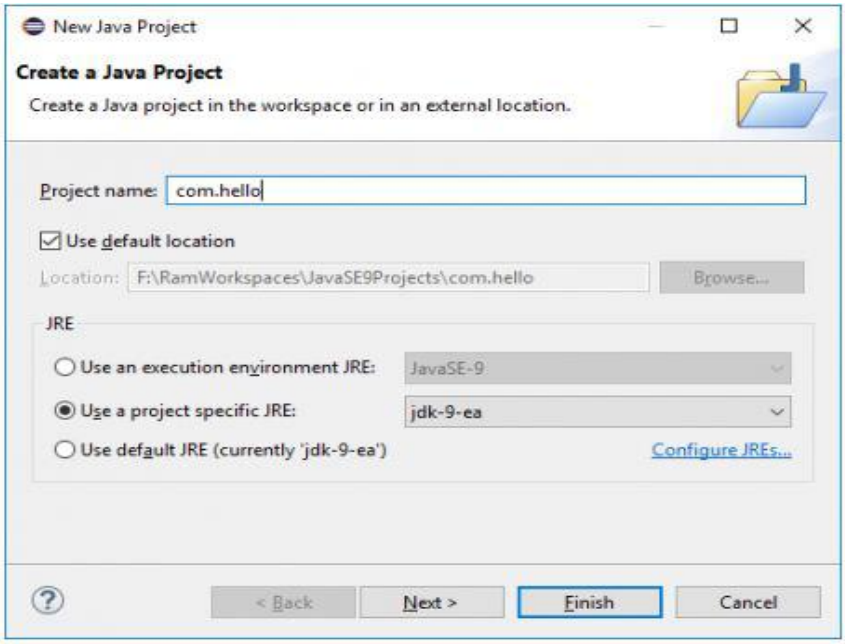
It is a set of related Packages, Types (classes, abstract classes, interfaces etc) with Code & Data and Resources.

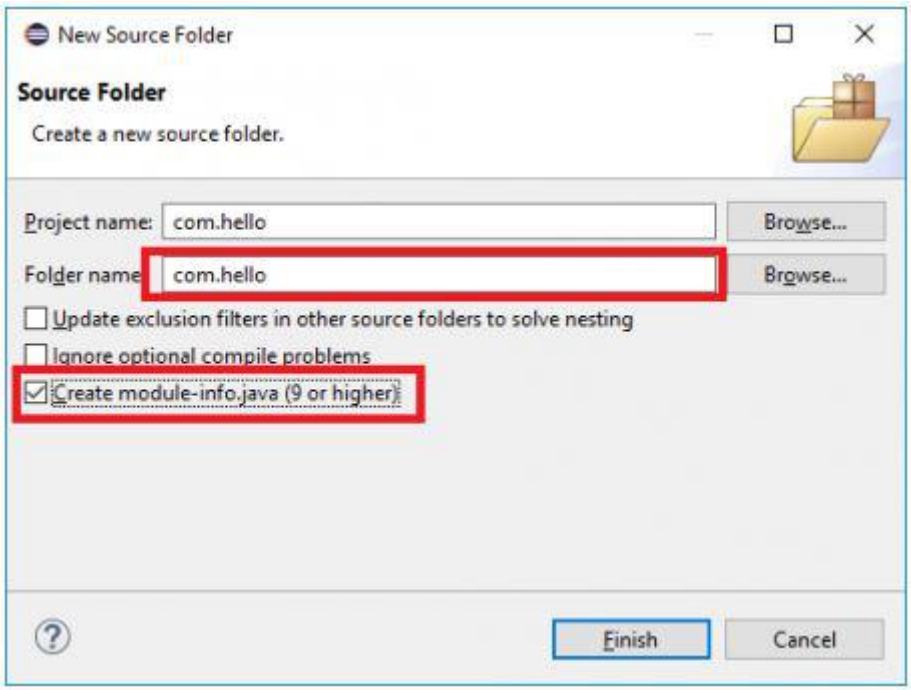
With Java 9, a developer can express that a package cannot be seen by other modules – ie. a package can be hidden within a module.

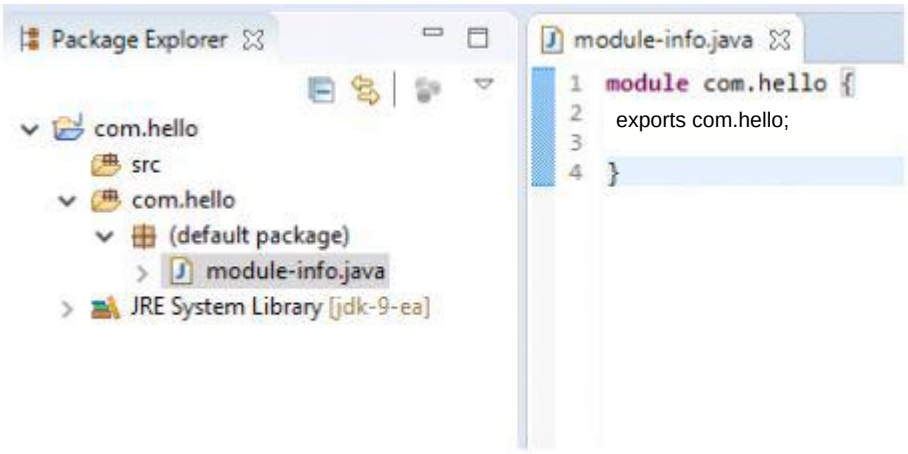
The module-info.java file contains the instructions that define a module.

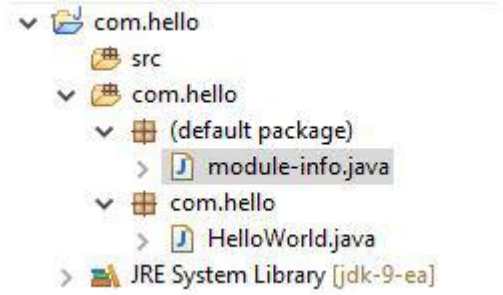
```
module org.myapp.mymodule
{
        requires org.myapp.myothermodule; // this is a module name, not a
                                                                //package name
        exports org.myapp.mypkg2;// this is a package name, not a module

      name
}
```

Currently, Eclipse requires you to create a separate project for each module (e. g. because each module has its own Java Build Path).

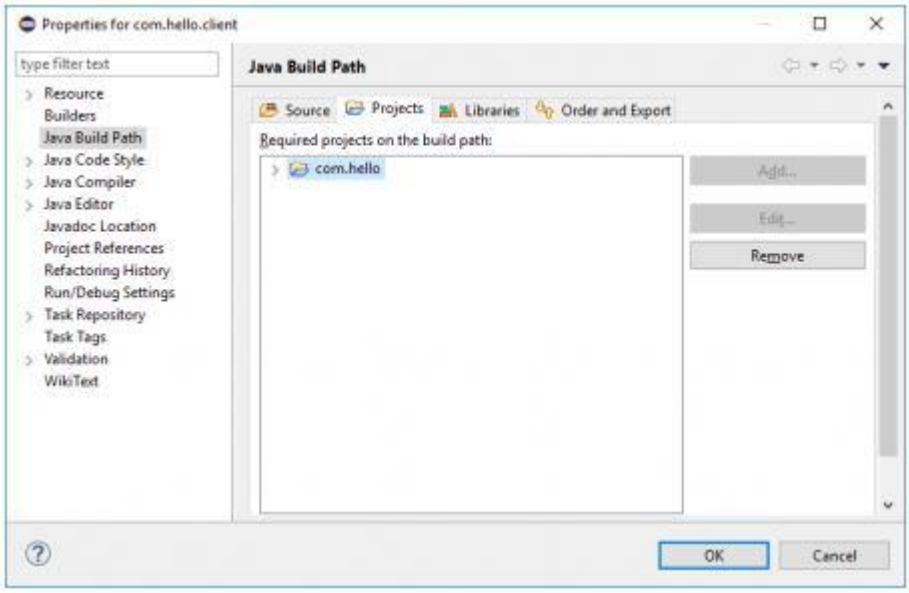



New Source Folder
Source Folder
Create a new source folder.
Project name: com.hello
Browse...
Folder name: com.hello
Browse...
Update exclusion filters in other source folders to solve nesting
Ignore optional compile problems
Create module-info.java (9 or higher)
Finish
Cancel


Package Explorer
com.hello
src
com.hello
(default package)
module-info.java
JRE System Library [jdk-9-ea]
module-info.java
1 module com.hello {
2 exports com.hello;
3
4 }


com.hello
src
com.hello
(default package)
module-info.java
com.hello
HelloWorld.java
JRE System Library [jdk-9-ea]


Package Explorer
com.hello
com.hello.client
src
com.hello.client
(default package)
module-info.java
com.hello.client
HelloWorldClient.java
JRE System Library [jdk-9-ea]
module-info.java
HelloWorldClient.java
1 module com.hello.client {
2
3 requires com.hello;
4
5 }
com.hello cannot be resolved to a module
Press 'F2' for focus



3) JLink:

Problem:

- To run 1kb of HelloWorld program , client machine requires 400mb of jre.
- Before java 9, jre was containing rt.jar file (which includes all required classes to run our programs).
- rt.jar consist nearly 4000 files ! ..size around 60M
- So the problem with the default JRE is that it executes the all predefined .class files whether you want to or not.
- So java was not suitable for Iot development as on Iot devices we don't have high memory.

Solution:

- With java 9 ,we can create our own jre (include only what is required).
- Suitable for Iot Development.
- Jlink is Java's new command line tool through which we can create our own customized JRE.

For example if I have an application called testapp in a directory called exampledir, and I want to create a run-time image in a directory called outputdir, I can use the following command:

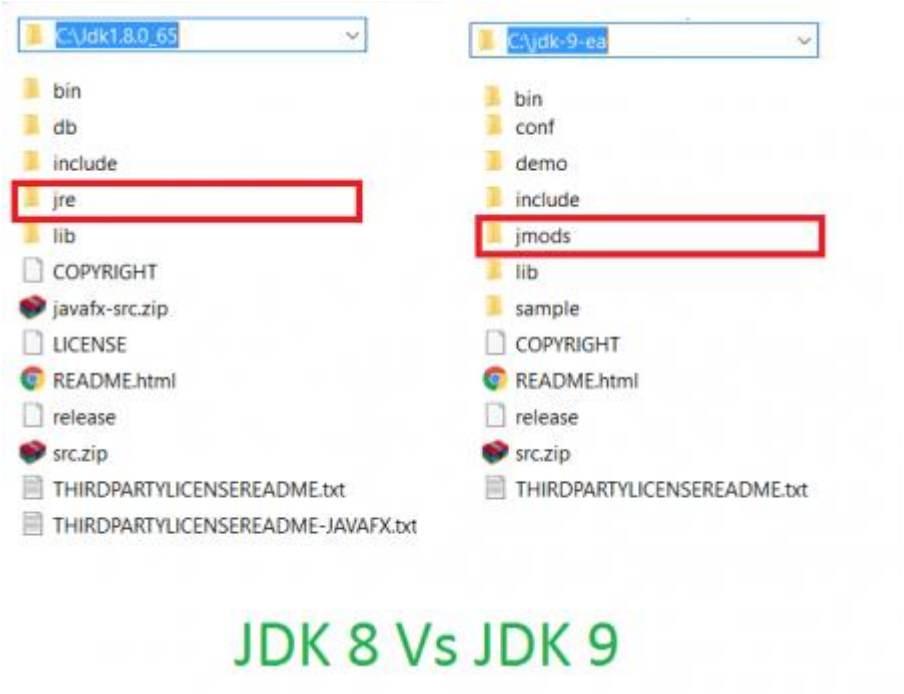
```
jlink --module-path exampledir:$MODS --add-modules
com.example.testapp --limit-modules com.example.testapp --output
outputdir
```

–module-path specifies where to find the modules for the JDK.

–add-modules adds the modules we need (in this example our app and nothing else).

–limit-modules limits to just the modules that we know our application needs.

–output is this directory where the run-time image will be



4) HTTP/2 Client:

HTTP client is used to send http request to server and get http response .

```
private static void sendGET() throws IOException {
        URL obj = new URL(GET_URL);
        HttpURLConnection con = (HttpURLConnection) obj.openConnection();
        con.setRequestMethod("GET");
        con.setRequestProperty("User-Agent", USER_AGENT);
        int responseCode = con.getResponseCode();
        System.out.println("GET Response Code :: " + responseCode);
        if (responseCode == HttpURLConnection.HTTP_OK) { // success
                BufferedReader in = new BufferedReader(new InputStreamReader(
                                con.getInputStream()));
                String inputLine;
                StringBuffer response = new StringBuffer();

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

```
                // print result
                System.out.println(response.toString());
        } else {
                System.out.println("GET request not worked");
        }

}
```

HTTP/2 is the newest version of the HTTP Protocol. The existing version's (HTTP 1.1) problems are eliminated in this newer version.

Problem: HTTP 1.1

- We cannot have more than six connections open at a time, so every request has to wait for the others to complete
- At a time only one request can be send per connection.
- Support for only text data.
- Always works on blocking mode(synchronous mode).so we have to wait till response comes.

Options:

-Apache HTTP client
-google http client

Solutions: HTTP/2

- Multiple request per connection.
- Support for text and binary.
- **Synchronous mode and asynchronous mode.**

The API consists of 3 core classes:

- **HttpRequest** represents the request to be sent via the HttpClient.
- **HttpClient** behaves as a container for configuration information common to multiple requests.
- **HttpResponse** represents the result of an HttpRequest call.

```
jshell> import java.net.http.*

jshell> import static java.net.http.HttpRequest.*

jshell> import static java.net.http.HttpResponse.*

jshell> URI uri = new URI("http://rams4java.blogspot.co.uk/2016/05/java-
news.html")
uri ==> http://rams4java.blogspot.co.uk/2016/05/java-news.html

jshell> HttpResponse response =
HttpRequest.create(uri).body(noBody()).GET().response()
response ==> java.net.http.HttpResponseImpl@79efed2d

jshell> System.out.println("Response was " + response.body(asString()))
```

5) Process API Updates:

Java 9 adds several new methods to the abstract Process class that let you identify direct child or descendant processes, obtain this Process's PID, return a snapshot of information about this Process, and more.

Problem:

- Communicate with OS was very difficult.
- We had to write very complex code.
- To do this we had to use 3rd party libs and native libs.

Solutions:

- Easy to use.
- Obtaining process Information easily.
- Easy to create,delete process.

87

This is another approach to parse the the process list from the command "**ps -e**":

```
try {
    String line;
    Process p = Runtime.getRuntime().exec("ps -e");
    BufferedReader input =
            new BufferedReader(new InputStreamReader(p.getInputStream()));
    while ((line = input.readLine()) != null) {
        System.out.println(line); //<-- Parse data here.
    }
    input.close();
} catch (Exception err) {
    err.printStackTrace();
}
```

If you are using Windows, then you should change the line: "Process p = Runtime.getRun..." etc... (3rd line), for one that looks like this:

```
Process p = Runtime.getRuntime().exec
    (System.getenv("windir") +"\\system32\\"+"tasklist.exe");
```

Hope the info helps!

```
public class ProcessApiExample {
    public static void main(String[] args) {
        ProcessHandle currentProcess = ProcessHandle.current();     // Current processhandle
        System.out.println("Process Id: "+currentProcess.pid());    // Process id
        System.out.println("Direct children: "+ currentProcess.children()); // Direct children of the process
        System.out.println("Class name: "+currentProcess.getClass());       // Class name
        System.out.println("All processes: "+ProcessHandle.allProcesses()); // All current processes
        System.out.println("Process info: "+currentProcess.info());         // Process info
        System.out.println("Is process alive: "+currentProcess.isAlive());
        System.out.println("Process's parent "+currentProcess.parent());  // Parent of the process
    }
}
```

So after java 9 , java is not only programmer friendly language but processor friendly language also!

6) Private Methods Inside Interface:

Before Java 1.8:

- Interface contains only abstract methods(public).
- No concrete methods in Interface.

After Java 1.8:

We can have concrete methods in Interface.
defaults methods:

- Without affecting existing functionality we can add methods to existing interface.

- default void methodName()
  {
  sop();
  }

After java 9:

private Methods: code re-usability

- if we have multiple default methods in interface
- We can separate common code in private methods
- No effect on implementing classes.
- private String methodName()
  {
  sop();
  }

7) **Try with Resources:**

Before java 1.6:

- If we want to close resource, we ware writing code in finally block.
- We had to manage Opening and Closing of Resource.

After Java 1.7:

- try block with resources was introduced.
- **Rule: resource variable should be local to try block.**

```
void testARM_Before_Java9() throws IOException{
 BufferedReader reader1 = new BufferedReader(new FileReader("journaldev.txt"));
 try (BufferedReader reader2 = reader1) {
   System.out.println(reader2.readLine());
 }
}
```

After Java 1.9:

```
void testARM_Java9() throws IOException{
 BufferedReader reader1 = new BufferedReader(new FileReader("journaldev.txt"));
 try (reader1) {
   System.out.println(reader1.readLine());
 }
}
```

8) Factory Methods to create immutable collections:

Java 9 has created factory methods for creating immutable Lists, Sets, Maps, and Map. Entry Objects. These utility methods are used to create empty or non-empty collection objects.

Before Java 9:
create immutable collections:

```
List<String) list=new ArrayList<String>();
list.add("string 1");
list.add("string 2");
list.add("string 3");

list= Collections.unmodifiableList(list);
```

After Java 9:

Empty List Example

```
List immutableList = List.of();
```

Non-Empty List Example

```
List immutableList = List.of("one","two","three");
```

Empty Map Example

```
jshell> Map emptyImmutableMap = Map.of()
emptyImmutableMap ==> {}
```

Non-Empty Map Example

```
jshell> Map nonemptyImmutableMap = Map.of(1, "one", 2, "two", 3, "three")
nonemptyImmutableMap ==> {2=two, 3=three, 1=one}
```

9) Stream API enhancements:

What is a Stream?

- A Stream is a sequence of elements and supports a set of aggregate operations on them easily.

In Java SE 9, Oracle Corp has added the followng useful new methods to java.util.Stream interface.

- dropWhile
- takeWhile

Java 8:

```
public class MyClass {

    List<Integer> myList= new ArrayList();

    public MyClass() {
        myList.add(5);
        myList.add(15);
        myList.add(10);
        myList.add(20);
        myList.add(50);
    }
```

```
  private void beforeStream()
  {
      List<Integer> list2 =new ArrayList<>();
      for(Integer i:myList)
      {
          if(i % 2 == 0)
          {
              list2.add(i);
          }
      }

  }
```

```
private void afterStream()
{
    Stream<Integer> stream= myList.stream();
    stream = stream.filter(i -> i % 2 == 0);
    List<Integer> list2 = stream.collect(Collectors.toList());
}
```

```
private void afterStream()
{
    List<Integer> list2 = myList.stream().filter(i -> i % 2 == 0)
            .collect(Collectors.toList());
}
```

Java 9:

```
jshell> Stream<Integer> stream = Stream.of(1,2,3,4,5,6,7,8,9,10)
stream ==> java.util.stream.ReferencePipeline$Head@55d56113

jshell> stream.takeWhile(x -> x < 4).forEach(a -> System.out.println(a))
1
2
3
```

```
jshell> Stream<Integer> stream = Stream.of(1,2,3,4,5,6,7,8,9,10)
stream ==> java.util.stream.ReferencePipeline$Head@55d56113

jshell> stream.dropWhile(x -> x < 4).forEach(a -> System.out.println(a))
4
5
6
7
8
9
10
```

`filter` will remove all items from the stream that do not satisfy the condition.

`takeWhile` will abort the stream on the first occurrence of an item which does not satisfy the condition.

e.g.

```
Stream.of(1,2,3,4,5,6,7,8,9,10,9,8,7,6,5,4,3,2,1)
    .filter(i -> i < 4 )
    .forEach(System.out::print);
```

will print

> 123321

but

```
Stream.of(1,2,3,4,5,6,7,8,9,10,9,8,7,6,5,4,3,2,1)
    .takeWhile(i -> i < 4 )
    .forEach(System.out::print);
```

will print

> 123