Welcome
To
Java 9 Upgrade

By
Srikanth Pragada

Make sure your phone doesn’t disturb us

Evolution of Java Language

Java 1.1

- ❑ Nested classes
- ❑ JDBC
- ❑ Java Beans

Java 1.2

- ❑ Collections Framework
- ❑ Swing Library

Java 1.3

Java 1.4

- ❑ Assertions

Java 5

- ❑ Auto-boxing and Auto-unboxing
- ❑ Annotations
- ❑ Generics
- ❑ Enhanced for loop
- ❑ Enumeration
- ❑ Static import
- ❑ Variable arguments

Evolution of Java Language

Java 6

- ❑ Scripting API

Java 7

- ❑ Try with resources
- ❑ Underscore in numeric literals and binary literals
- ❑ Diamond operator (<>)
- ❑ Support for strings in switch
- ❑ Multi-catch and more precise rethrow
- ❑ New IO API

Java 8

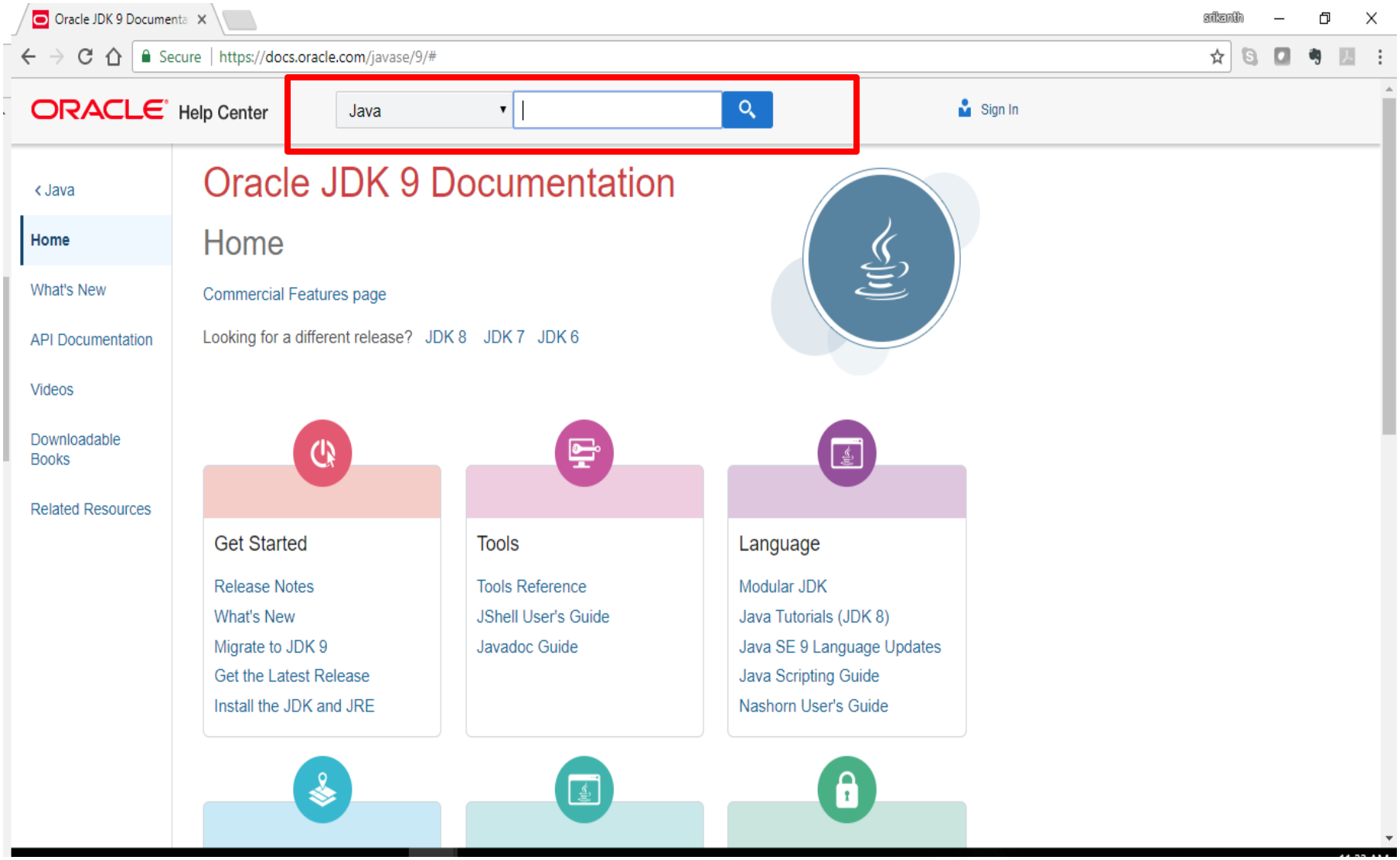
- ❑ Default and static methods in Interface
- ❑ Lambda Expressions
- ❑ Streams
- ❑ New Date API

What's New In Java 9

- ❑ Jshell (REPL)
- ❑ Language Enhancements
- ❑ New factory methods in Collections
- ❑ Optional class improvements
- ❑ Stream API Improvements
- ❑ Compact Strings
- ❑ LocalDate improvements
- ❑ Java Platform Module System (JPMS)

Released in Sep, 2017

Search facility in Java 9 Documentation



jShell – Java Shell

- Also known as REPL (Read Evaluate Print Loop)
- Used to execute any java code from command prompt
- Allows small code to be executed without having to create a class and main method
- Start by using jshell at command prompt
- We can give any Java statement or expression at jshell prompt (jshell>)
- Results of expressions are stored in variable named as $n where n is a number. These variables are called as internal variables
- Code can be typed in multiple lines
- Commands of jshell start with /. For example, /help and /exit
- It supports TAB for completion of expression
- It automatically imports some packages. We can get the list of packages imported using /import command

jshell – commands

/drop	Drops a snippet
/edit	Opens an editor
/exit	Exits jshell
/history	Displays what was entered in this session
/help	Displays information about commands and subjects
/imports	Displays active imports
/list	Displays list of snippets and their ID
/methods	Lists methods that were created
/open file	Opens the file and reads snippet into jshell. In place of file you can give DEFAULT, JAVASE or PRINTING.
/reset	Resets the session
/save file	Saves snippets and commands to the file specified
/set	Sets configuration information
/types	Displays types created in session
/vars	Displays variables created in session

jshell – shortcuts

Tab Completion

When entering snippets, commands, subcommands, command arguments, or command options, use the Tab key to automatically complete the item. If the item can't be determined from what was entered, then possible options are provided.

Command abbreviations

An abbreviation of a command is accepted if the abbreviation uniquely identifies a command.

History navigation

A history of what was entered is maintained across sessions. Use the up and down arrows to scroll through commands and snippets from the current and past sessions.

History search

Use the Ctrl+R key combination to search the history for the string entered. The prompt changes to show the string and the match. Ctrl+R searches backwards from the current location in the history through earlier entries. Ctrl+S searches forward from the current location in the history through later entries.

Saving Context

```
jshell> /methods
|    double getNetPrice(double)

jshell> /types

jshell> /var
|    double TAX = 12.5
|    double $3 = 13500.0

jshell> /save 6-jan.jsh

jshell> /exit
|  Goodbye
```

Saving and Loading

```
>jshell
|  Welcome to JShell -- Version 9
|  For an introduction type: /help intro

jshell> /open 6-jan.jsh

jshell> /methods
|    double getNetPrice(double)

jshell> /var
|    double TAX = 12.5
|    double $3 = 13500.0

jshell>
```

Defining Method

```
jshell> int add(int n1, int n2)
   ...> {
   ...>   return n1 + n2;
   ...> }
|  created method add(int,int)

Jshell>
```

```
jshell> /methods
|    int add(int,int)
```

Forward Reference

```
jshell> double getNetPrice(double price) {
   ...>   return  price + price * TAX;
   ...> }
|  created method getNetPrice(double), however, it
cannot be invoked until variable TAX is declared

jshell> /set feedback verbose
|  Feedback mode: verbose

jshell> double TAX = 12.5;
TAX ==> 12.5
|  created variable TAX : double
|    update modified method getNetPrice(double)

jshell> getNetPrice(1000)
$3 ==> 13500.0
|  created scratch variable $3 : double
```

Defining Class

```
jshell> class Person{
   ...>  private String name;
   ...>  private int age;
   ...>  public Person(String name, int age) {
   ...>   this.name=name;
   ...>   this.age=age;
   ...>  }
   ...>  public String toString() {
   ...>   return  name + " : "  + age;
   ...>  }
   ...> }
|  created class Person
jshell> /types
|    class Person

Jshell>
```

Language Enhancements – Milling Project Coin

- Private methods in interface
- Enhancements to try with resources
- It is possible to use <> operator with anonymous inner class
- _ cannot be identifier

Interface Evolution

Java 7

- ❑ Static and final variables
- ❑ Abstract methods

Java 8

- ❑ Static methods
- ❑ Default methods

Java 9

- ❑ Private methods

Interface Extensions – Private Methods

- Interface can now have private methods that can be called from other methods of the interface
- Private methods in interface cannot be called from implementing class
- Provides more reusability in interfaces as these private methods can be called from other default methods
- Must be declared with private keyword and must contain body

Private Methods in Interface

```
public interface Logger {
 private void log(String message, String target) {
     // log message to target
 }

 default void logToFile(String message) {
     log(message, "file");
 }

 default void logToDatabase(String message) {
     log(message, "database");
 }
}
```

Try with Resources Enhancements

- Try with resource can now use a resource that is already declared outside the Try-With-Resource Statement as final or effectively final

```
// Before Java 9
Scanner s1 = new Scanner(System.in);
// some other code
try (Scanner s2 = s1) {
      String line = s2.nextLine();
}
```

```
// From Java 9
Scanner s = new Scanner(System.in);

// Variable s can be used with try with resource
// even through it was created earlier.

try (s) {
      String line = s.nextLine();
}
```

Diamond operator with anonymous inner class

- Before Java 9, diamond operator could not be used with anonymous inner class
- Starting from Java 9, it is possible to use diamond (<>) operator with anonymous inner class just like normal classes

```
Comparator<String> lengthCompare = new Comparator<>() {

        @Override
        public int compare(String s1, String s2) {
                return s1.length() - s2.length();
        }

};
```

Optional Class

- Optional class was introduced in Java 8 to represent a value that is optional
- Optional class object may contain value or empty
- It has methods get(), isPresent(), ifPresent()

```
Optional<LocalDate> startDate = Optional.of(LocalDate.now());
Optional<LocalDate> endDate = Optional.empty();

startDate.ifPresent(System.out::println);

System.out.println(endDate.isPresent()?endDate.get():"Not
available");
```

Optional Class New Methods

```
Stream<T> stream()

void ifPresentOrElse(Consumer<? super T> action,
                     Runnable emptyAction)

Optional<T> or(Supplier<? extends
                        Optional<?extends T>> supplier)
```

Optional new methods examples

```
Optional<LocalDate> startDate =
                    Optional.of(LocalDate.now());
Optional<LocalDate> endDate = Optional.empty();

startDate.ifPresentOrElse( System.out::println,
         () -> System.out.println("Missing start date"));
endDate.ifPresentOrElse( System.out::println,
         () -> System.out.println("Missing end date"));
```

```
Optional<LocalDate> finalDate =
          endDate.or(() -> Optional.of(LocalDate.now()));

System.out.println(finalDate.get());
```

New Methods in LocalDate class

```
Stream<LocalDate> datesUntil(LocalDate endExclusive)
Stream<LocalDate> datesUntil
                  (LocalDate endExclusive, Period step)
```

```
// Print all dates from 1-10-2017 to 31-12-2017

LocalDate start = LocalDate.of(2017, 10, 1);
start.datesUntil(LocalDate.of(2018,1,1))
     .forEach(System.out::println);
```

New Methods in List, Set, Map and Map.Entry

- List.of()
- List.of(values)
- Set.of()
- Set.of(values)
- Map.of()
- Map.of(key, value, key, value,…)

List.of()

```
// Immutable Empty List
List<String>  emptyList = List.of();

// Java 8
List<String> list = new ArrayList<>();
list.add("one");
list.add("two");
list.add("three");
List<String>
immutableList=Collections.unmodifiableList(list);

// Java 9
List<String>
immutableList=List.of("one","two","three");
```

Immutable List Features

- List is immutable i.e. we can't add, delete or modify elements
- Nulls are not allowed
- Serializable if all elements are serializable
- We can create a list of up to 10 elements as there are 10 parameters for of() method
- List.<E>of(E…elements) can take any number of parameters including an array

Map.of() and Map.entry()

```
// Immutable Empty Map
Map<String, String> emptyMap = Map.of();

Map<String,String> phones=
     Map.of("Bill","39393393988","Larry","9988776655");


Map<Integer,String> maps2 =
 Map.ofEntries( Map.entry(1,"One"),Map.entry(2,"Two"));
```

Stream API Improvements

Stream API added new methods to Stream Interface

- ❑ **takeWhile()**
- ❑ **dropWhile()**
- ❑ **iterate()**
- ❑ **ofNullable()**

takeWhile()

default Stream<T> takeWhile(Predicate<? super T> predicate)

Returns elements of stream that match predicate. It stops taking elements when predicate is false.

```
Stream<Integer> nums = Stream.of(10,22,31,34,87,12);
Stream<Integer> evenNums =
          nums.takeWhile( n -> n % 2 == 0 );
evenNums.forEach(System.out::println);
```

```
Output
======
10
22
```

dropWhile()

default Stream<T> dropWhile(Predicate<? super T> predicate)

Returns elements of stream after dropping elements that match predicate. It stops dropping elements when predicate is false.

```
Stream<Integer> nums = Stream.of(10,22,31,34,87,12);
Stream<Integer> nums2 = nums.dropWhile(n -> n % 2 == 0);
nums2.forEach(System.out::println);
```

```
Output
======
31
34
87
12
```

iterate()

static <T> Stream<T> iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> next)

Returns a stream of elements that are produced by **next** until **hasNext** is false.

```
Stream<Integer> oddNums =
      Stream.iterate(1,v->v < 10,v->v + 2);
oddNums.forEach(System.out::println);
```

```
Output
======
1
3
5
7
9
```

ofNullable()

static <T> Stream<T> ofNullable(T t)

Returns a sequential Stream containing a single element, if non-null, otherwise returns an empty Stream.

```
Stream<Integer> s1 =  Stream.ofNullable(null);
System.out.println(s1.count());


Stream<Integer> s2 =  Stream.ofNullable(35);
System.out.println(s2.iterator().next());
```

```
Output
======
0
35
```

Compact Strings

- Compact string is a string that uses only one byte for character instead of 2 bytes as it is the case in Java 8
- Java decides whether to use byte[] or char[] to store strings based on character set used
- For LATIN-1 representation, a byte array is used and in other cases char array is used (UTF-16 representation)
- To know whether a string is using char[] or byte[] , a field called coder is used internally
- The whole implementation is automatic and transparent to developer
- Compact strings reduce the amount of memory occupied by strings by half

Compact Strings

String in Java 8

String in Java 9

Java Platform Module System

- It is also known as Project Jigsaw
- Has been in development for more than a decade (First started in 2005)
- Finally made it to Java 9
- JSR 376
- Allows classes to be divided into modules, which in turn contain packages
- All Java packages are incorporated into modules in Java 9
- Java SE 9 created modules into which all packages are included
- Different topics have different modules – sql, xml, logging etc.
- All important packages are placed in java.base module
- All other modules depend on java.base
- Modules like java.se.ee and java.se are aggregator modules
- Java platform is modularized into 95 modules

Goals of JPMS

- ❑ Strong Encapsulation
- ❑ Reliable Configuration
- ❑ Scalable Java Platform
- ❑ Greater Platform Integrity
- ❑ Better performance

Why JPMS?

- To modularize JDK so that only required parts can be used instead of whole rt.jar (runtime jar which is 60 MB) being used
- To Modularize Applications
- To increase security. Only exposed classes can be used outside module - Strong encapsulation
- Reduced footprint - Scalable Java Platform
- Easy depreciation - Unnecessary modules can be eliminated
- Reliable configuration - dependency between modules is clearly defined
- Can introduce new experimental features using incubator modules

What is a Module?

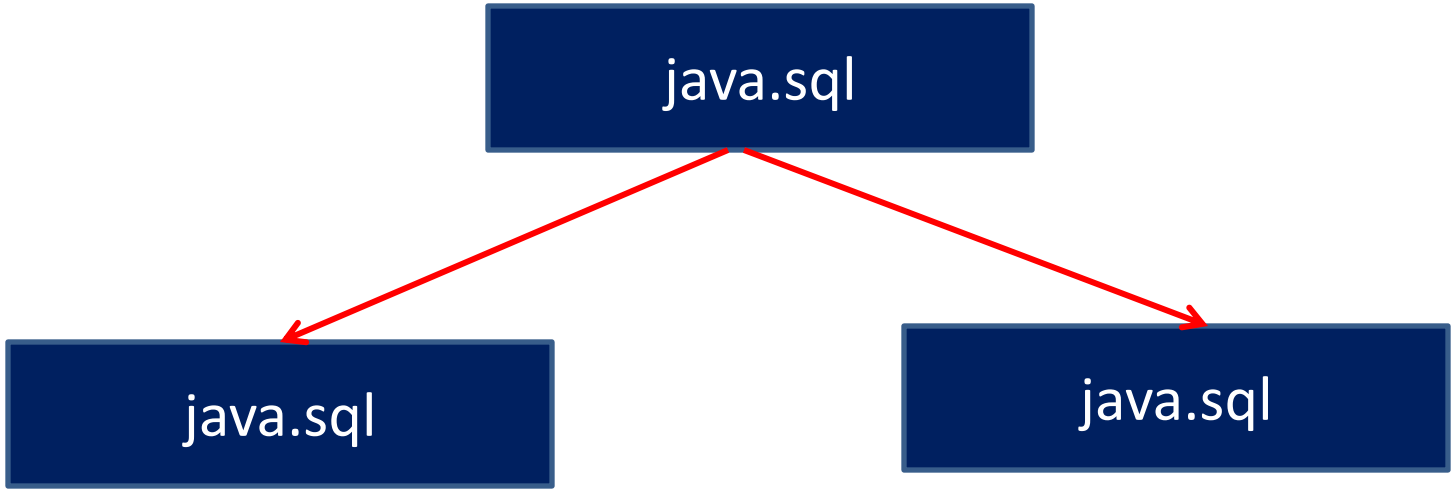
- A module is a collection of packages
- Each module contains information about other module that it depends on using requires clause
- Every module depends on java.base by default
- JVM validates module graph at startup
- Module descriptor (module-info.class) is used to provide details about module
- Typically packaged as .jar
- A new variable called modulepath is used to find modules
- When .jar is found in modulepath it is considered as module
- Each module specifies which packages it exports using exports clause

```
java -list-modules
```

Module java.sql

```
module java.sql
{
  exports  java.sql;
  exports  javax.sql;
  exports  javax.transaction.xa;

  // Modules that java.sql depends on
  requires java.logging;
  requires java.xml;
}
```



Module java.base

```
module java.base
{
  exports  java.lang;
  exports  java.util;
  exports  java.io;
  // more
}
```

```
java --describe-module java.base
```

Describing Module

```
C:\dev\java\test>java --describe-module java.sql
java.sql@9
exports java.sql
exports javax.sql
exports javax.transaction.xa
requires java.base mandated
requires java.logging transitive
requires java.xml transitive
uses java.sql.Driver
```

Keywords used in Module Descriptor

```
requires  modulename
requires  transitive modulename

exports
exports to module-names     // qualified export
```

Types of Modules

Application Modules

Modules created by programmers.

Automated Modules

JAR files placed in the module path without module descriptor are called automated modules.

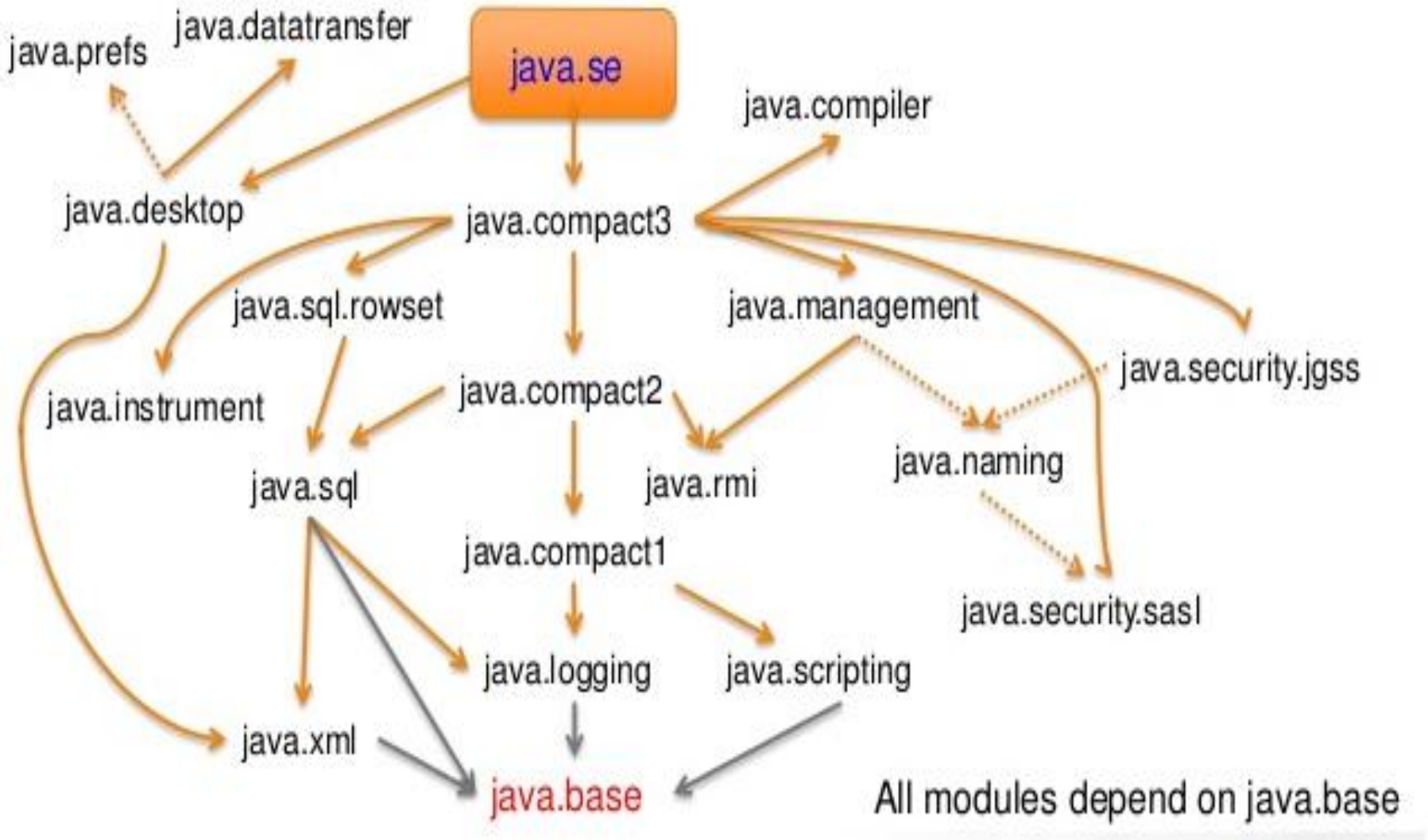
Unnamed Modules

JAR files in classpath are treated as unnamed modules. They don't have any name so they can read and export all modules.

Platform Modules

Modules provided by JDK. JDK is providing more than 90 modules.

Platform Modules



Platform Modules

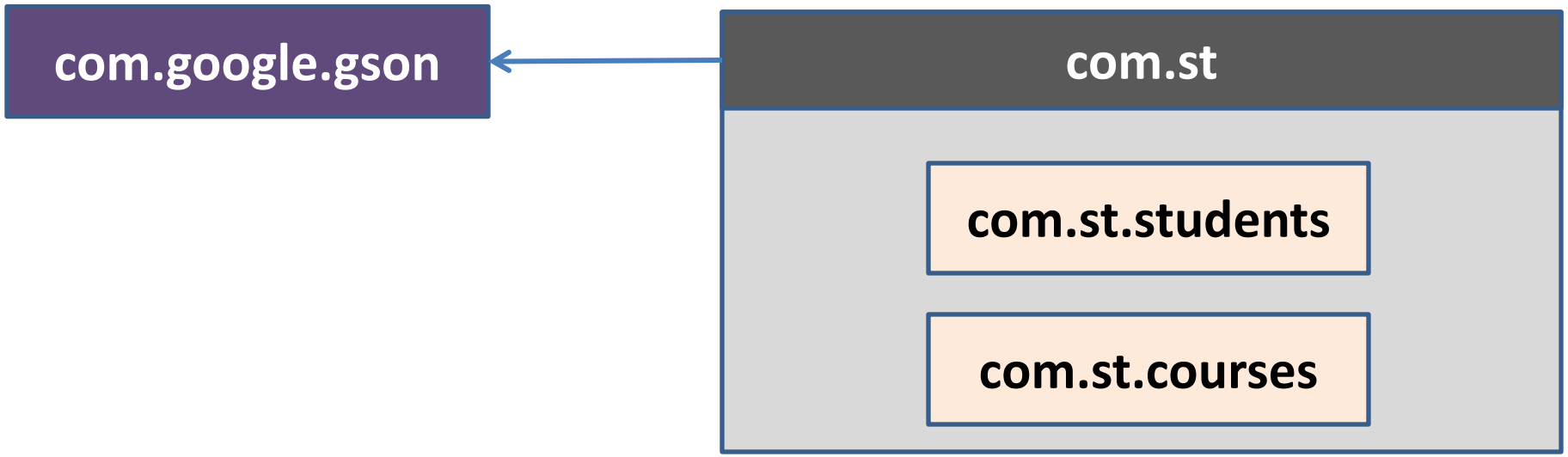
Module	Description
java.base	Defines the foundational APIs of the Java SE Platform.
java.compiler	Defines the Language Model, Annotation Processing, and Java Compiler APIs.
java.desktop	Defines the AWT and Swing user interface toolkits, plus APIs for accessibility, audio, imaging, printing, and JavaBeans.
java.instrument	Defines services that allow agents to instrument programs running on the JVM.
java.logging	Defines the Java Logging API.
java.management	Defines the Java Management Extensions (JMX) API.
java.naming	Defines the Java Naming and Directory Interface (JNDI) API.
java.prefs	Defines the Preferences API.
java.scripting	Defines the Scripting API.
java.se	Defines the core Java SE API.
java.se.ee	Defines the full API of the Java SE Platform.
java.sql	Defines the JDBC API.
java.sql.rowset	Defines the JDBC RowSet API.
java.xml	Defines the Java API for XML Processing (JAXP), the Streaming API for XML (StAX), the Simple API for XML (SAX), and the W3C Document Object Model (DOM) API.

Creating a module

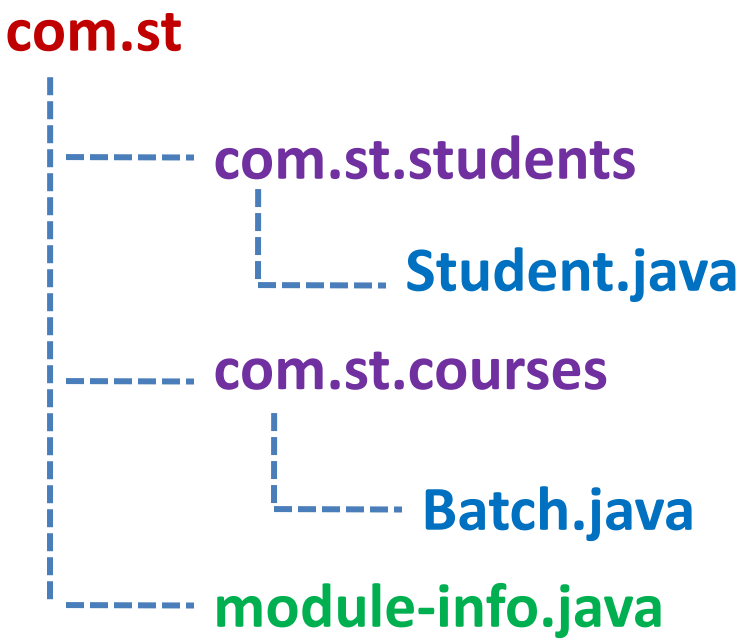
module-info.java

```
module com.st
{
   requires com.google.gson;

   exports  com.st.students;
   exports  com.st.courses;
}
```



Module Directory Structure



Creating com.st module

Source Directory Structure

```
src
    com.st
      module-info.java
      com
          st
              Student.java
              Test.java
```

module-info.java

```
module com.st {
        exports com.st;
}
```

com/st/Student.java

```
package com.st;

public class Student {
 private String name, course;

 public Student(String name, String course) {
   this.name = name;
   this.course = course;
 }

 public String toString() {
   return  this.name + " pursuing " + this.course;
 }

}
```

com/st/Test.java

```
package com.st;

public class Test {
 public static void main(String[] args) {
  Student s = new Student("George Mike","Java");
  System.out.println(s.toString());
 }
}
```

Commands

```
javac -d mods/com.st ^
      src/com.st/module-info.java ^
      src/com.st/com/st/*.java
```

```
java --module-path mods --describe-module com.st
com.st file:///C:/dev/java/test/mods/com.st/
exports com.st
requires java.base mandated
```

```
java --module-path mods --module com.st/com.st.Test
George Mike pursuing Java
```

Unnamed Module

- Any class that is not a member of a named module is considered to be part of unnamed module
- It is similar to unnamed package
- All classes compiled in Java 8 and before belong to unnamed module
- It requires every other named module automatically. It means it read all other modules without any explicit declaration
- Unnamed module exports all its packages
- However, unnamed modules cannot be read by named modules, they can be read only by other unnamed modules

Unnamed Module Example

```
public class TestModule {
  public static void main(String[] args) {
   Module module = TestModule.class.getModule();
   System.out.println("Module : " + module);
   System.out.println("Name   : " + module.getName());
   System.out.println("isNamed: " + module.isNamed());
   System.out.println("Descriptor: " +
                      module.getDescriptor());
  }
}
```

```
Module : unnamed module @2d554825
Name   : null
isNamed: false
Descriptor: null
```

Using named module from unnamed module

```
import com.st.Student;

public class TestModule {
  public static void main(String[] args) {
        Student s = new Student("Tim", "Python");
        System.out.println(s.toString());
  }
}
```

```
javac -p mods --add-modules com.st TestModule.java
```

```
java -p  mods --add-modules com.st TestModule
```

Summary

- More modular Java Platform
- Easier to test code using Jshell
- Minor improvements in Language, making it more easy to use
- Useful additions to library to make it more productive