



May 2019, IPT Course  
Java Web Debelopment

# Java Core, Functional, I/O & Concurrent Programming

Trayan Iliev

[tiliev@iproduct.org](mailto:tiliev@iproduct.org)

<http://iproduct.org>

Copyright © 2003-2019 IPT - Intellectual  
Products & Technologies

# About me



## Trayan Iliev

- CEO of IPT – Intellectual Products & Technologies
- Oracle® certified programmer 15+ Y
- end-to-end reactive fullstack apps with Java, ES6/7, TypeScript, Angular, React and Vue.js
- 12+ years IT trainer
- Voxxed Days, jPrime, jProfessionals, BGOUG, BGJUG, DEV.BG speaker
- Organizer RoboLearn hackathons and IoT enthusiast (<http://robolearn.org>)



Since 2003 we provide trainings and share skills in  
JS/ TypeScript/ Node/ Express/ Socket.IO/ NoSQL/  
Angular/ React / Java SE/ EE/ Spring/ REST / SOA:

- ❖ Spring, Java EE, JSF, Portals: Liferay, GateIn
- ❖ Node.js + Express + React + Redux + GraphQL
- ❖ Angular + TypeScript + Redux (ngrx) + Progressive WA
- ❖ Reactive IoT with Reactor / RxJava / RxJS
- ❖ SOA & Distributed Hypermedia APIs (REST)
- ❖ Domain Driven Design & Reactive Microservices

# What Will You Learn in the Course?

- ❖ Object Oriented Programming (OOP), SOLID principles
- ❖ Object creation and initialization
- ❖ Strings, Regular Expressions, Date and Time API, property files, ResourceBundles and Localization
- ❖ Exception handling
- ❖ Packages and access modifiers, code reuse, inheritance
- ❖ Polymorphism, abstract classes and methods, interfaces
- ❖ Inner classes and interfaces
- ❖ Functional programming with Java 8+

# What Will You Learn in the Course? (cont.)

- ❖ Data structures in Java
- ❖ Generics
- ❖ Java I/O, NIO, and NIO2
- ❖ Concurrent Programming with Java



# Course Schedule

- ❖ Block 1: 9.00 - 10.30
  - ❖ Pause: 10.30 - 10.45
  - ❖ Block 2: 10.45 - 12.15
  - ❖ Lunch: 12.15 - 12.45
  - ❖ Block 3: 12.45 - 14.15
  - ❖ Pause: 14.15 - 14.30
  - ❖ Block 4: 14.30 – 16.00
  - ❖ Pause: 16.00 - 16.15
  - ❖ Block 5: 16.15 - 17.15
- 
- ❖ Training – Monday, Thursday

# Where to Find the Code?

Java Core, Functional, I/O & Concurrent  
Programming projects and examples are  
available @ GitHub:

<https://github.com/iproduct/course-java-web-development>

# Agenda for This Session

- ❖ SOLID design principles of OOP
- ❖ Key features of Java language
- ❖ Stack and Heap (quick review)
- ❖ Literals and Operators
- ❖ Assignments and variables
- ❖ Scope
- ❖ Garbage collection
- ❖ Java Platform Module System (JPMS)
- ❖ Java basics (recap)
- ❖ Object creation and initialization
- ❖ Class loaders - Bootstrap, Extensions, System, Context



# Basic Concepts in OOP and OOAD

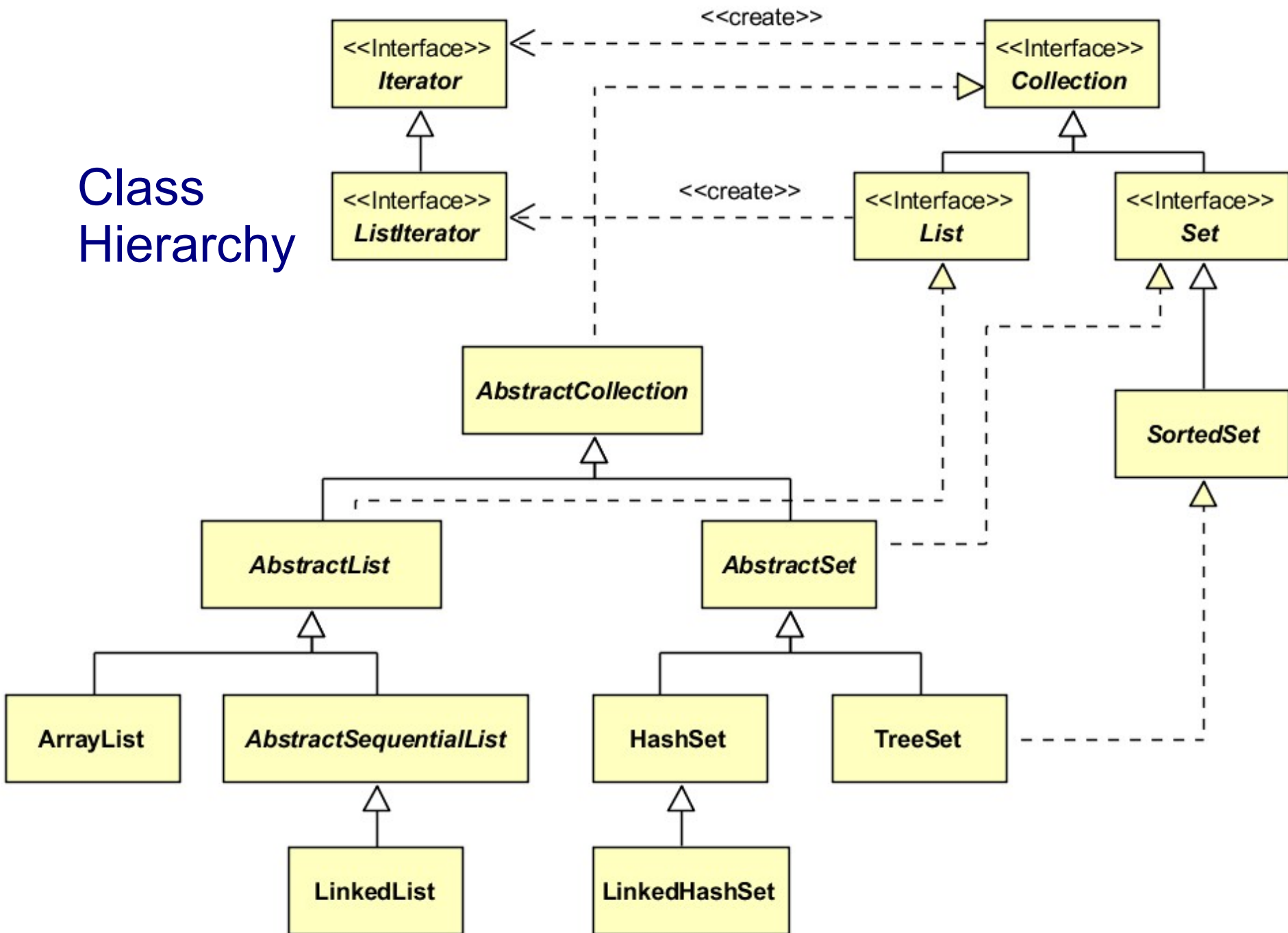
- ❖ interface and implementation – we divide what remains constant (contractual interface) from what we would like to keep our freedom to change (hidden realization of this interface)
- ❖ interface = **public**
- ❖ implementation = **private**
- ❖ This separation allows the system to evolve while maintaining backward compatibility to already implemented solutions, enables parallel development of multiple teams
- ❖ **programming based on contractual interfaces**

# Object-Oriented Approach to Programming

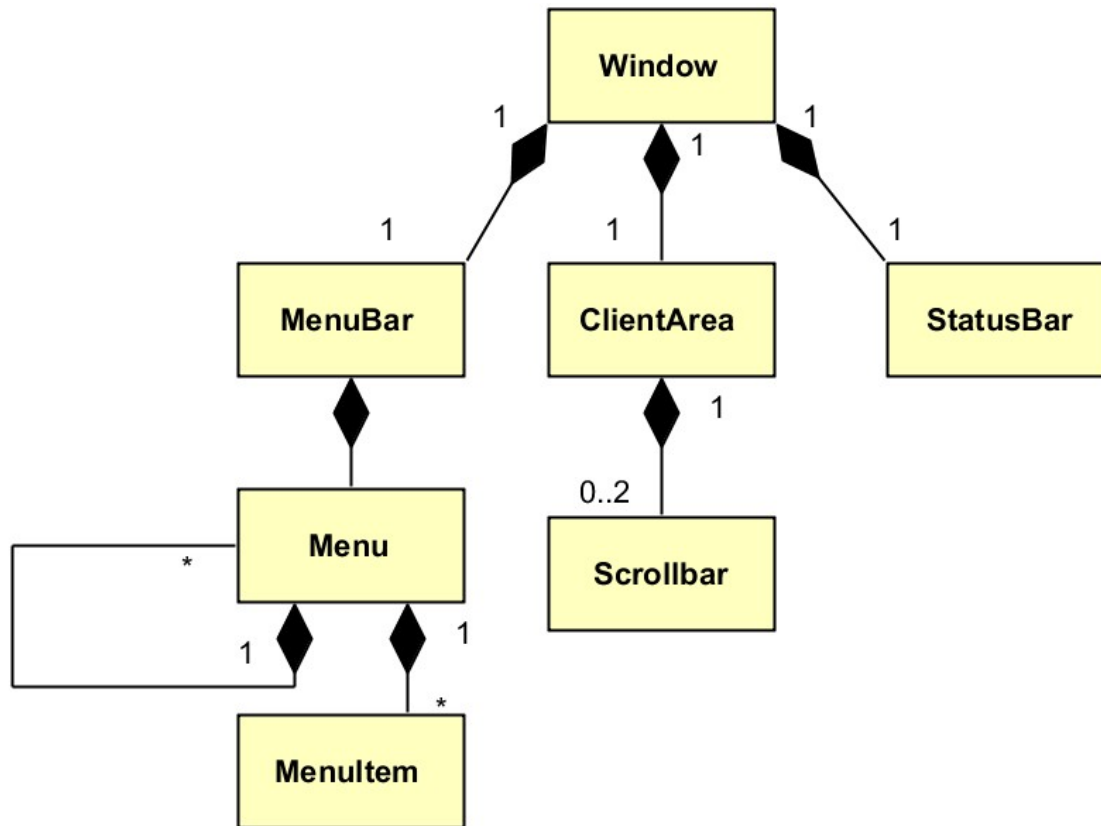
Key elements of the object model [Booch]:

- ❖ **class, object, interface and implementation**
- ❖ **abstraction** – basic distinguishing characteristics of an object
- ❖ **capsulation** – separating the elements of abstraction that make up its structure and behavior - interface and implementation
- ❖ **modularity** – decomposing the system into a plurality of components and loosely connected modules - principle: maximum coherence and the minimum connectivity
- ❖ **hierarchy** – class and object hierarchies

# Class Hierarchy



# Object Hierarchy



# Object-Oriented Approach to Programming

Additional elements of the object model [Booch]:

- ❖ **typing** – requirement for the class of an object such that objects of different types can not be replaced (or can in a strictly limited way)
  - static and dynamic binding
  - polymorphism
- ❖ **concurrency** – abstraction and synchronization of processes
- ❖ **length of life** – object-oriented databases

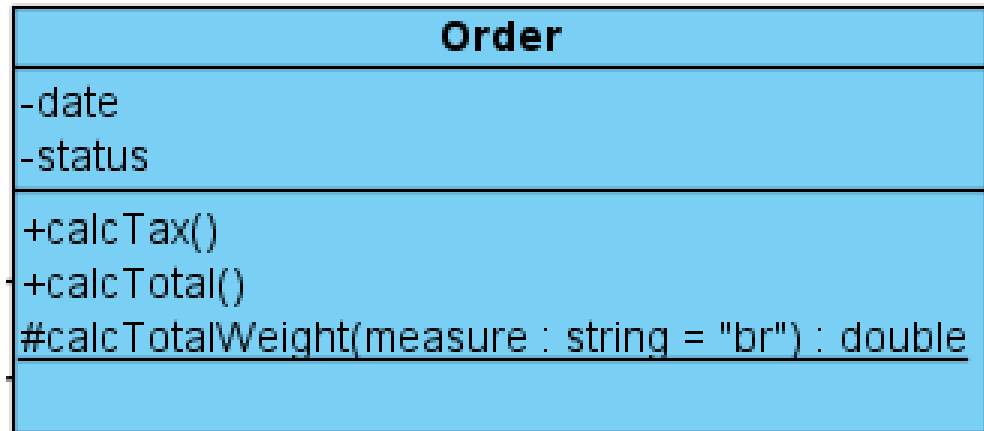
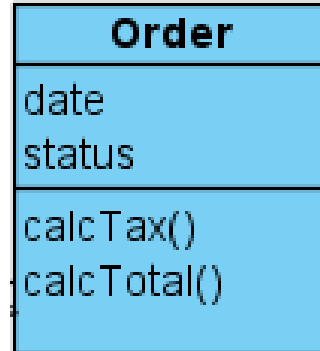
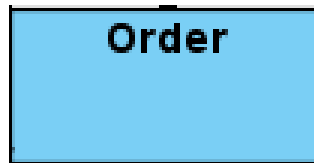


# Classes

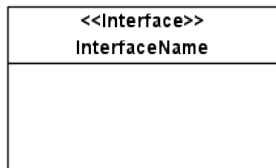
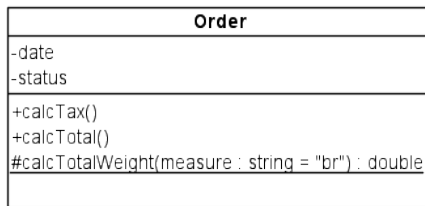
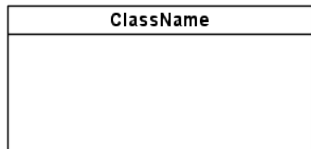
**Class** – describes a set of objects that share the same specifications of the characteristics (attributes and methods), constraints and semantics

- attributes – instances of properties in UML, they can provide end of association, object *structure*
- operations - behavioral characteristics of a classifier, specifying name, type, parameters and constraints for invoking definitely associated with the operation behavior

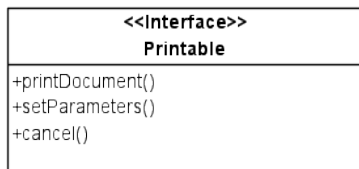
# Classes - Graphical Notation in UML



# Elements of Class Diagrams

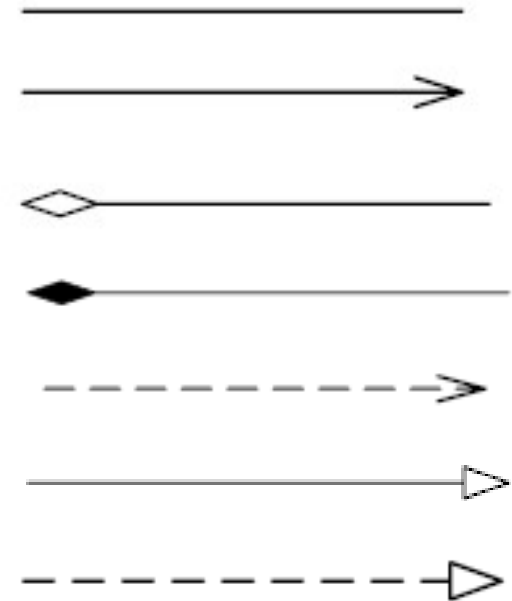


InterfaceName

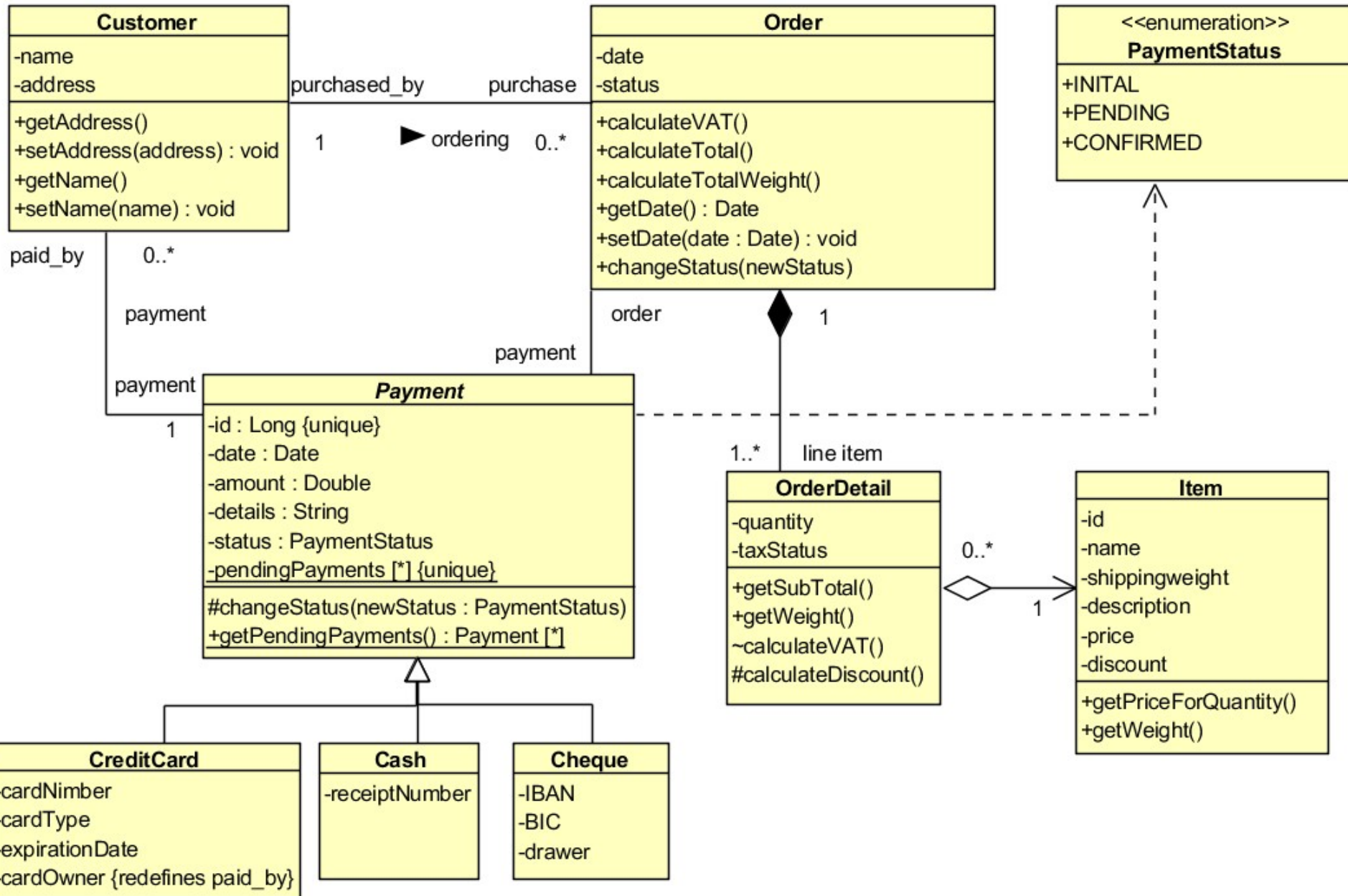


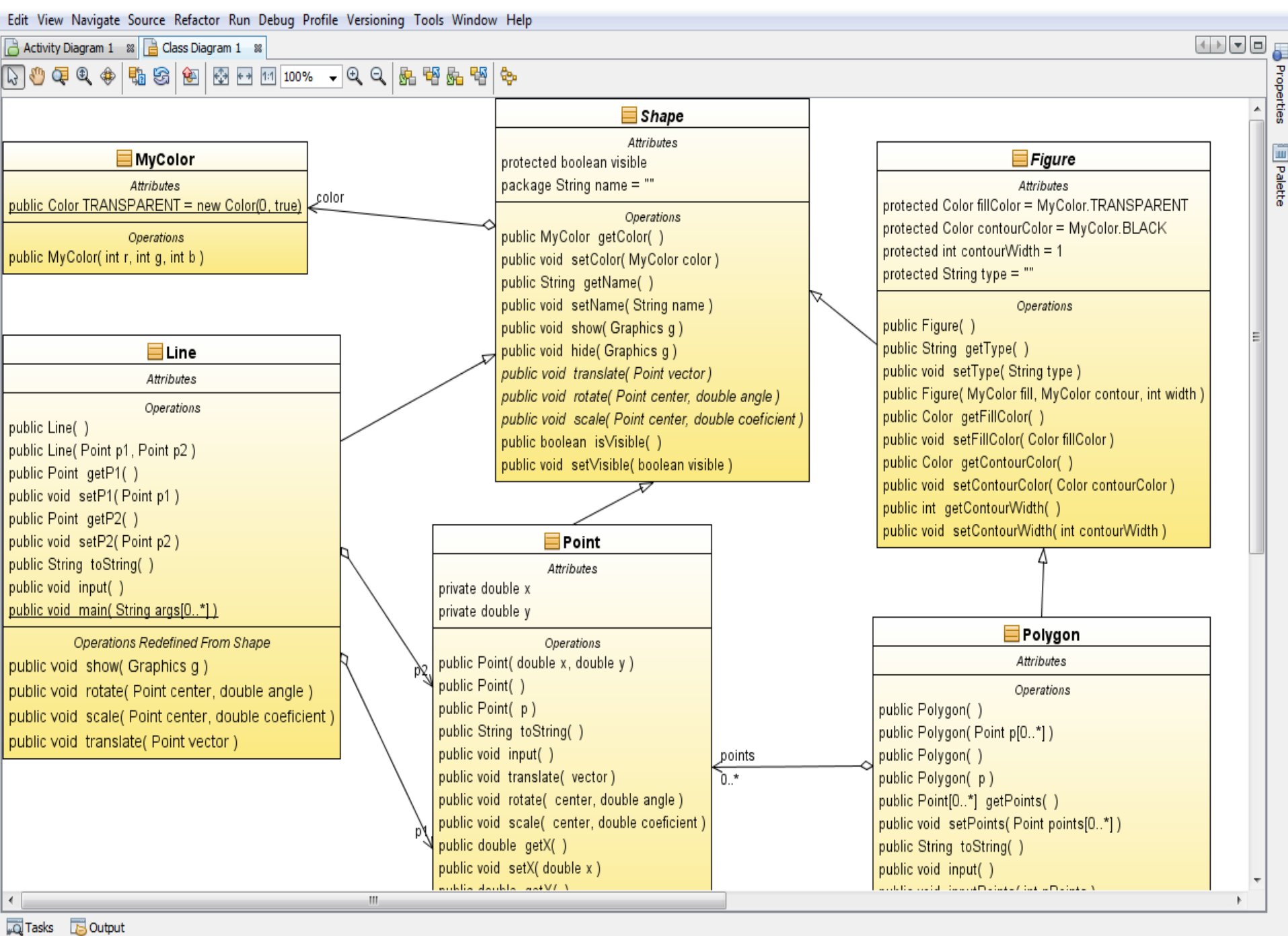
## Types of connections:

- association
- aggregation
- composition
- dependence
- generalization
- realization



# Class Diagram - 1







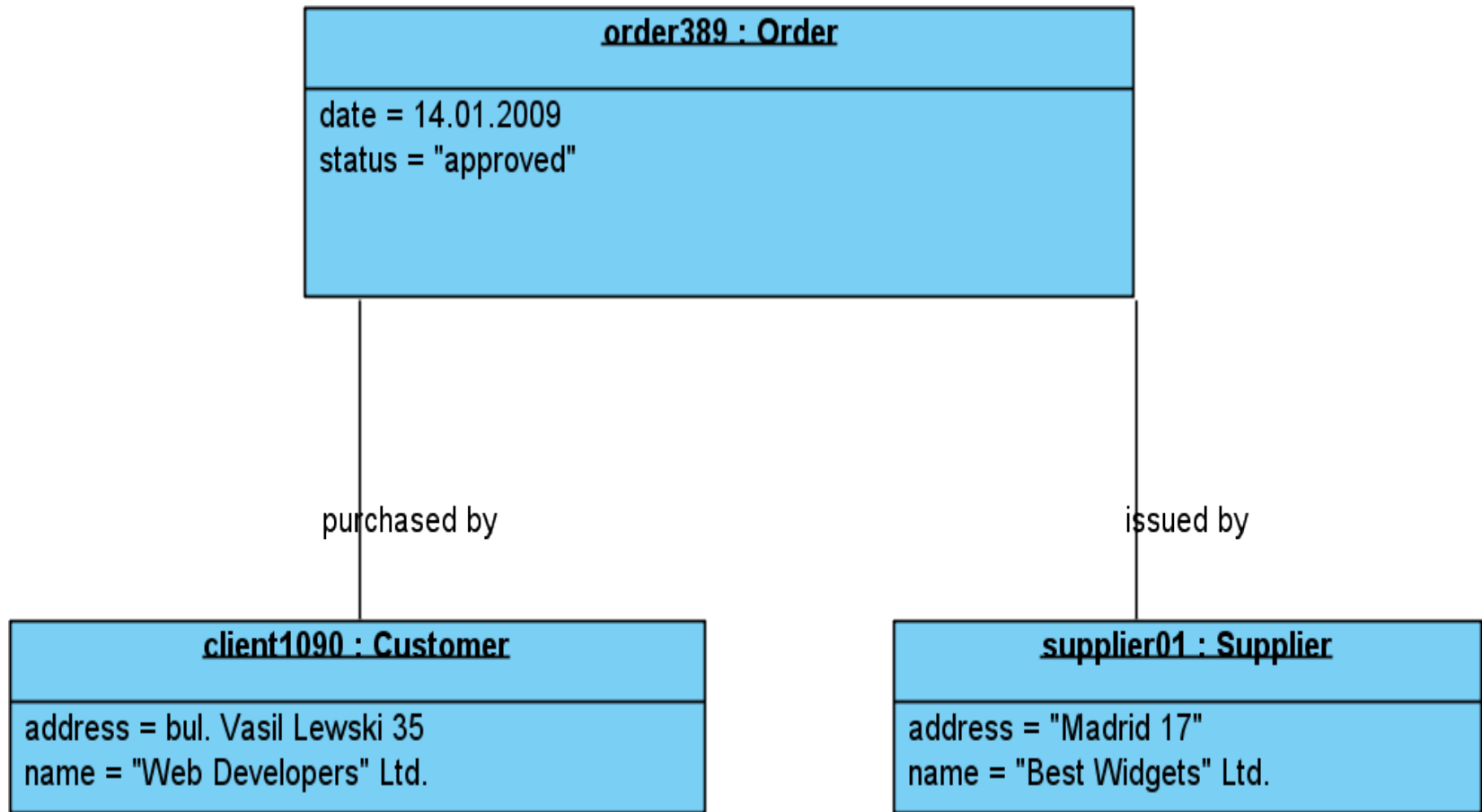
# Objects

**Instance specification = Object** – represents an instance of the modeled system, for example class -> object association -> link, property -> attribute, etc.

- can provide illustration or example of object
- describes the object in a particular moment of time
- may be uncomplete
- Example:

<b>order389 : Order</b>
date = 14.01.2009 status = "approved"

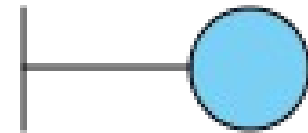
# Object Diagram



# Analysis Classes Stereotypes

Analysis classes are used in the mapping and analysis of system architecture - they present rather different roles and responsibilities, than specific classes to be realized, and are independent of implementation technology:

- <<control>> - business logic
- <<entity>> - data
- <<boundary>> - user or system interface



...

# SOLID design principles of OOP

- ❖ **Single responsibility principle** - a class should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the class.
- ❖ **Open–closed principle** - software entities should be open for extension, but closed for modification.
- ❖ **Liskov substitution principle** - Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
- ❖ **Interface segregation principle** - Many client-specific interfaces are better than one general-purpose interface.
- ❖ **Dependency inversion principle** - depend upon abstractions, not concretions.

# Key Features of Java Language

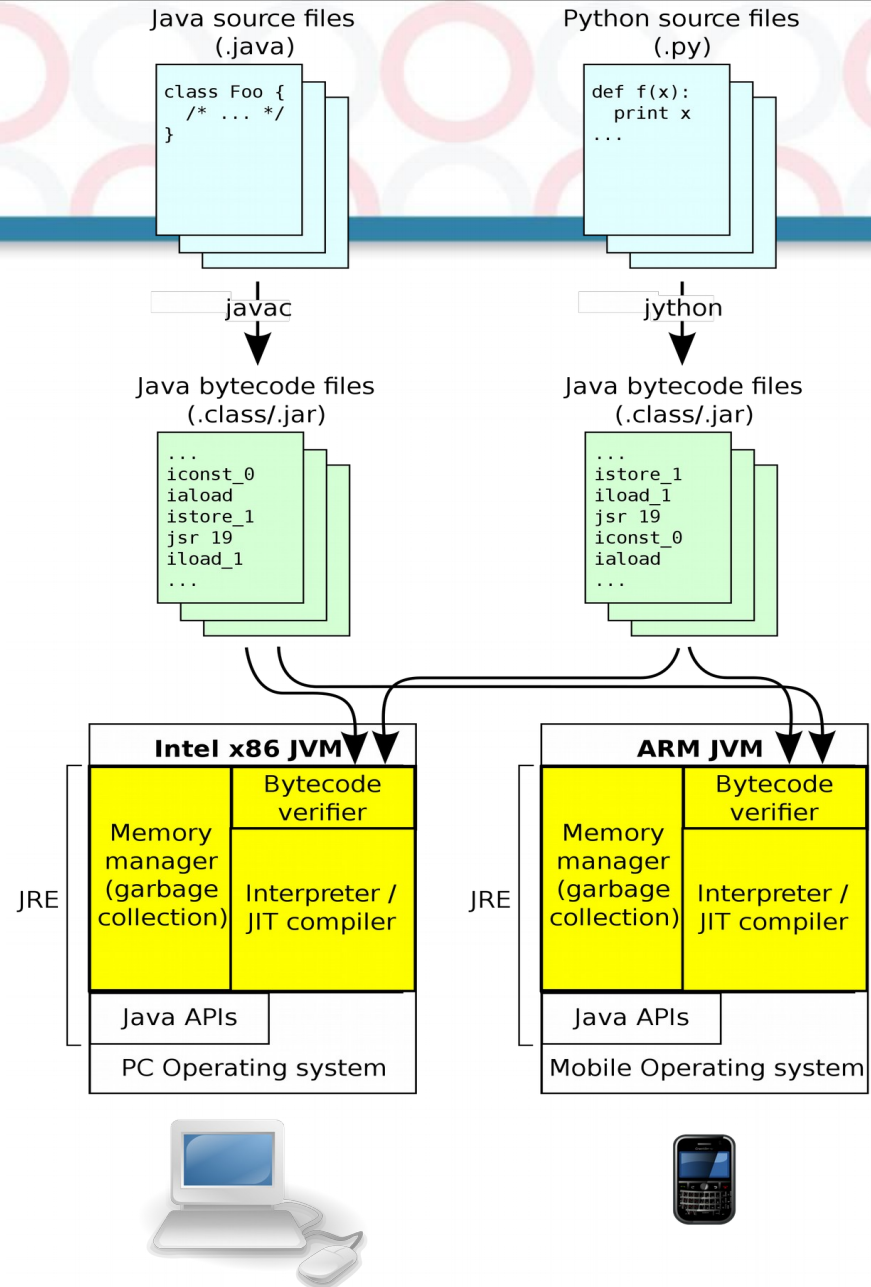
- ❖ **Single base hierarchy** - inheritance from only one parent class, with the possibility of implementation of multiple interfaces
- ❖ **Garbage Collector** – portability and platform independence, fewer errors
- ❖ **Secure Code** – separation of business logic from the error handling and exceptions
- ❖ **Multithreading** - easy realization of parallel processing
- ❖ **Persistence** – Java Database Connectivity (JDBC) and Java Persistence API (JPA)



# Integrated Development Environments for Java Applications

- ❖ Java™ development environment types:  
**JavaSE, JavaEE, JavaME, JavaFX**
- ❖ **JavaSE: Java Development Kit (JDK) and Java Runtime Environment (JRE)**
- ❖ Java™ compiler - `javac`
- ❖ **Java Virtual Machine (JVM) - `java`**
- ❖ Source code → Byte code
- ❖ Installing JDK 8
- ❖ Compile and run programs from the command line
- ❖ IDEs: **Eclipse, IntelliJ IDEA, NetBeans**

# Java Virtual Machine (JVM)



# Java Application Stack

Level of Optimization

**Java™ Custom Application** – Level & patterns of garbage production, Concurrency, IO/Net, Algorithms & Data structures, API & Frameworks

**Application Server** – Web Container, EJB Container, Distributed Transactions Dependency Injection, Persistence - Connection Pooling, Non-blocking IO

**Java™ Virtual Machine (JVM)** – Garbage Collection, Threads & Concurrency, NIO

**Operating System** – Virtual Memory, Paging, OS Processes and IO/Net libraries

**Hardware Platform** – CPU, Memory, IO, Network

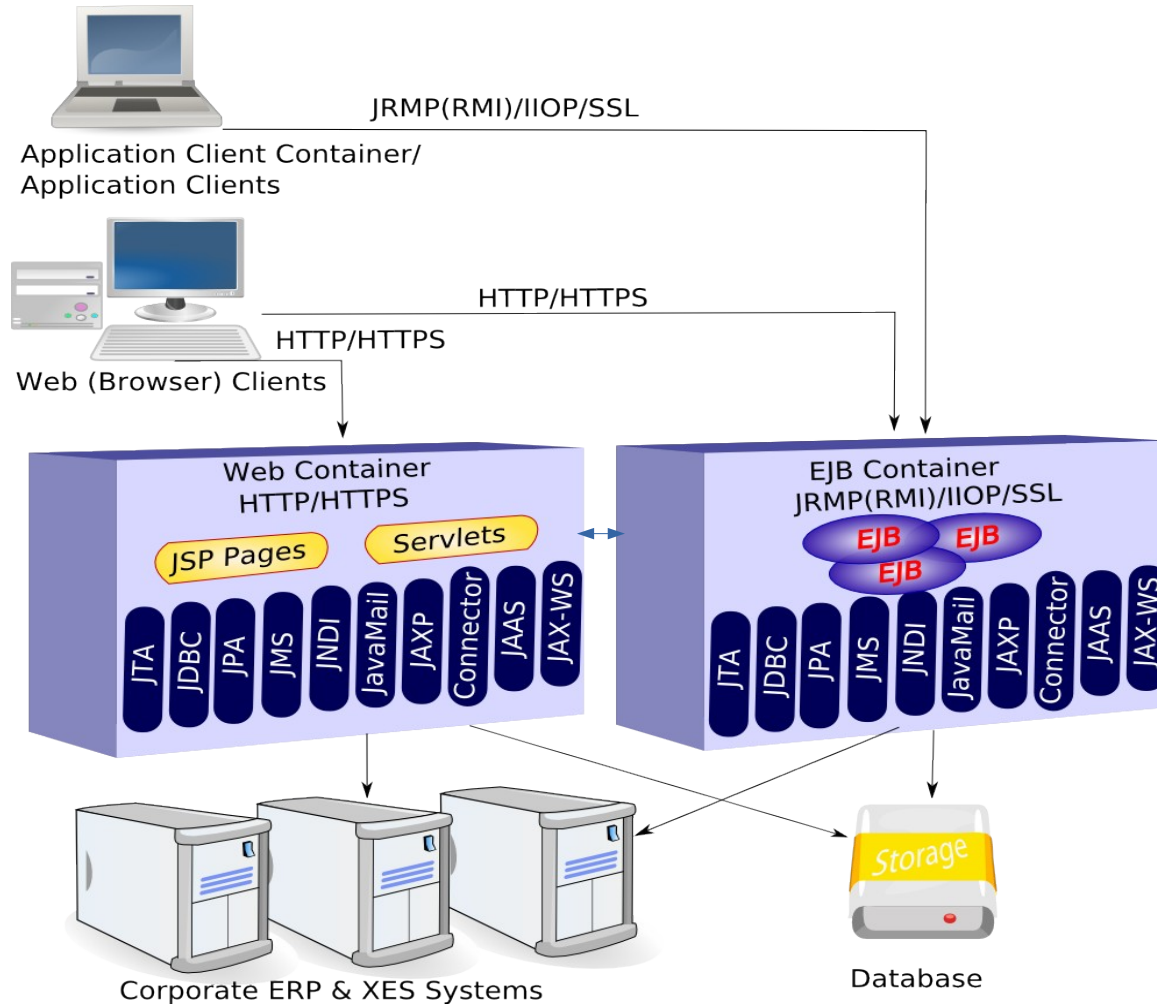
Processing Node 1

Processing Node2

...

Processing Node N

# Java EE Architecture



# Key Elements of Java™ Language - Data Types, Variables and Constants

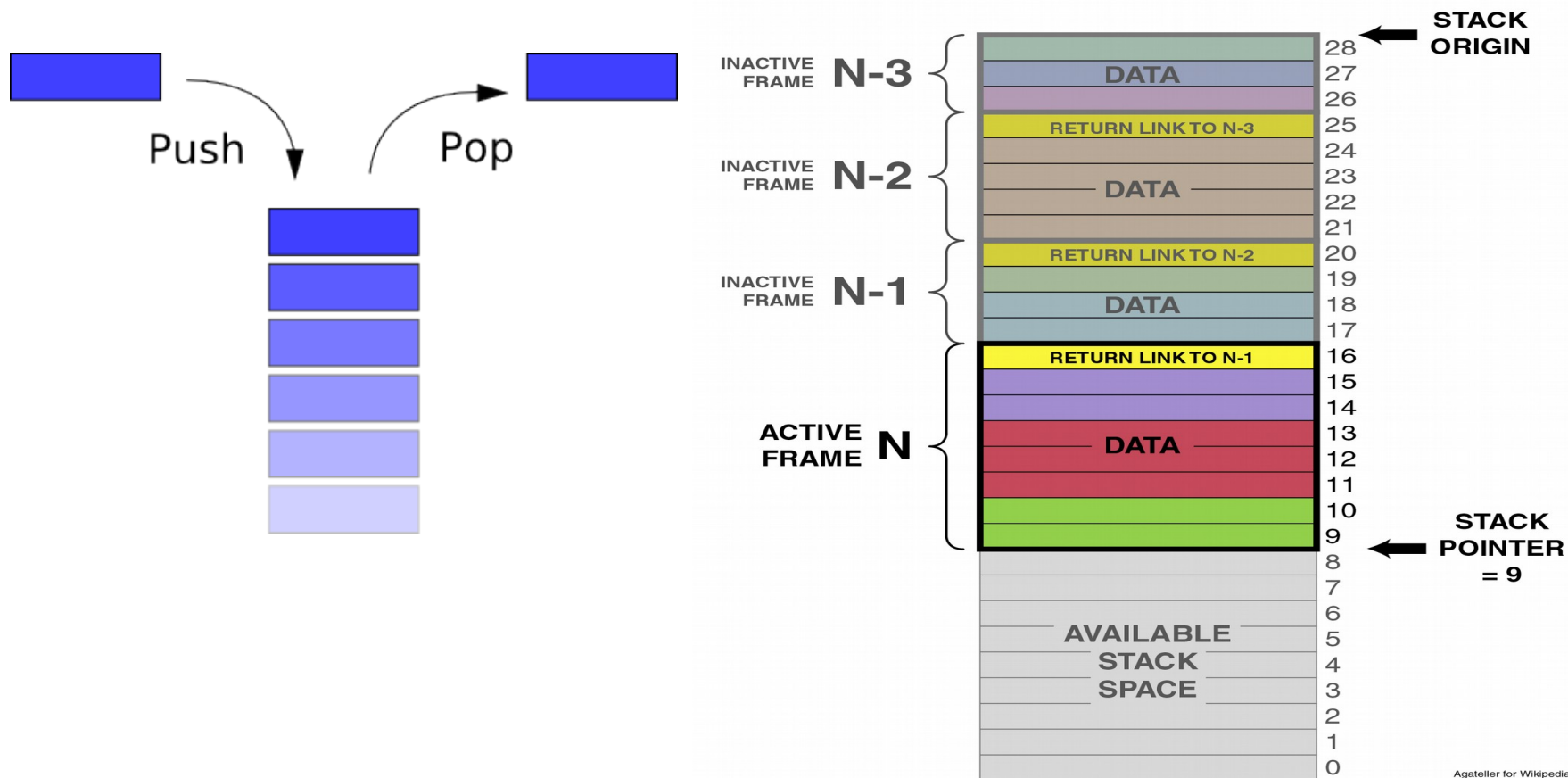
- ❖ Objects and references
- ❖ Creating objects
- ❖ Primitive and object data types
- ❖ Data Structures - arrays
- ❖ Fields and methods of an object
- ❖ Using ready libraries
- ❖ Static attributes and methods - static
- ❖ Variables and constants - final



# Memory Types

- ❖ **Register memory** - CPU registers, fast, small numbers stored operand instructions just before treatment
- ❖ **Program Stack** = Last In, First Out (LIFO) – Keep primitive data types and references to objects during program execution
- ❖ **Dynamically allocated memory – Heap** – can store different sized objects for different periods of time, can create new objects dynamically and to be released – Garbage Collector
  - Young generation – objects that exist for short period
  - Old generation – objects that exist longer
  - ~~Permanent Generation – class definitions.~~ **Java 8 Metaspace**
- ❖ **Constant storage, non-RAM storage (external memory)**

# Program Stack



Agateller for Wikipedia  
Public Domain 2006

c:\CourseAdvancedJavaVerint\Temp&gt;jstack 1612

2015-07-16 15:52:18

Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.45-b02 mixed mode):

```
"DestroyJavaVM" #21 prio=5 os_prio=0 tid=0x0000000024b8000 nid=0x1f04 waiting on condition [0x0000000000000000]
  java.lang.Thread.State: RUNNABLE

"Thread-9" #20 prio=5 os_prio=0 tid=0x00000000bea7000 nid=0x2348 waiting for monitor entry [0x00000000d14f000]
  java.lang.Thread.State: BLOCKED (on object monitor)
    at simpletest.TwoThreadsSynchronizedCounter.lambda$0(TwoThreadsSynchronizedCounter.java:14)
    - waiting to lock <0x00000000d5e660a0> (a java.lang.Object)
    at simpletest.TwoThreadsSynchronizedCounter$$Lambda$1/424058530.run(Unknown Source)
    at java.lang.Thread.run(Thread.java:745)

"Thread-8" #19 prio=5 os_prio=0 tid=0x00000000bea5800 nid=0x6ac waiting for monitor entry [0x00000000ca2e000]
  java.lang.Thread.State: BLOCKED (on object monitor)
    at simpletest.TwoThreadsSynchronizedCounter.lambda$0(TwoThreadsSynchronizedCounter.java:14)
    - waiting to lock <0x00000000d5e660a0> (a java.lang.Object)
    at simpletest.TwoThreadsSynchronizedCounter$$Lambda$1/424058530.run(Unknown Source)
    at java.lang.Thread.run(Thread.java:745)

"Thread-7" #18 prio=5 os_prio=0 tid=0x00000000bea5000 nid=0x1ffc waiting for monitor entry [0x00000000cfcf000]
  java.lang.Thread.State: BLOCKED (on object monitor)
    at simpletest.TwoThreadsSynchronizedCounter.lambda$0(TwoThreadsSynchronizedCounter.java:14)
    - waiting to lock <0x00000000d5e660a0> (a java.lang.Object)
    at simpletest.TwoThreadsSynchronizedCounter$$Lambda$1/424058530.run(Unknown Source)
    at java.lang.Thread.run(Thread.java:745)

"Thread-6" #17 prio=5 os_prio=0 tid=0x00000000bea2000 nid=0x40c waiting for monitor entry [0x00000000cd5f000]
  java.lang.Thread.State: BLOCKED (on object monitor)
    at simpletest.TwoThreadsSynchronizedCounter.lambda$0(TwoThreadsSynchronizedCounter.java:14)
    - waiting to lock <0x00000000d5e660a0> (a java.lang.Object)
    at simpletest.TwoThreadsSynchronizedCounter$$Lambda$1/424058530.run(Unknown Source)
    at java.lang.Thread.run(Thread.java:745)

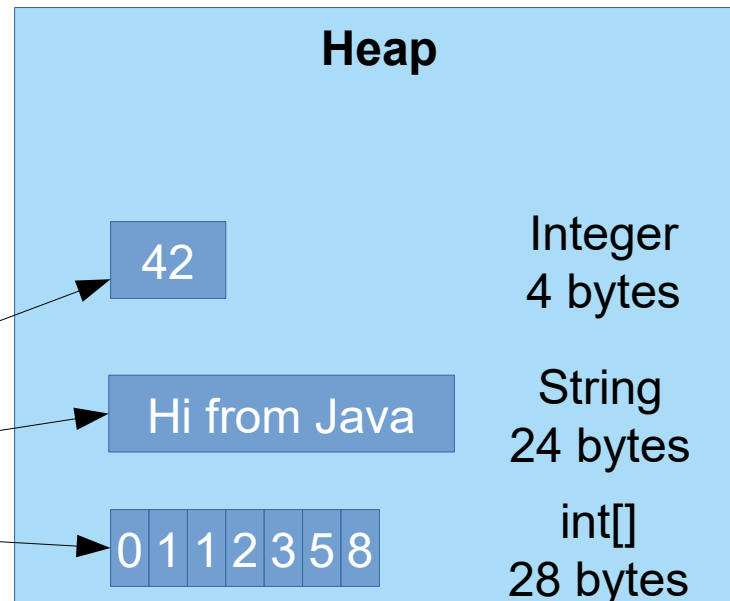
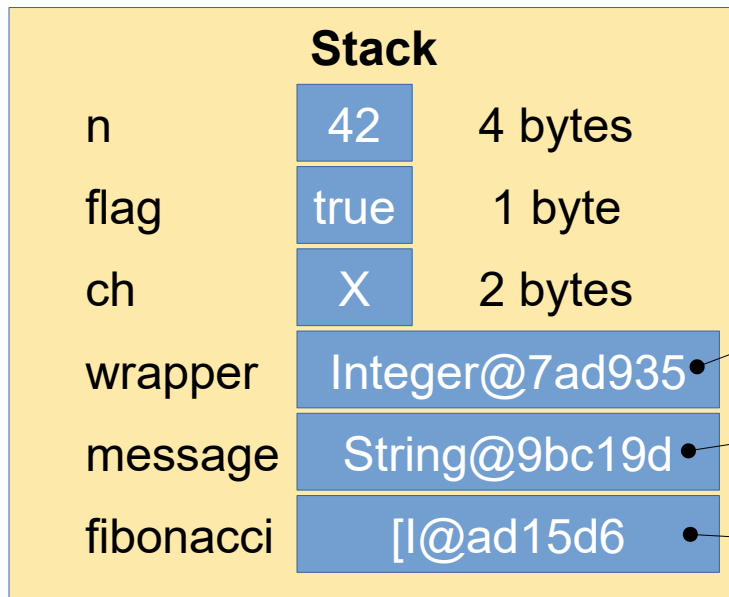
"Thread-5" #16 prio=5 os_prio=0 tid=0x00000000bea0800 nid=0x1708 waiting for monitor entry [0x00000000ceae000]
  java.lang.Thread.State: BLOCKED (on object monitor)
    at simpletest.TwoThreadsSynchronizedCounter.lambda$0(TwoThreadsSynchronizedCounter.java:14)
    - waiting to lock <0x00000000d5e660a0> (a java.lang.Object)
    at simpletest.TwoThreadsSynchronizedCounter$$Lambda$1/424058530.run(Unknown Source)
    at java.lang.Thread.run(Thread.java:745)

"Thread-4" #15 prio=5 os_prio=0 tid=0x00000000be9d000 nid=0xc0c waiting for monitor entry [0x00000000c7df000]
  java.lang.Thread.State: BLOCKED (on object monitor)
    at simpletest.TwoThreadsSynchronizedCounter.lambda$0(TwoThreadsSynchronizedCounter.java:14)
    - waiting to lock <0x00000000d5e660a0> (a java.lang.Object)
    at simpletest.TwoThreadsSynchronizedCounter$$Lambda$1/424058530.run(Unknown Source)
    at java.lang.Thread.run(Thread.java:745)

"Thread-3" #14 prio=5 os_prio=0 tid=0x00000000be9c800 nid=0x2394 waiting for monitor entry [0x00000000cc2f000]
  java.lang.Thread.State: BLOCKED (on object monitor)
```

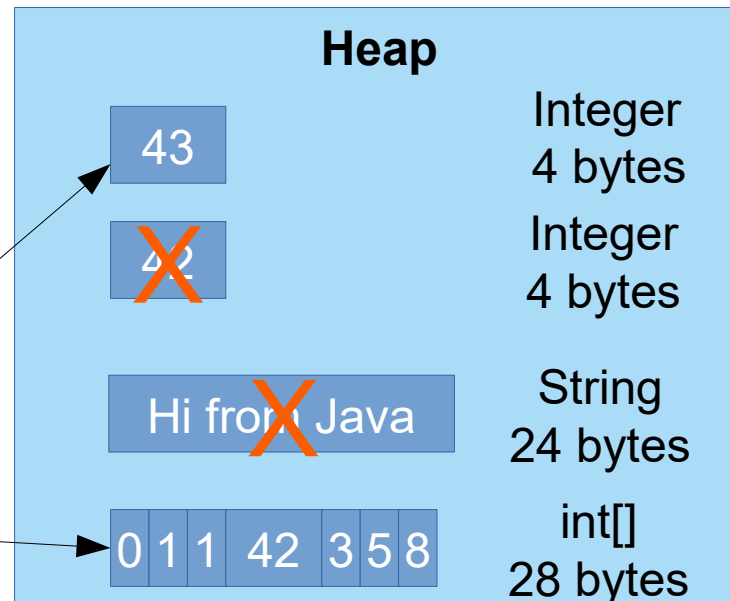
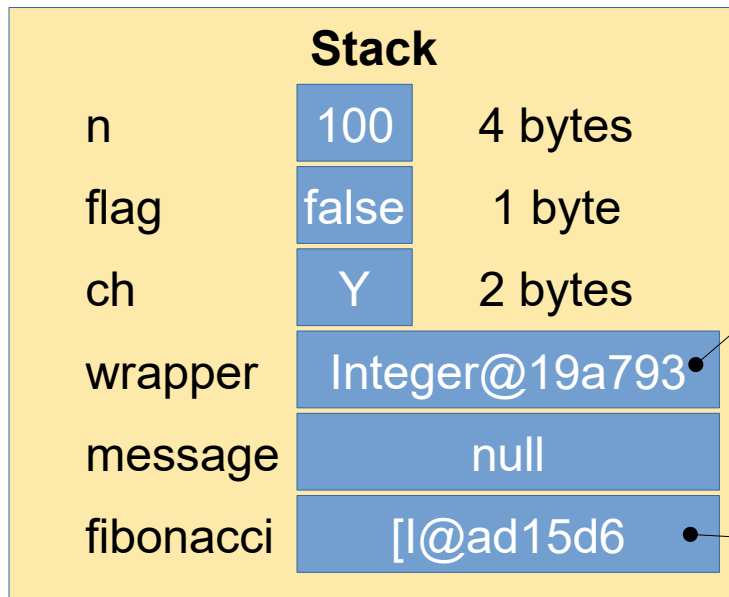
# Stack and Heap (Quick Review)

```
int n = 42;
boolean flag = true;
char ch = 'X';
Integer wrapper = n;
String message = "Hi from Java!";
int[] fibonacci = { 0, 1, 1, 2, 3, 5, 8 };
```



# Stack and Heap (Quick Review)

```
n = 100;  
flag = !flag;  
ch = ++ch;  
wrapper = ++wrapper;  
message = null;  
fibonacci[3] = 42;
```



# Variable Scopes

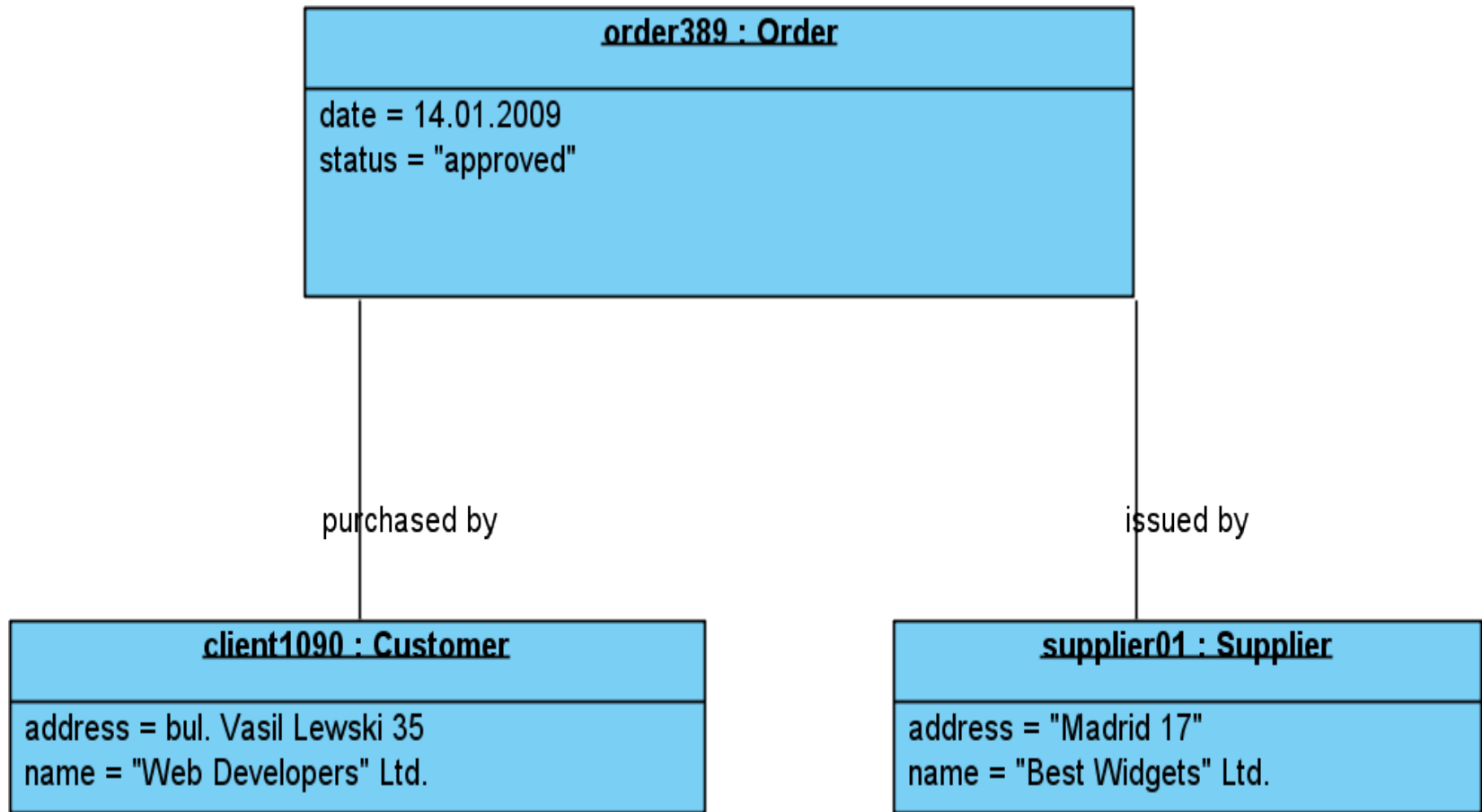
```
public class VarScopes {  
    static int s1 = 25;  
    int i1 = 350;  
    public static void main(String[] args) {  
        if(s1 > 10){  
            int a = 42;  
            // Only a available  
            {  
                int b = 108; // Both a & b are  
available  
            }  
            // Only a available, b is out of scope  
        }  
        // a & b are out of scope  
    }  
}
```



# Classes, Objects and References

- ❖ **Class** - set of objects that share a common structure, behavior and possible links to objects of other classes = objects type
  - **structure** = attributes, properties, member variables
  - **behaviour** = methods, operations, member functions, messages
  - **relations** between classes: **association, inheritance, aggregation, composition** – modeled as attributes (**references** to objects from the connected class)
- ❖ **Objects** are instances of the class, which is their addition:
  - own state
  - unique identifier = **reference** pointing towards object

# Object Diagram



# Creating Objects

- ❖ Class **String** – modeling string of characters:
  - declaration:  
`String s;`
  - initialization (on separate line):  
`s = new String("Hello Java World");`
  - declaration + initialization:  
`String s = new String("Hello Java World");`
  - declaration + initialization (shorter form, applies only to the class `String`):  
`String s = "Hello Java World";`

# Primitive and Object Data Types

❖ **Primitive** data types, **object wrapper** types and default values for attributes of primitive type

- **boolean** --> **Boolean** **false**
- **char** --> **Character** **'\u0000'**
- **byte** --> **Byte** **(byte) 0**
- **short** --> **Short** **(short) 0**
- **int** --> **Integer** **0**
- **long** --> **Long** **0L**
- **float** --> **Float** **0.0F**
- **double** --> **Double** **0.0D**
- **void** --> **Void**

❖ **BigInteger** and **BigDecimal** - higher-precision numbers

# Object (Reference) Data Types

- ❖ Creating a class (a new data type)

```
class MyClass { /* attributes and methods of the class */ }
```

- ❖ Create an object (instance) from the class MyClass :

```
MyClass myObject = new MyClass();
```

- ❖ Declaration and initialization of attributes:

```
class Person {  
    String name = "Anonymous";  
    int age;  
}
```

- ❖ Access to attribute: 

```
Person p1 = new Person();  
p1.name = "Ivan Petrov";    p1.age = 28;
```

# Object (Reference) Data Types

- ❖ Initialization with default values
- ❖ Value of uninitialized reference = **null**
- ❖ Declaring class methods

```
class Person {
```

```
    String name;
```

```
    int age;
```

```
    String setNameAndAge (String aName, int anAge) {
```

```
        name = aName;
```

```
        age = anAge;
```

```
        return "Name: " + name + "Age: " + age;
```

```
    }
```

Method Name

Arguments

Return Type

Method Body

Returning Value



# Primitive Type Literals

- ❖ in decimal notation:

**int**: 145, 2147483647, -2147483648

**long**: 145L, -1l, 9223372036854775807L

**float**: 145F, -1f, 42E-12F, 42e12f

**double**: 145D, -1d, 42E-12D, 42e12d

- ❖ in hexadecimal notation:

0x7ff, 0x7FF, 0X7ff, 0X7FF

- ❖ in octal notation: 0177

- ❖ in binary notation: 0b11100101, 0B11100101

# Operators in Java - I

- ❖ Assignment operator
- ❖ Mathematical operators
- ❖ Relational operators
- ❖ Logical operators
- ❖ Bitwise operators
- ❖ String operators
- ❖ Operators for type conversion
- ❖ Priorities of operators

# Operators in Java - II

- ❖ Each operator has priority and associativity - for example,  $+$  and  $-$  have a lower priority from  $*$  and  $/$
- ❖ The priority can be set clearly using brackets ( and ) - for example  $(y - 1) / (2 + x)$
- ❖ According associativity operators are left-associative, right-associative and non-associative: For example:  
 $x + y + z \Rightarrow (x + y) + z$ , because the operator  $+$  is left-associative
- ❖ if it was right associative, the result would be  $x + (y + z)$

# Operators in Java - III

- ❖ Assignment operator: **=**
  - is not symmetrical – i.e. **x = 42** is OK, **42 = x** is NOT
  - to the left always stands a variable of a certain type, and to the right an expression from the same type or type, which can be automatically converted to present
- ❖ Mathematical operators:
  - with one argument (unary): **-**, **++**, **--**
  - with two arguments (binary): **+**, **-**, **\***, **/**, **%** (remainder)
- ❖ Combined: **+=**, **-=**, **\*=**, **/=**, **%=**  
For example: **a += 2**  $\Leftrightarrow$  **a = a + 2**

# Send Arguments by Reference and Value

## ❖ Formal and actual arguments - Example:

Static method - no **this**

Formal Argument  
- copies the actual value

```
public static void incrementAgeBy10(Person p){  
    p.age = p.age + 10;  
}
```

```
Person p2 = new Person(23434345435L, "Petar  
Georgiev", "Plovdiv", 39);
```

```
incrementAgeBy10(p2);
```

Actual Argument

```
System.out.println(p2);
```

# Send Arguments by Reference and Value

- ❖ **Case A:** When the argument is a primitive type, the formal argument copies the actual value
- ❖ **Case B:** When the argument is a **object type**, the formal argument **copies reference** to the actual value
- ❖ **Cases A & B:** Changes in the copy (formal argument) **does not reflect** the actual argument
- ❖ However, if formal and actual argument point to the same object (**Case B**) – then **changes in properties (attribute values) of this object are available from the calling method** – i.e. we can return value from this argument



# Operators in Java - IV

- ❖ Relational operators (comparison): `==`, `!=`, `<=`, `>=`
- ❖ Logical operators: `&&` (AND), `||` (OR) and `!` (NOT)
  - the expression is calculated from left to right **only when it's necessary** for determining the final outcome
- ❖ Bitwise operators: `&` (AND), `|` (OR) and `~` (NOT),  
`^` (XOR), `&=`, `|=`, `^=`
  - bitwise shift: `<<`, `>>` (preserves character), `>>>` (always inserts zeros left – does not preserve character), `<<=`, `>>=`, `>>>=`

# Operators in Java - V

- ❖ Triple **if-then-else** operator:

**<boolean-expr> ? <then-value> : <else-value>**

- ❖ String concatenation operator: **+**

- ❖ Operators for type conversion (type casting):

**(byte), (short), (char), (int), (long), (float) ...**

- ❖ Priorities of operators:

**unary > binary arithmetical > relational > logical > three-argumentative operator if-then-else > operators to assign a value**

# Controlling Program Flow - I

- Conditional operator - **if-else**
- Returning Value – **return**
- Operators organizing cycle - **while, do while, for, break, continue**
- Operator to select one from many options - **switch**

# Controlling Program Flow - II

❖ Conditional operator **if-else**:

```
if(<boolean-expr>)  
  <then-statement>
```

or

```
if(<boolean-expr>)  
  <then-statement>  
else  
  <else-statement>
```

# Controlling Program Flow - III

- ❖ Returning value to exit the method: **return;** or **return <value>;**

- ❖ Operator to organize cycle **while**:

**while(<boolean-expr>)**  
**<body-statement>**

- ❖ Operator to organize cycle **do-while**:

**do <body-statement>**  
**while(<boolean-expr>);**

# Controlling Program Flow - IV

❖ Operator to organize cycle **for**:

**for(<initialization>; <boolean-expr>; <step>)  
<body-statement>**

❖ Operator to organize cycle **foreach**:

**for(<value-type> x : <collection-of-values>)  
<body-statement-using-x>**

Ex.: **for(Point p : pointsArray)**

**System.out.println("(" + p.x + ", " + p.y + ");");**



# Controlling Program Flow - V

- ❖ Operators to exit block (cycle) **break** and to exit iteration cycle **continue**:

```
<loop-iteration> {
```

```
    //do some work
```

```
    continue; // goes directly to next loop iteration
```

```
    //do more work
```

```
    break; // leaves the loop
```

```
    //do more work
```

```
}
```

# Controlling Program Flow - VI

❖ Use of labels with **break** and **continue**:

**outer\_label:**

```
<outer-loop> {  
    <inner-loop> {  
        //do some work  
        continue; // continues inner-loop  
        //do more work  
        break outer_label; // breaks outer-loop  
        //do more work  
        continue outer_label; // continues outer-loop  
    }  
}
```

# Controlling Program Flow - VII

❖ Selecting one of several options **switch**:

```
switch(<selector-expr>) {  
  case <value1> : <statement1>; break;  
  case <value2> : <statement2>; break;  
  case <value3> : <statement3>; break;  
  case <value4> : <statement4>; break;  
  // more cases here ...  
  default: <default-statement>;  
}
```

# Garbage Collection – Main Concepts

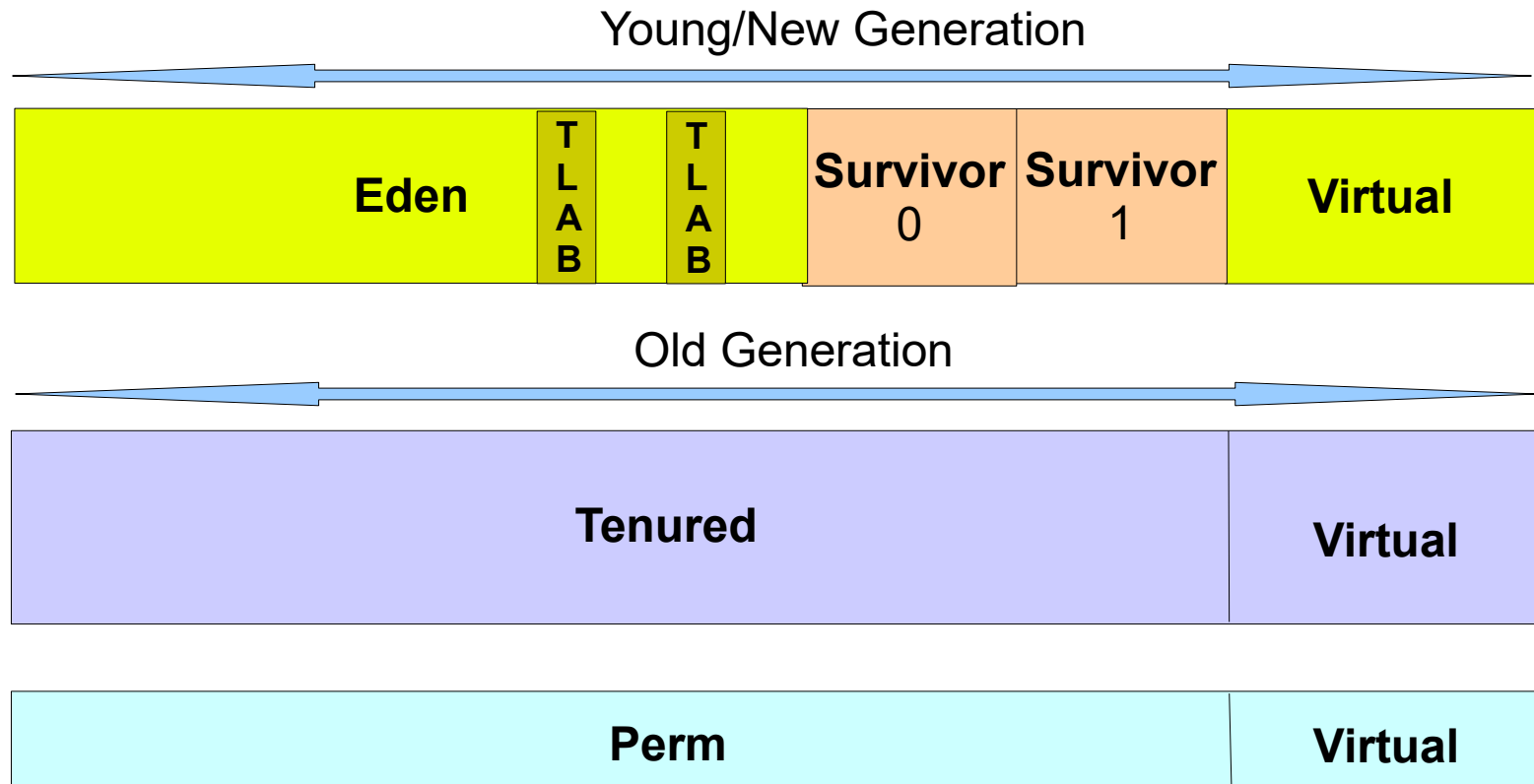
- ❖ Client and Server VMs (≠ JIT Compilers & Defaults), x86, x64
- ❖ Generational Garbage Collection – **Young, Old & ~~Permanent~~** (in Java 8 → **Metaspace**) – Weak generational hypothesis:
  - Most of the objects become unreachable soon;
  - Small number of references exist from old to young objects.
- ❖ Tuning for **Higher Throughput**:

```
java -d64 -server -XX:+AggressiveOpts -XX:+UseLargePages  
-Xmn10g -Xms26g -Xmx26g
```

- ❖ Tuning for **Lower Latency**

```
java -d64 -XX:+UseG1GC -Xms26g Xmx26g -  
XX:MaxGCPauseMillis=500 -XX:+PrintGCTimeStamp
```

# Garbage Collection – Main Concepts



# Garbage Collection – Basic Settings



- Xms** – Heap area size when starting JVM
- Xmx** – Maximum heap area size
- Xmn, -XX:NewSize** – размер на young generation (nursery)
- XX:MinHeapFreeRatio=<N>    -XX:MaxHeapFreeRatio=<N>**
- XX:NewRatio** – Ratio of New area and Old area
- XX:NewSize   -XX:MaxNewSize** – New area size  $\leq$  Max
- XX:SurvivorRatio** – Ratio of Eden area and Survivor area
- XX:+PrintTenuringDistribution** – treshold and ages of New gen
- XX:PermSize   -XX:MaxPermSize** – Initial/Max Permanent generation heap size (**not supported in Java 8**)



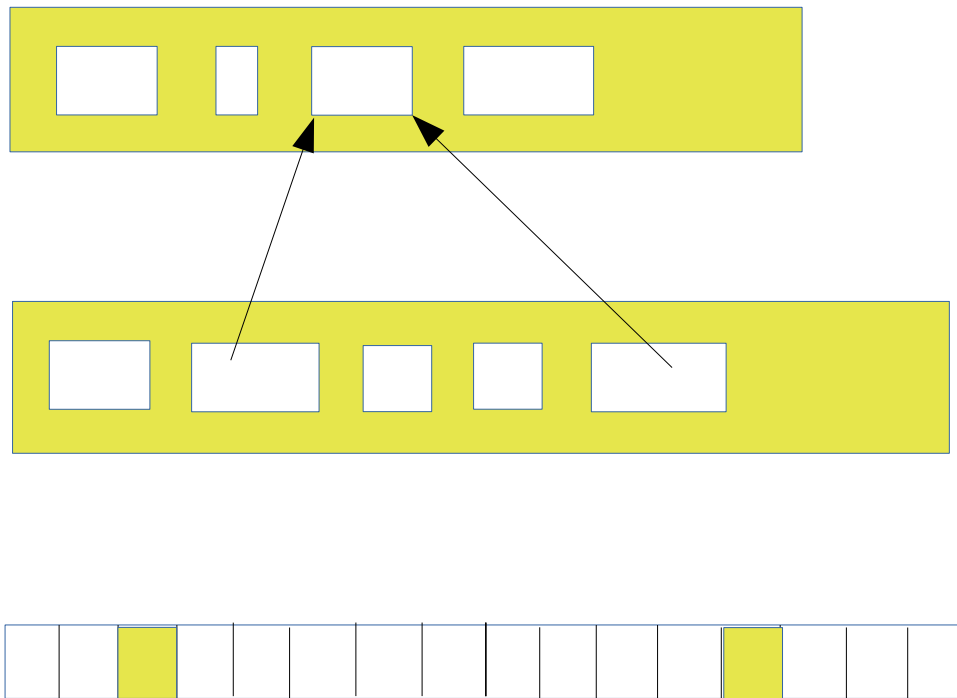
# GC Strategies and Settings

- Serial GC **-XX:+UseSerialGC**
- Parallel GC **-XX:+UseParallelGC**  
**-XX:ParallelGCThreads=<N>**
- Parallel Compacting GC **-XX:+UseParallelOldGC**
- Conc. Mark Sweep CMS GC **-XX:**  
**+UseConcMarkSweepGC**  
**-XX:+UseParNewGC**  
**-XX:+CMSParallelRemarkEnabled**  
**-XX:CMSInitiatingOccupancyFraction=<N>**  
**-XX:+UseCMSInitiatingOccupancyOnly**
- G1 **-XX:+UseG1GC**

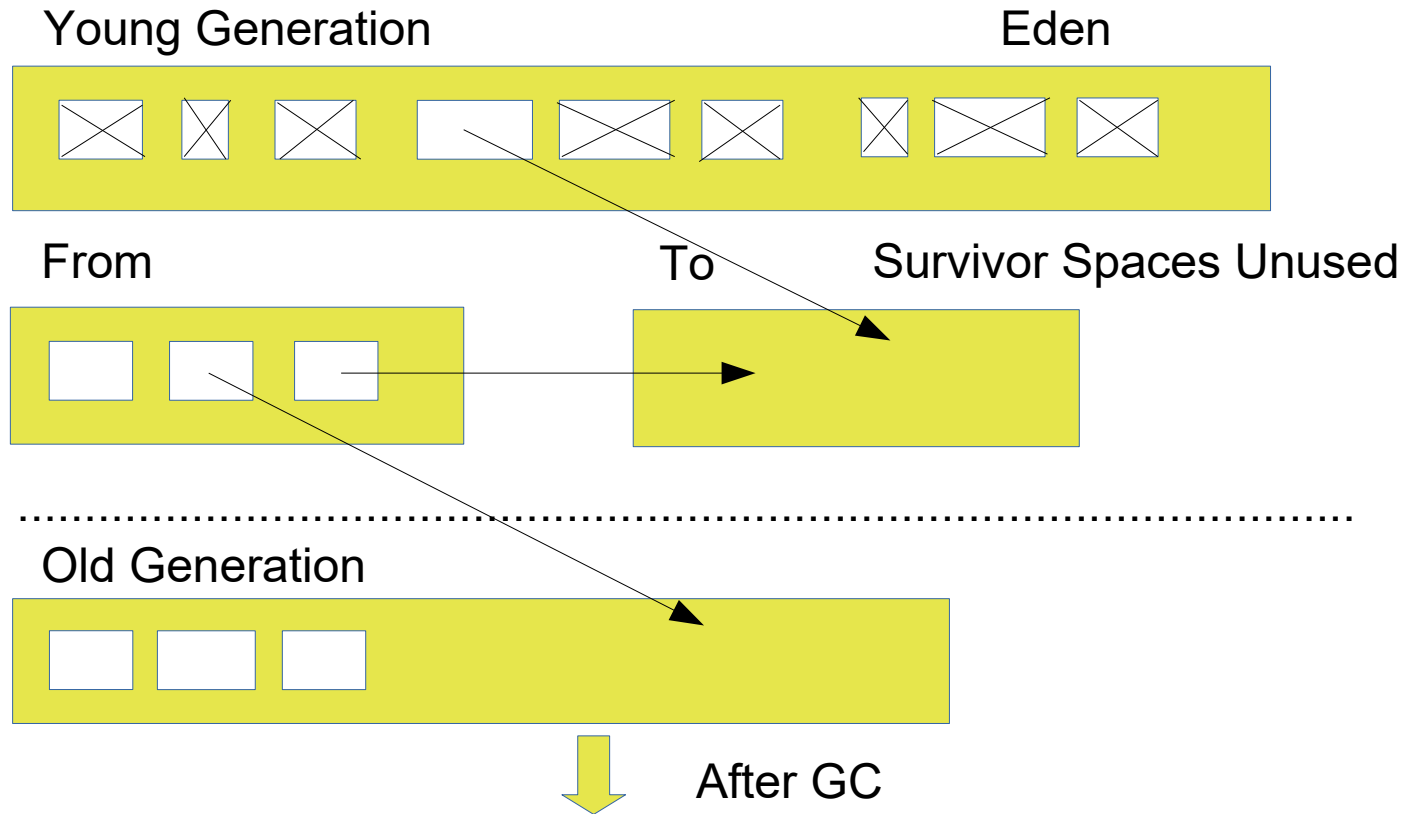
# GC Strategies and Settings

-XX:+PrintTenuringDistribution -XX:+PrintGCDetails -XX:  
+PrintGCTimeStamps

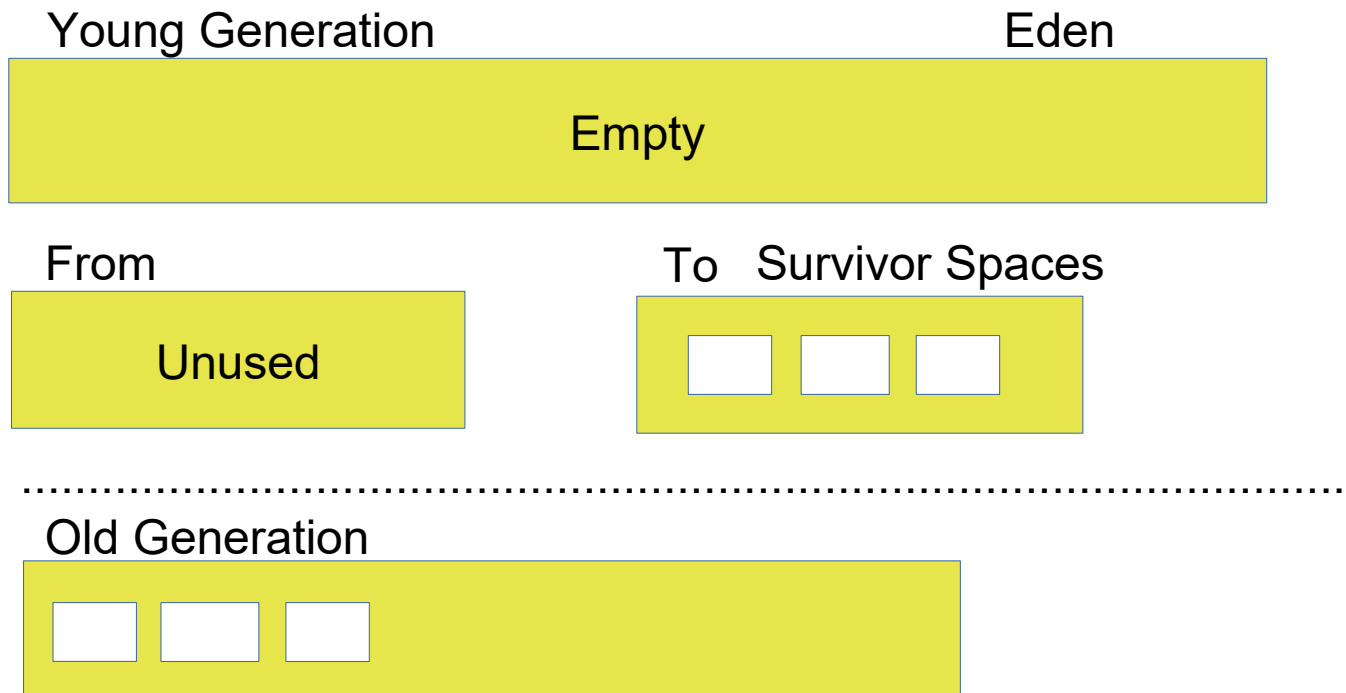
# Card Table Structure



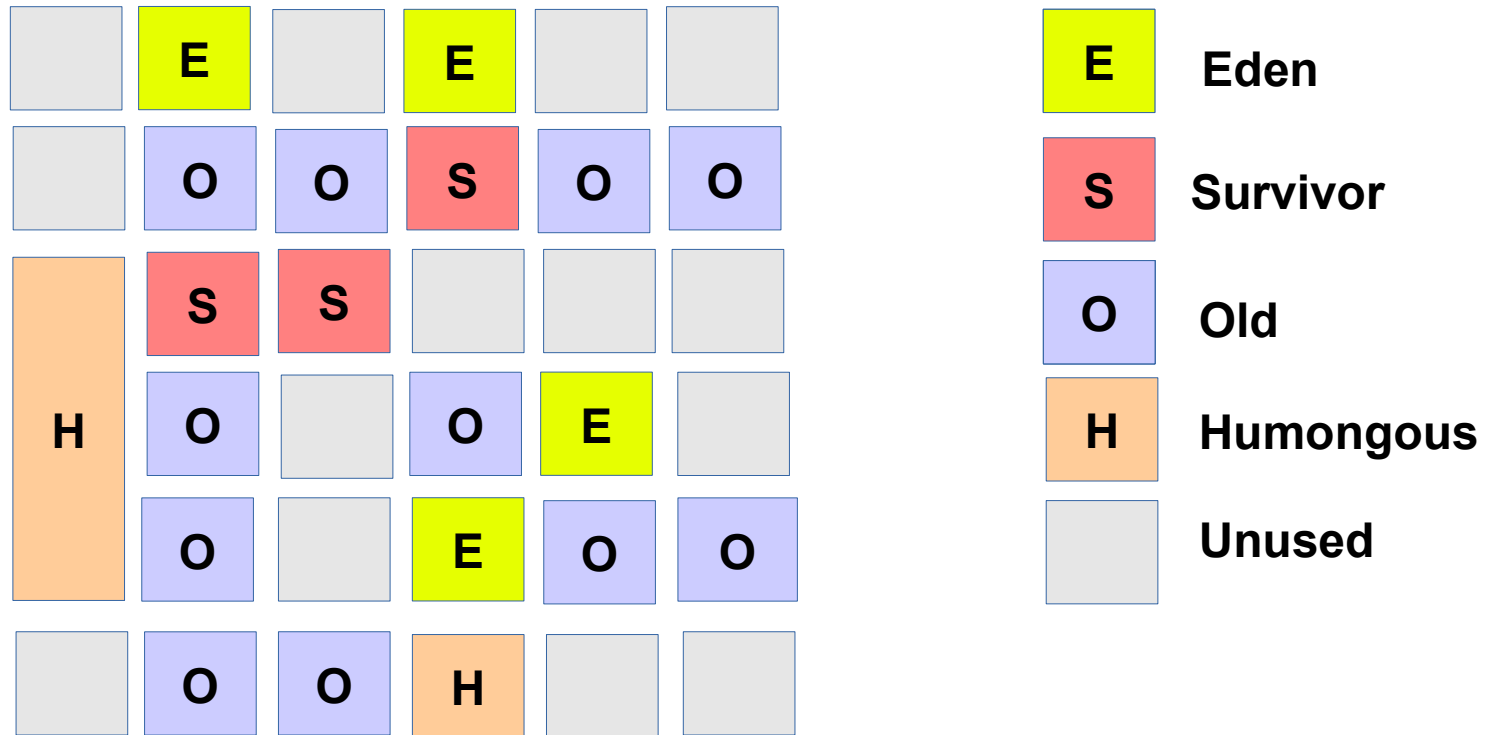
# Before GC



# After GC



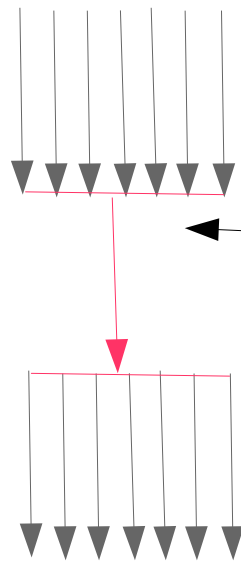
# Garbage First G1 Partially Concurrent Collector





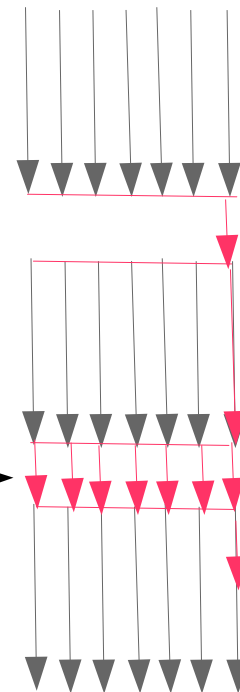
# CMS GC (-XX:+UseConcMarkSweepGC)

Serial Mark-Sweep-Compact Collector



Stop-the-world  
pause

Concurrent Mark-Sweep Collector



Initial Mark

Concurrent Mark

Remark

Concurrent Sweep

Stop-the-world  
pause

# Profiling Recommendations: GC

- ❖ **Garbage Collection** – be sure to minimize the GC interference by calling **System.gc()** several times before benchmark start. Call **System.runFinalization()** also. GC activity can be monitored using **-verbose:gc** JVM command. Another way to minimize GC interference is to use serial garbage collector using **-XX:+UseSerialGC** and same value for **-Xmx** and **-Xms**, as well as explicitly setting **-Xnm** flags.
- ❖ Use more precise **System.nanoTime()**, but be aware that the time can be reported with varying degree of accuracy in different JVM implementations.

# Java Command Line Monitoring/Tuning Tools - I

- **jps** – reports the local VM identifier (**lvmid** - typically the process identifier - **PID** for the JVM process), for each instrumented JVM found on the target system.
- **jcmd** – reports class, thread and VM information for a java process: **jcmd <PID> <command> <optional arguments>**
- **jinfo** – provides information about current system properties of the JVM and for some properties allows to be set dynamically:

**jinfo -sysprops <PID>**

**jinfo -flags <PID>**

**jinfo -flag PrintGCDetails <PID>**

**jinfo -flag -PrintGCDetails <PID>** - sets **-XX:-PrintGCDetails**

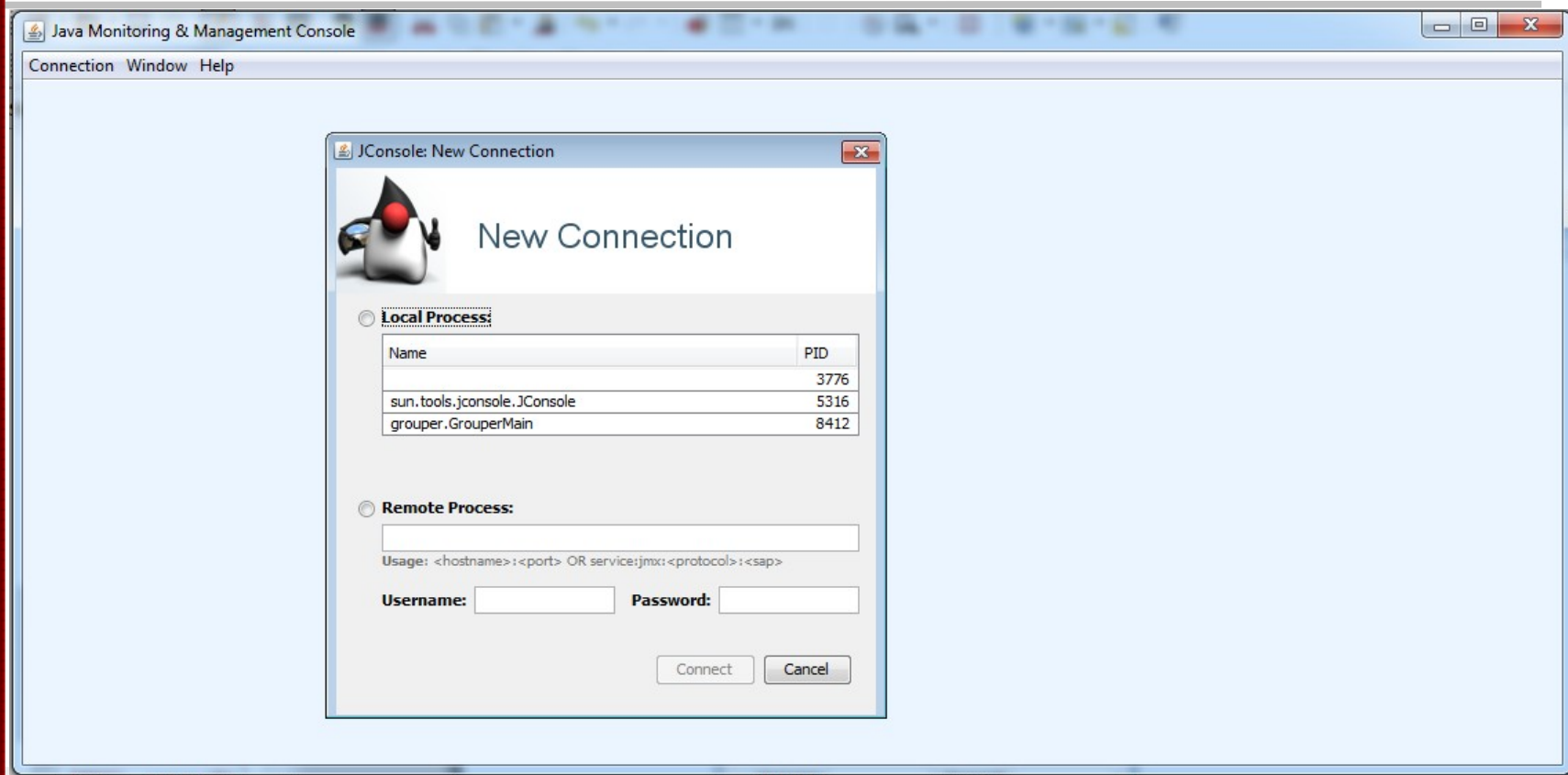
# Java Command Line Monitoring/Tuning Tools -II

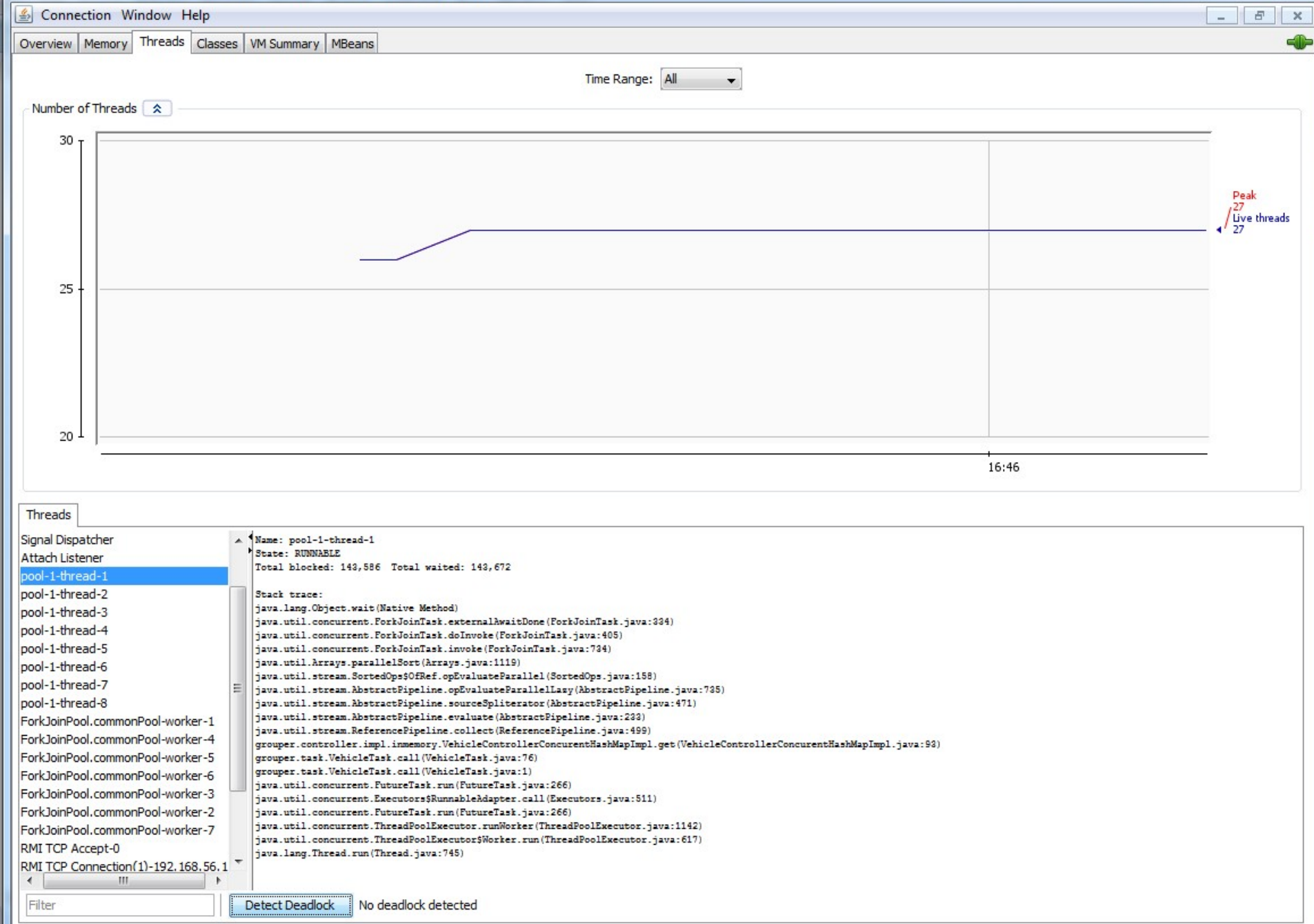
- ❖ **jstat & jstatd** – provide information about GC and class loading activities, useful for automated scripting (**jstatd** = RMI daemon):

**jstat [ generalOption | outputOptions vmid [interval[s|ms] [count]]]** **Ex: jstat -gc -t -h20 4572 2s**

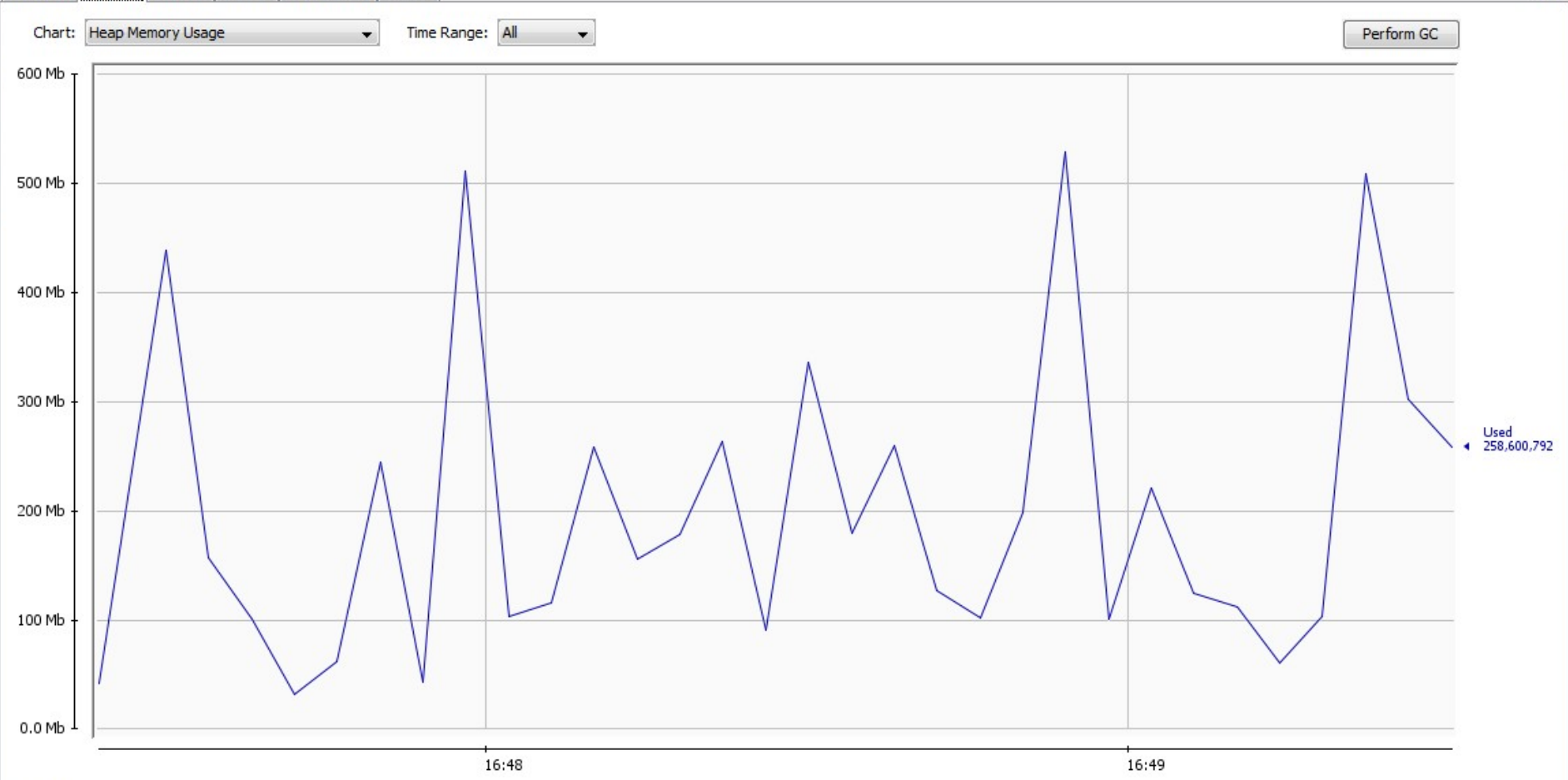
- ❖ Statistics options (part of **outputOptions**):
  - class** - statistics on the behavior of the class loader;
  - compiler** - behavior of the HotSpot Just-in-Time compiler;
  - gc** - statistics of the behavior of the garbage collected heap;
  - gccapacity** - capacities of the generations and their spaces;
  - gccause, -gcutil** - summary of garbage collection statistics/causes;
  - gcnnew, -gcnewcapacity, -gcold, -gcoldcapacity, -gcpermcapacity**
    - Young/Old/Permanent generation stats
  - printcompilation** - HotSpot compilation method statistics

## Java GUI tools – JConsole









Details

**Time:** 2015-07-16 16:49:30

**Used:** 295,252 kbytes

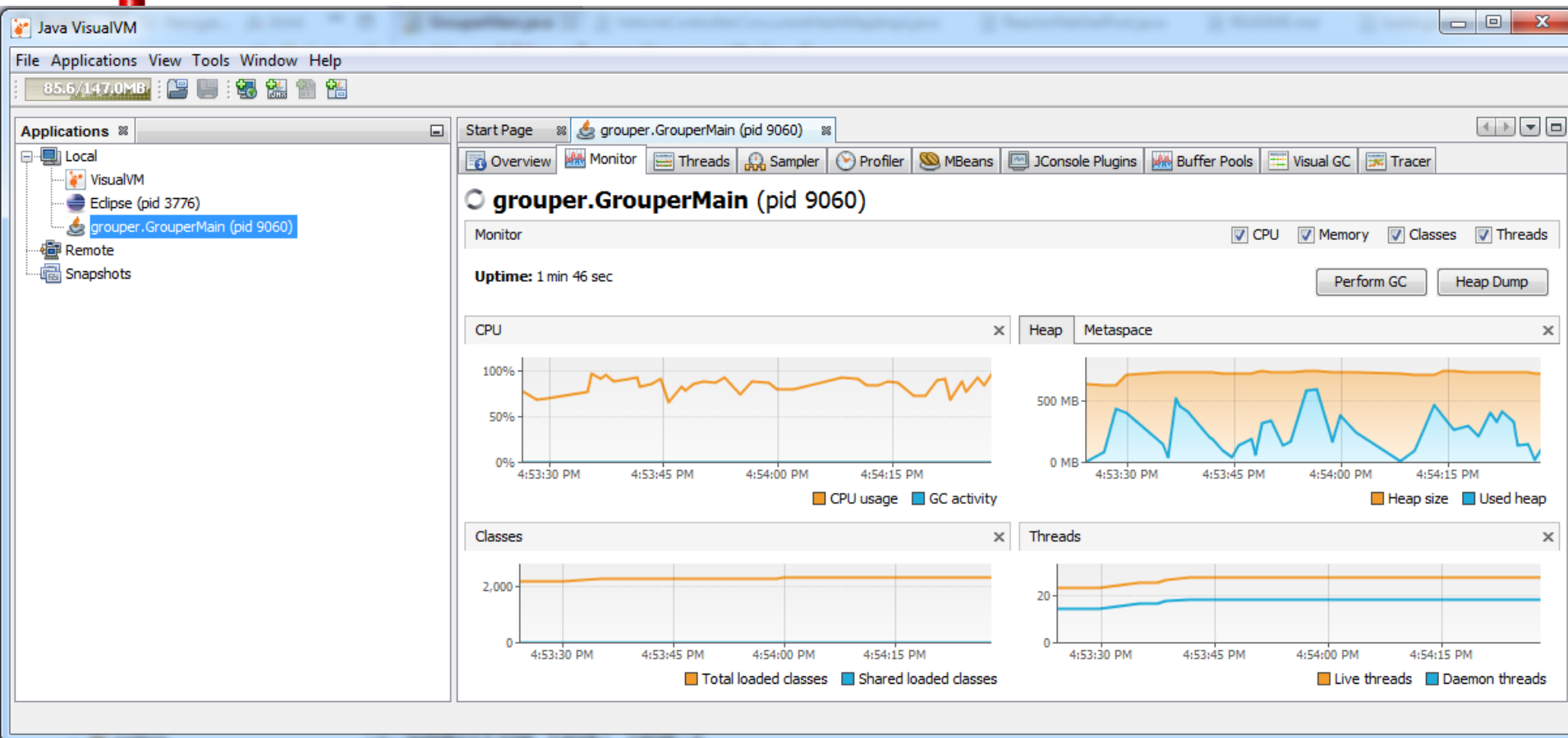
**Committed:** 474,624 kbytes

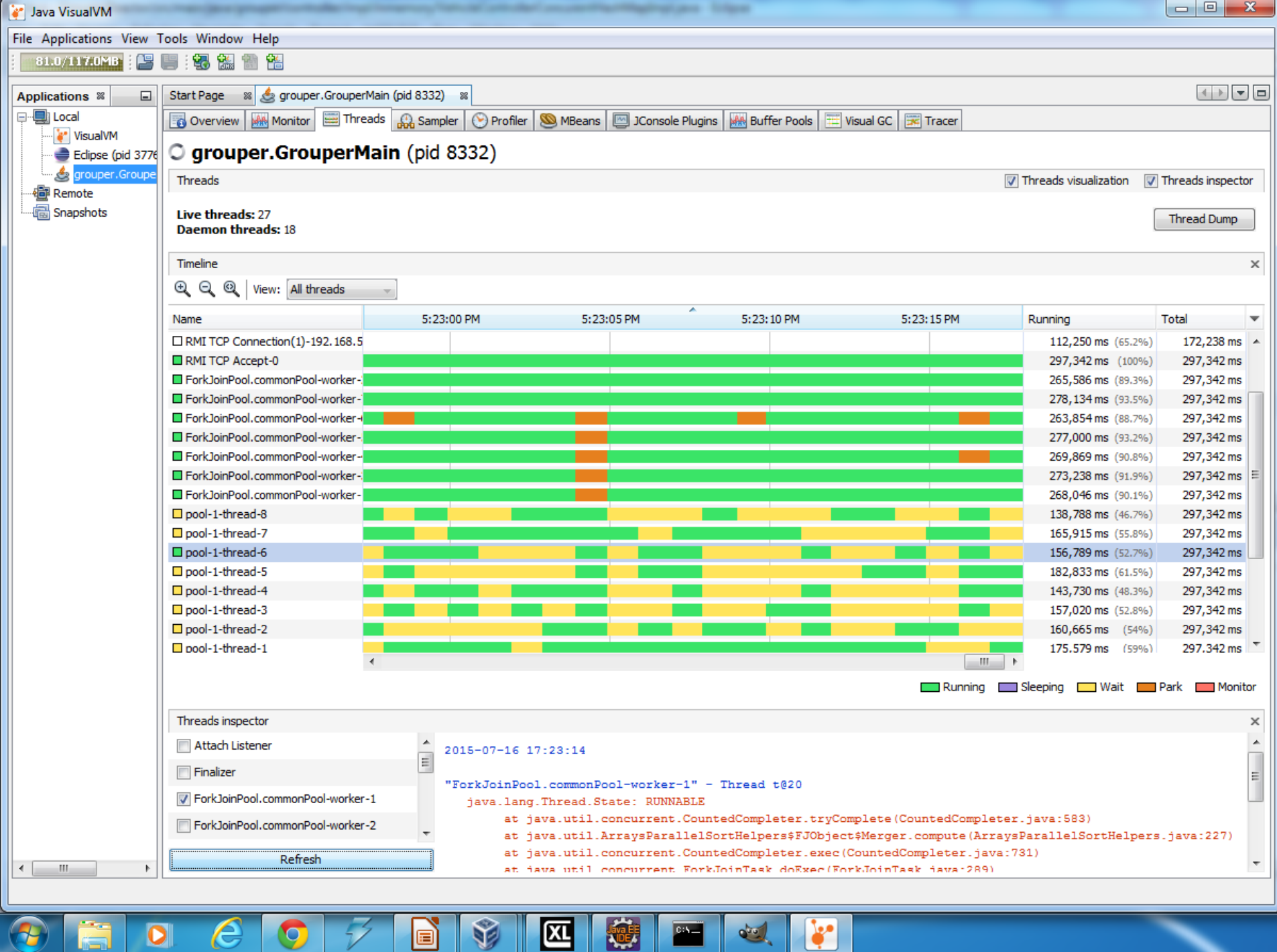
**Max:** 1,840,640 kbytes

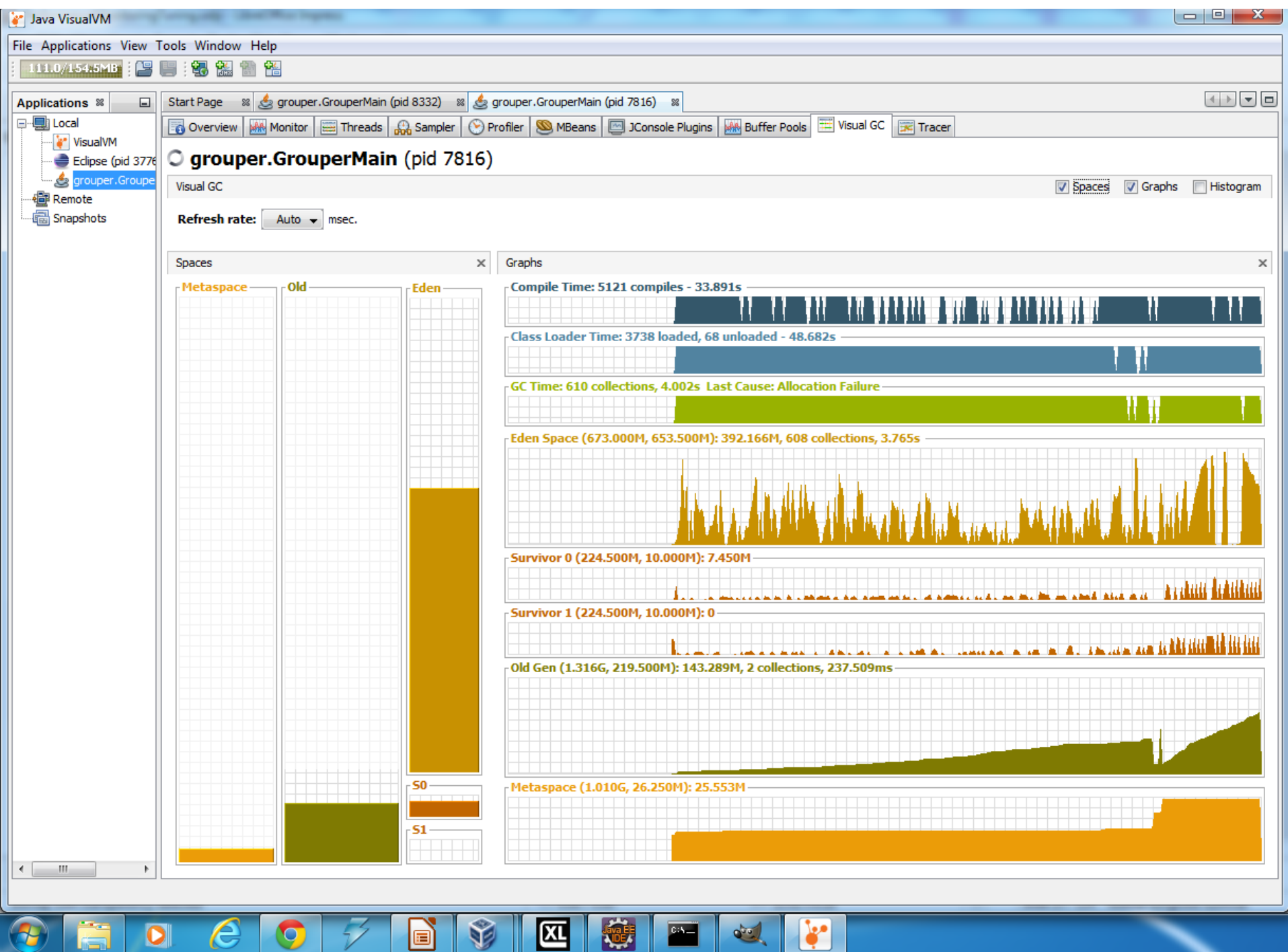
**GC time:** 0.000 seconds on PS MarkSweep (0 collections)  
1.143 seconds on PS Scavenge (455 collections)

Memory Type	Usage Percentage
Heap	~25%
Non-Heap	~95%

## Java GUI tools – jvisualvm







# Object Constructors in Java

- ❖ Initialization of objects with constructors
- ❖ **Overloading** of constructors and other methods
- ❖ Default constructors
- ❖ Reference to the current object – **this**

# Objects Initialization. Array initialization

- ❖ Initialization in declaration
- ❖ Initialization in constructor
- ❖ „Lazy“ initialization
- ❖ Initialization of static class members
- ❖ One-dimensional and multi-dimensional arrays
- ❖ Array initialization



# Class Loaders

- ❖ Bootstrap class loaders
- ❖ Extensions class loaders
- ❖ System class loaders
- ❖ Context class loaders

# Class Loaders

```
public class CustomClassLoader extends ClassLoader {

    @Override
    public Class findClass(String name) throws ClassNotFoundException {
        byte[] b = loadClassFromFile(name);
        return defineClass(name, b, 0, b.length);
    }

    private byte[] loadClassFromFile(String fileName) {
        InputStream inputStream = getClass().getClassLoader().getResourceAsStream(
            fileName.replace('.', File.separatorChar) + ".class");
        byte[] buffer;
        ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
        int nextValue = 0;
        try {
            while ( (nextValue = inputStream.read()) != -1 ) {
                byteStream.write(nextValue);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

        buffer = byteStream.toByteArray();
        return buffer;
    }
}
```

# What Modularity Means?

- ❖ Modularization is the decomposition of a system to set of **highly coherent** and **loosely coupled** modules.
- ❖ All the communication between modules is done through **well defined interfaces**.
- ❖ Modules are artifacts containing **code and metadata** describing the module.
- ❖ Each module is **uniquely identifiable** and ideally recognizable from compile-time to run-time.

# Why Modularity is Important?

- ❖ **Strong encapsulation** – separation between public and private module APIs (e.g. `sun.misc.Base64Encoder`)
- ❖ **Well-defined interfaces** and interaction protocols
- ❖ **Explicit module dependencies** and library version management => module graph
- ❖ **Reliable configuration** – no more *NoClassDefFoundError* in runtime
- ❖ **Mitigates the 'JAR / Classpath hell'** problem

# Java 9 Modularity – Project Jigsaw

- ❖ **Reliable configuration** – makes easier for developers to construct and maintain libraries and large apps
- ❖ Improved the **security and maintainability** of Java SE Platform Implementations, JDK and app libraries
- ❖ Enable improved **application performance**
- ❖ Make Java SE Platform / JDK to **scale down** for use with small devices and dense cloud deployments
- ❖ To achieve these goals, **a standard module system for the Java SE 9 Platform** was implemented.

# Project Jigsaw – JEPs and JSR

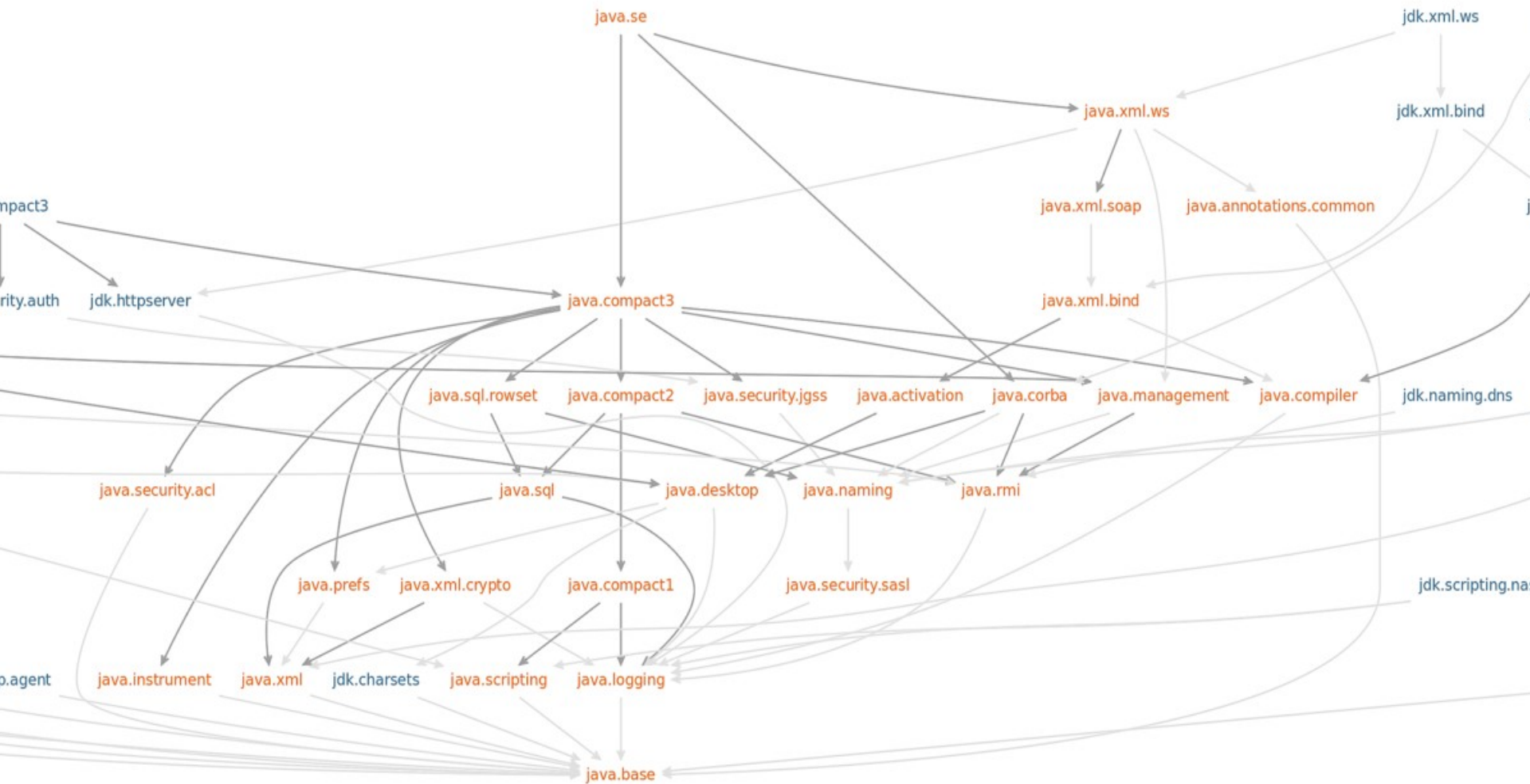
## ❖ JEPs:

- 200: Modular JDK
- 201: Modular Source Code
- 220: Modular Run-time Images
- 260: Encapsulate Most Internal APIs
- 261: Module System
- 282: jlink: Java Linker

## ❖ JSR 376 Java Platform Module System



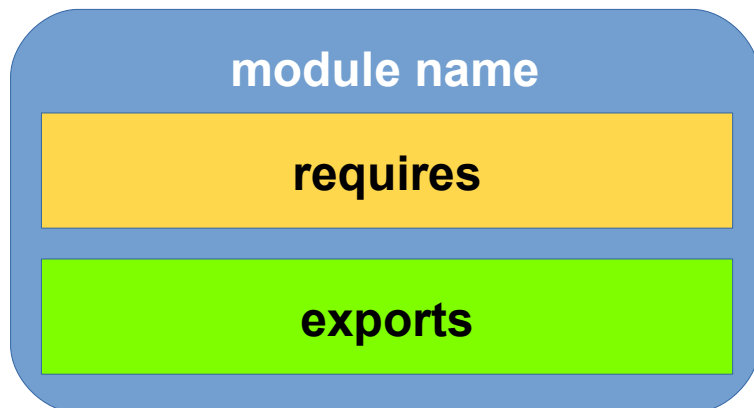
# JDK Modularization





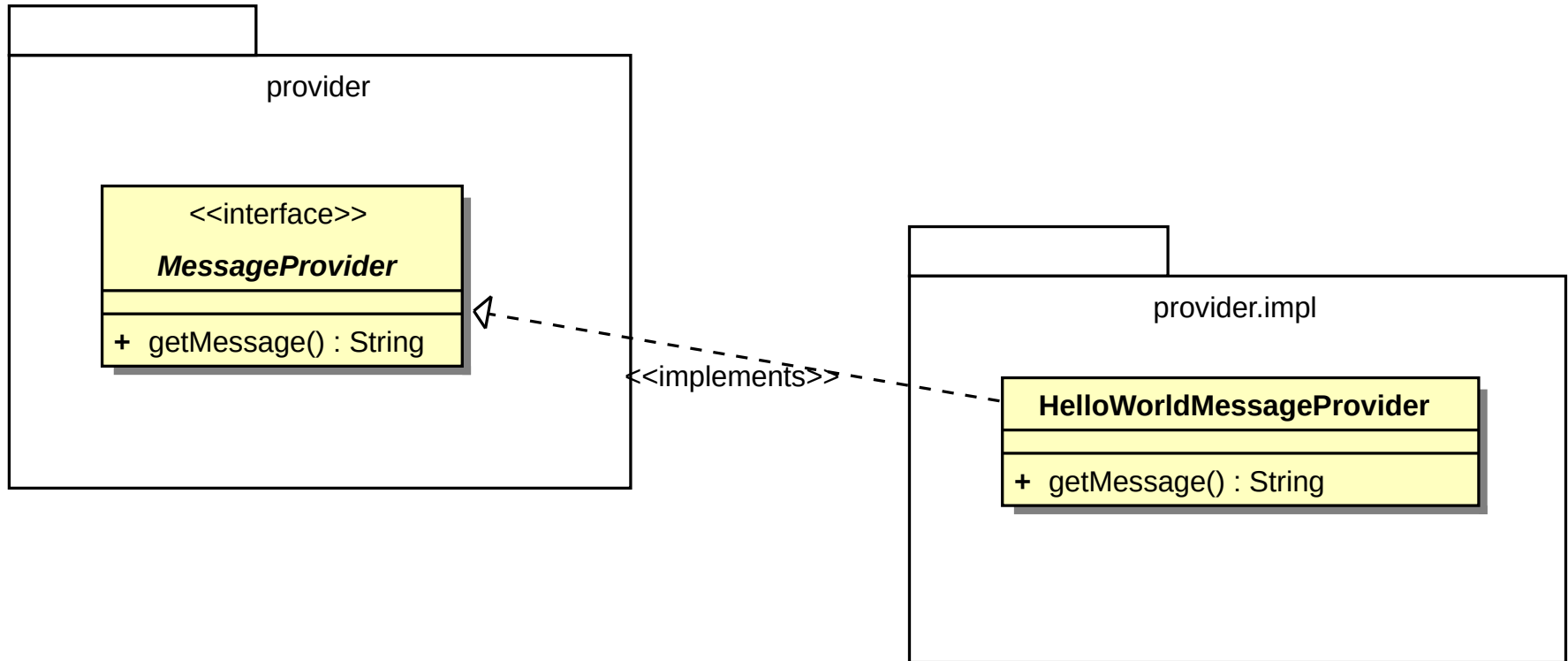
# Java 9 Modules

- ❖ Each JAR becomes a module, containing **explicit references** to other modules specified in a module metadata descriptor file called **module-info.java** available at the root of the classpath for that JAR.
- ❖ Dependencies are **available at runtime** and are eagerly **resolved before the application is started**



```
module renderer {  
    requires provider;  
    exports renderer;  
}
```

# Separation of Public & Private APIs



- ❖ *Problem:* How to publish the **MessageProvider** service interface but to hide the service implementation?

# Java 9 Accessibility Options

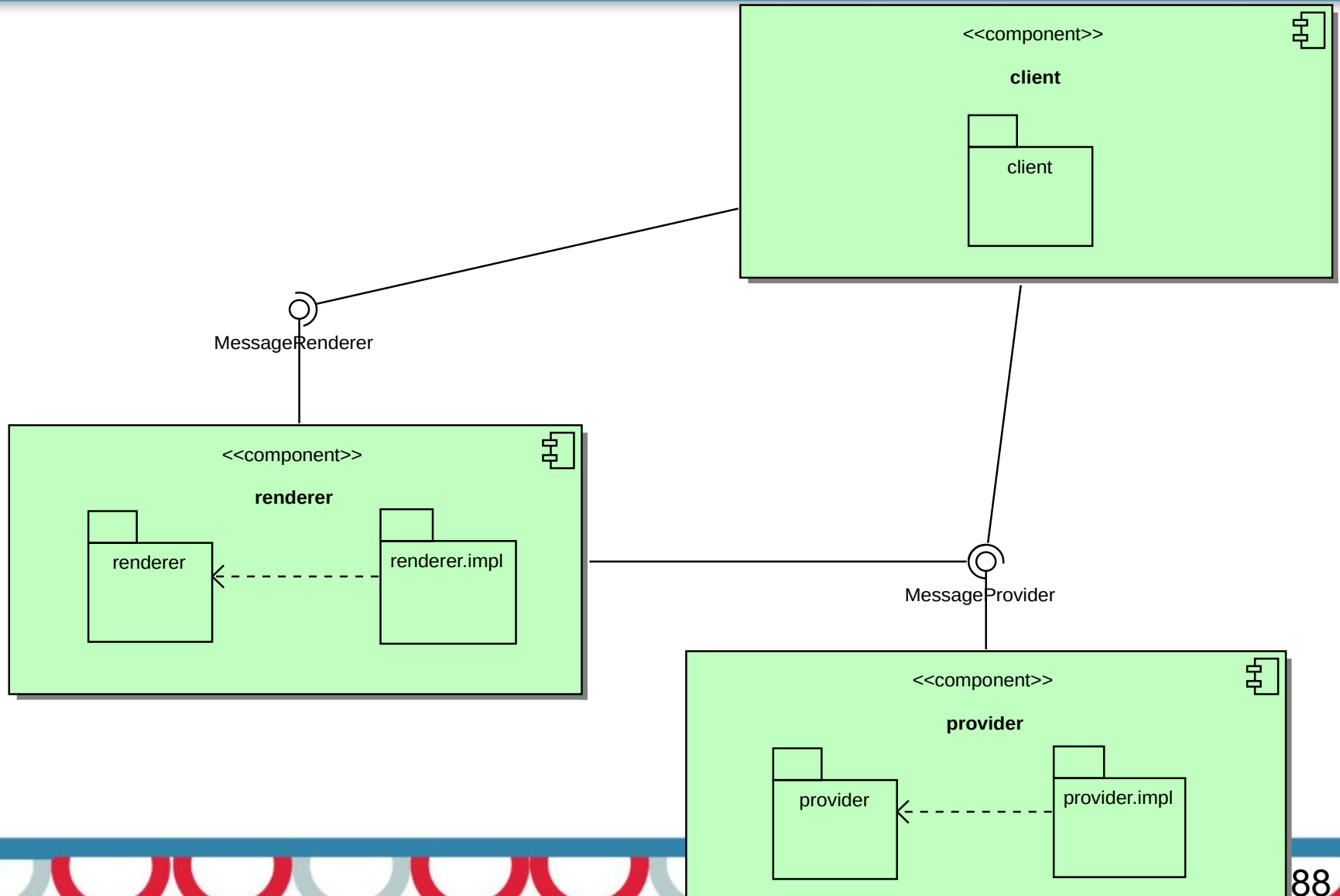
Java 1.1 – 1.8	Java 9
private	private
default (package private)	default (package private)
protected	protected
public	public within a module
	public to specific modules
	public for all modules

# Module Definition: module-info.java

[<https://www.baeldung.com/java-9-modularity>]

```
<open> module <module-name> {  
    [export <java package> [to <module name>]  
    [requires [transitive] <module-name>]  
    [opens <module name> [to <module name>]]  
    [provides <interface> with <implementation>]  
    [uses <interface>]  
}
```

# Demo App: Module Dependencies



# Demo App: Provider Module

```
| -- provider
|   |-- bin
|   |   |-- module-info.class
|   |   `-- provider
|   |       |-- MessageProvider.class
|   |       `-- impl
|   |           |-- HelloWorldMessageProvider.class
|   `-- src
|       |-- module-info.java
|       `-- provider
|           |-- MessageProvider.java
|           `-- impl
|               |-- HelloWorldMessageProvider.java
```

# Provider Module Code

```
package provider;
public interface MessageProvider {
    String getMessage();
}

package provider.impl;
import provider.MessageProvider;
public class HelloWorldMessageProvider
        implements MessageProvider {
    @Override
    public String getMessage() {
        return "Hello Java 9 Modularity!!!";
    }
}
```



# Provider Module module-info.java

```
module provider {  
    exports provider;  
    exports provider.impl;  
}
```

# Demo App: Renderer Module

```
`-- renderer
  |-- bin
  |   |-- module-info.class
  |   |-- renderer
  |   |   |-- MessageRenderer.class
  |   |   |-- impl
  |   |       |-- StandardOutMessageRenderer.class
  |-- src
  |   |-- module-info.java
  |   |-- renderer
  |   |   |-- MessageRenderer.java
  |   |   |-- impl
  |   |       |-- StandardOutMessageRenderer.java
```

# Renderer Module Interface

```
package renderer;

import provider.MessageProvider;

public interface MessageRenderer {
    void render();
    void setMessageProvider(MessageProvider provider);
    MessageProvider getMessageProvider();
}
```

# Renderer Module Implementation

```
public class StandardOutMessageRenderer
    implements MessageRenderer {
    private MessageProvider provider;

    @Override
    public void render() {
        if (provider == null)
            throw new RuntimeException("No Provider");
        System.out.println(provider.getMessage());
    }

    @Override
    public void setMessageProvider(MessageProvider
        provider) {    this.provider = provider;    }

    ...
}
```

# Renderer Module module-info.java

```
module renderer {  
    requires provider;  
    exports renderer;  
    exports renderer.impl;  
}
```

# Demo App: Client Module

```
.
|-- client
|   |-- bin
|   |   |-- client
|   |   |   `-- HelloModularityClient.class
|   |   `-- module-info.class
|   `-- src
|       |-- client
|       |   `-- HelloModularityClient.java
|       `-- module-info.java
```



# Client Module Code

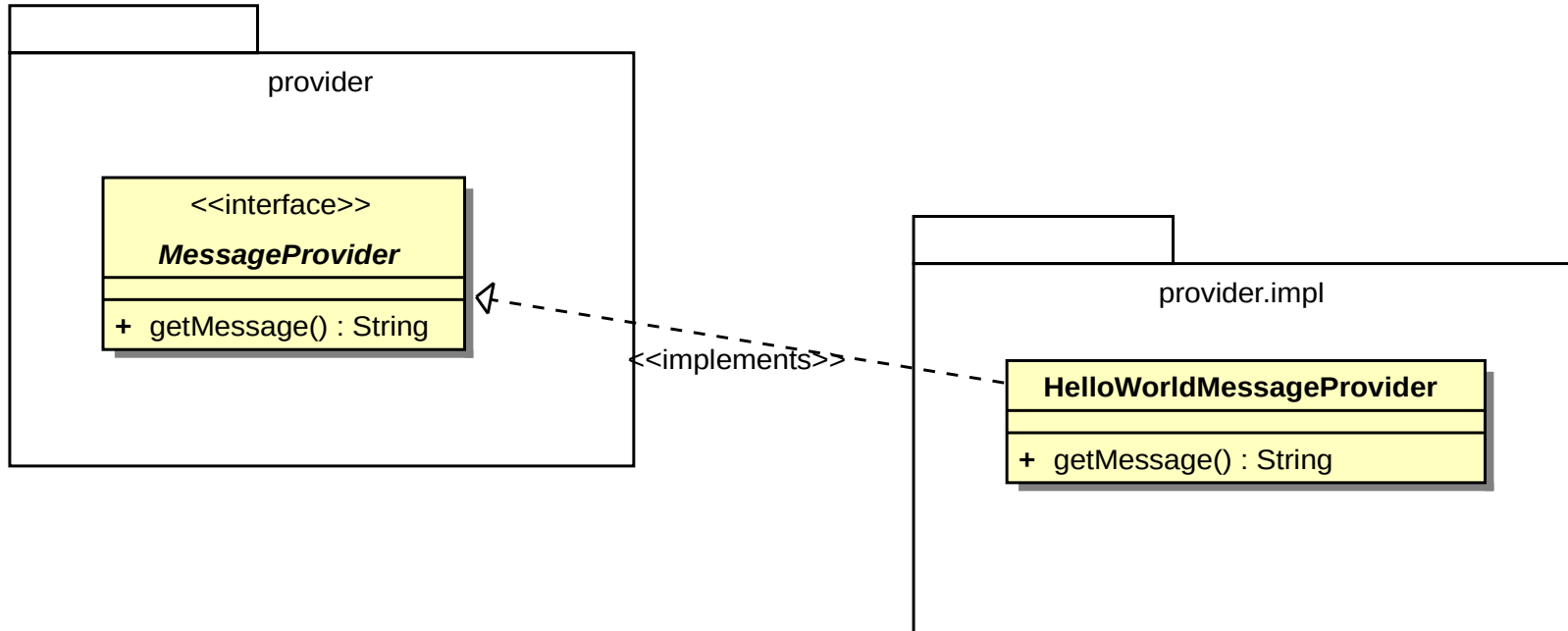
```
package client;
import provider.MessageProvider;
import renderer.MessageRenderer;
import provider.impl.HelloWorldMessageProvider;
import renderer.impl.StandardOutMessageRenderer;

public class HelloModularityClient {
    public static void main(String[] args) {
        MessageProvider provider = new MessageProvider();
        MessageRenderer renderer =
            new StandardOutMessageRenderer();
        renderer.setMessageProvider(provider);
        renderer.render();
    }
}
```

# Client Module module-info.java

```
import provider.MessageProvider;  
import renderer.MessageRenderer;  
  
module client {  
    requires provider;  
    requires renderer;  
    exports client;  
}
```

# Separation of Public & Private APIs



- ❖ *Problem:* How to publish the **MessageProvider** service interface but to hide the service implementation?
- ❖ *Solution:* **class ServiceLoader<S>**

# Provider Module module-info.java

```
module provider {  
    exports provider;  
    // exports renderer.impl;  
  
    provides provider.MessageProvider  
        with provider.impl.HelloWorldMessageProvider;  
}
```

# Renderer Module module-info.java

```
module renderer {  
    requires provider;  
    exports renderer;  
    // exports renderer.impl;  
  
    provides renderer.MessageRenderer with  
        renderer.impl.StdoutMessageRenderer;  
}
```

# Client Module module-info.java

```
import provider.MessageProvider;  
import renderer.MessageRenderer;  
  
module client {  
    requires provider;  
    requires renderer;  
    uses MessageProvider;  
    uses MessageRenderer;  
    exports client;  
}
```



# Client Module with ServiceLoader I

```
package client;
import java.util.Iterator;
import java.util.ServiceLoader;
import provider.MessageProvider;
import renderer.MessageRenderer;

public class HelloModularityClient {
    public static void main(String[] args) {
// MessageProvider provider = new MessageProvider();
ServiceLoader<MessageProvider> loaderProvider =
    ServiceLoader.Load(MessageProvider.class);
Iterator<MessageProvider> iterProvider =
    loaderProvider.iterator();
(-continue on next slide -)
```

# Client Module with ServiceLoader II

(- continued -)

```
if (!iterProvider.hasNext()) throw new
    RuntimeException("No MessageProvider!");
MessageProvider provider = iterProvider.next();

ServiceLoader<MessageRenderer> loaderRenderer =
    ServiceLoader.Load(MessageRenderer.class);
Iterator<MessageRenderer> iterRenderer =
    loaderRenderer.iterator();
    ...
MessageRenderer renderer = iterRenderer.next();

renderer.setMessageProvider(provider);
renderer.render();
}}
```

# Module Artifacts

- ❖ **Modular JAR files** – regular JAR files with **module-info.class** on top of the classpath:

```
| -- META-INF
|   `-- MANIFEST.MF
| -- module-info.class
`-- provider
    |-- MessageProvider.class
    `-- impl
        `-- HelloWorldMessageProvider.class
```

- ❖ **Exploded module directory**
- ❖ **JMOD module format** – can include native code

# Platform Modules

```
module java.base {  
    exports java.io;  
    exports java.lang;  
    exports java.lang.annotation;  
    exports java.lang.invoke;  
    exports java.lang.module;  
    exports java.lang.ref;  
    exports java.lang.reflect;  
    exports java.math;  
    exports java.net;  
    ...  
}
```

# Module Resolution

- ❖ **Module path** (**--module-path**) – different from classpath, includes platform modules + app modules, resolved upfront → acyclic module dependency graph
- ❖ **Reliable configuration** – module system ensures that each dependence is fulfilled exactly by one module
- ❖ Each **package belongs to exactly one module**
- ❖ **Faster class search** – no need to search the entire *classpath*
- ❖ **Implied readability**  
Example: **requires** **transitive** **java.logging**

# Module Types

- ❖ **System Modules** – *java --list-modules*
- ❖ **Named modules** – explicitly define **module-info**
- ❖ **Unnamed module** – if a request is made to load a type which does not belong to any observable module, module system will attempt to load it from *classpath*. It is assumed that such type belongs to *unnamed module*. **Unnamed module reads every named module, exports all its packages, but is not readable by default by named modules, use: ALL-UNNAMED**
- ❖ **Automatic modules** – the name of the module is derived from the name of jar file implicitly, **reads all modules including unnamed** → top-down migration.

# Advanced Modules: Reflection

- ❖ Each class has **Class::getModule()** method
- ❖ **Module::getDescriptor()** returns a **ModuleDescriptor**
- ❖ Can be built programmatically using **ModuleDescriptor.Builder** – e.g.:

```
ModuleDescriptor descr =  
ModuleDescriptor.newModule("org.iproductstats.core")  
    .requires("java.base")  
    .exports("org.iproduct.core.clustering")  
    .exports("org.iproduct.core.stats")  
    .packages(Set.of("org.iproduct.core.internal"))  
    .build();
```



# And More ...

- ❖ **ClassLoaders** – each module is loaded by exactly one *class loader*, but one *class loader* can load several modules, and there may be different *class loaders* for different modules.
- ❖ **Layers** – can be thought of as resolving modules in parallel module universes which can have hierarchical dependencies (parent-child) between them. There is always initial **ModuleLayer** called **boot layer**. Sophisticated app containers like application servers may load different versions of library modules / service providers for each contained app.

# Essential Module Tooling

- ❖ **jdeps** – allows to obtain crucial information about module inter-dependencies – e.g:

```
jdeps -jdkinternals --module-path modules  
modules/client.jar
```

- ❖ **jlink** – produces run-able image containing only used JDK and application modules (~70 MB) – e.g:

```
jlink -p "%JAVA_HOME%\jmods";modules --add-  
modules client --output small-image --bind-  
services --launcher start-app=client --  
strip-debug -compress=2
```

- ❖ **jmod** – produces mixed java native code modules

# Java 9 Process API Updates

- ❖ Retrieve PID of Current Process:

```
long pid = ProcessHandle.current().pid();
```

- ❖ Checking if process is running:

```
Optional<ProcessHandle> handle = ProcessHandle.of(pid);  
boolean isAlive = processHandle.isPresent() &&  
    processHandle.get().isAlive();
```

- ❖ Retrieving process information

```
ProcessHandle.Info info = handle.get().info();  
System.out.println("CPU time: " +  
    info.totalCpuDuration().orElse(null));
```

- ❖ Running post-termination code, getting children, etc.

# Immutable Collections

- ❖ **List**, **Set** and **Map** have been added new factory methods for immutable collections:
- ❖ **of(...)** factory methods for **Set** and **List**, one with varargs parameters.
- ❖ **of(...)** factory methods for **Map** with key and value arguments, one with varargs of Entry type **ofEntries(Entry<? extends K, ? extends V>... entries)**
- ❖ Returned collections are instances of nested types defined under **java.util.ImmutableCollections**

# Stack-Walking API (JEP 259)

- ❖ Replaces now deprecated **sun.reflect.Reflection** with **StackWalker** class:
- ❖ **public <T> T walk(Function<Stream<StackFrame>,T> function)** – traverses the stackframes of the current thread as stream and applying the given function
- ❖ **public Class<?> getCallerClass()** – returns the invoking class

# Reactive Streams Spec.

- ❖ **Reactive Streams** – provides standard for **asynchronous stream processing** with non-blocking back pressure.
- ❖ Minimal set of interfaces, methods and protocols for asynchronous data streams
- ❖ April 30, 2015: has been released version 1.0.0 of **Reactive Streams for the JVM** (Java API, Specification, TCK and implementation examples)
- ❖ Java 9: **`java.util.concurrent.Flow`**



# Reactive Streams Spec.

- ❖ **Publisher** – provider of potentially unbounded number of sequenced elements, according to Subscriber(s) demand.

`Publisher.subscribe(Subscriber) => onSubscribe onNext*  
(onError | onComplete)?`

- ❖ **Subscriber** – calls **Subscription.request(long)** to receive notifications
- ❖ **Subscription** – one-to-one **Subscriber ↔ Publisher**, request data and cancel demand (allow cleanup).
- ❖ **Processor** = **Subscriber** + **Publisher**



# Futures in Java 8 - I

- ❖ **Future** (implemented by **FutureTask**) – represents the result of an cancelable asynchronous computation. Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation (blocking till its ready).
- ❖ **RunnableFuture** – a Future that is Runnable. Successful execution of the **run** method causes Future completion, and allows access to its results.
- ❖ **ScheduledFuture** – delayed cancelable action that returns result. Usually a scheduled future is the result of scheduling a task with a **ScheduledExecutorService**

# Future Use Example

```
Future<String> future = executor.submit(  
    new Callable<String>() {  
        public String call() {  
            return searchService.findByTags(tags);  
        }  
    }  
);  
  
DoSomethingOther();  
  
try {  
    showResult(future.get()); // use future result  
} catch (ExecutionException ex) { cleanup(); }
```

# Futures in Java 8 - II

- ❖ **CompletableFuture** – a Future that may be explicitly completed (by setting its value and status), and may be used as a **CompletionStage**, supporting dependent functions and actions that trigger upon its completion.
- ❖ **CompletionStage** – a stage of possibly **asynchronous** computation, that is triggered by completion of previous stage or stages. A stage performs an action or computes value and completes, triggering next **dependent stages**. Computation may be **Function** (**apply**), **Consumer** (**accept**), or **Runnable** (**run**).

# CompletableFuture Example - III

```
try {  
    System.out.println(results.get(10, TimeUnit.SECONDS));  
} catch (ExecutionException | TimeoutException  
        | InterruptedException e) {  
    e.printStackTrace();  
}  
executor.shutdown();  
}
```

```
// OR just:  
System.out.println(results.join());  
executor.shutdown();
```



Which is better?

# CompletionStage

- ❖ Computation may be **Function** (**apply**), **Consumer** (**accept**), or **Runnable** (**run**) – e.g.:

```
completionStage.thenApply( x -> x * x )  
                .thenAccept(System.out::print )  
                .thenRun( System.out::println )
```

- ❖ Stage computation can be triggered by completion of 1 (**then**), 2 (**combine**), or either 1 of 2 (**either**)
- ❖ Functional composition can be applied to stages themselves instead to their results using **compose**
- ❖ **handle** & **whenComplete** – support unconditional computation – both normal or exceptional triggering

# CompletionStages Composition

```
public void test1CompletableFutureComposition() throws
InterruptedException, ExecutionException {
    Double priceInEuro = CompletableFuture.supplyAsync(
        () -> getStockPrice("GOOGL") )
        .thenCombine(CompletableFuture.supplyAsync(
            () -> getExchangeRate(USD, EUR)), this::convertPrice)
        .exceptionally(throwable -> {
            System.out.println("Error: " + throwable.getMessage());
            return -1d;
        }).get();

    System.out.println("GOOGL stock price in Euro: " +
        priceInEuro );
}
```



# New in Java 9: CompletableFuture

- ❖ Executor **defaultExecutor()**
- ❖ CompletableFuture<U> **newIncompleteFuture()**
- ❖ CompletableFuture<T> **copy()**
- ❖ CompletionStage<T> **minimalCompletionStage()**
- ❖ CompletableFuture<T> **completeAsync**(  
Supplier<? extends T> supplier[, Executor executor])
- ❖ CompletableFuture<T> **orTimeout**(  
long timeout, TimeUnit unit)
- ❖ CompletableFuture<T> **completeOnTimeout**(  
T value, long timeout, TimeUnit unit)



# Async HTTP/2 & WebSocket clients

## ❖ Why HTTP/2?

- Header compression and binary encoding
- bidirectional communication using push requests
- multiplexing within a single TCP connection
- long running connections

## ❖ *module-info.java* :

```
module org.iproduct.demo.profiler.client {  
    requires java.se;  
    requires jdk.incubator.httpclient;  
    requires gson;  
    exports org.iproduct.demo.profiler.client;  
    exports org.iproduct.demo.profiler.client.model;  
    opens org.iproduct.demo.profiler.client.model to gson;  
}
```

# Async HTTP/2 Client Example I

```
HttpClient client = HttpClient.newHttpClient();

HttpRequest processesReq = HttpRequest.newBuilder()
    .uri(new URI(PROFILER_API_URL + "processes"))
    .GET()
    .build();

TypeToken<ArrayList<ProcessInfo>> token =
    new TypeToken<ArrayList<ProcessInfo>>() {};

Gson gson = new GsonBuilder().create();
```

# Async HTTP/2 Client Example II

```
client.sendAsync(processesReq, HttpResponse.BodyHandler.asString())
    .thenApply( (HttpResponse<String> processesStr) -> {
        List<ProcessInfo> something =
            gson.fromJson(processesStr.body(), token.getType());
        return something;
    }).thenApply(proc -> {
        proc.stream().forEach(System.out::println);
        return null;
    }).exceptionally((Throwable ex) -> {
        System.out.println("Error: " + ex);
        return null;
    }).thenRun(() -> {System.exit(0);});

Thread.sleep(5000);
```

# More Demos ...

Java 9 modules, `CompletableFuture`,... @ GitHub:

<https://github.com/iproduct/reactive-demos-java-9>

- ❖ `modularity-demo` – Java 9 modules in action :)
- ❖ `http2-client` – Java 9 modules + HTTP/2 client + GSON
- ❖ `completable-future-demo` – composition, delayed, ...
- ❖ `flow-demo` – custom Flow implementations using CFs
- ❖ `completable-future-jaxrs-cdi-cxf` – async observers, ...
- ❖ `completable-future-jaxrs-cdi-jersey`

# Thank's for Your Attention!



**Trayan Iliev**

**CEO of IPT – Intellectual Products  
& Technologies**

<http://iproduct.org/>

<http://robolearn.org/>

<https://github.com/iproduct>

<https://twitter.com/trayaniliev>

<https://www.facebook.com/IPT.EACAD>

<https://plus.google.com/+IproductOrg>