

February 2022,  
Programming in Java

# Programming in Java

Trayan Iliev  
[tiliev@iproduct.org](mailto:tiliev@iproduct.org)  
<http://iproduct.org>

Copyright © 2003-2022 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

# Course Schedule

- ❖ Block 1: 9:00 – 11:00
- ❖ Pause: 11:00 – 11:15
- ❖ Block 2: 11:15 – 13:15

# Where to Find the Code?

**Java Web Development** projects and examples are available @ GitHub:

<https://github.com/iproduct/java-fundamentals-2022>

# Agenda for This Session

- **Java Class structure** – package, imports, fields, methods, access modifiers;
- **Creating objects** – constructors, order of initialization, static members, keyword this, constructors overloading;
- **Working with methods** – designing methods, arguments and return values, overloading, static methods, access modifiers;
- **Define the scope of variables** – class(static), local, instance variables;
- **Apply encapsulation** principles to a class;
- **Understand objects equality** – the difference between “==” and equals();
- **Wrapper Classes**;
- **Distinguish between Object reference and primitive variables**, type casting; Methods reference and primitive arguments;
- **Enumerations**;
- **Object lifecycle** – destroying objects, garbage collection – finalize();

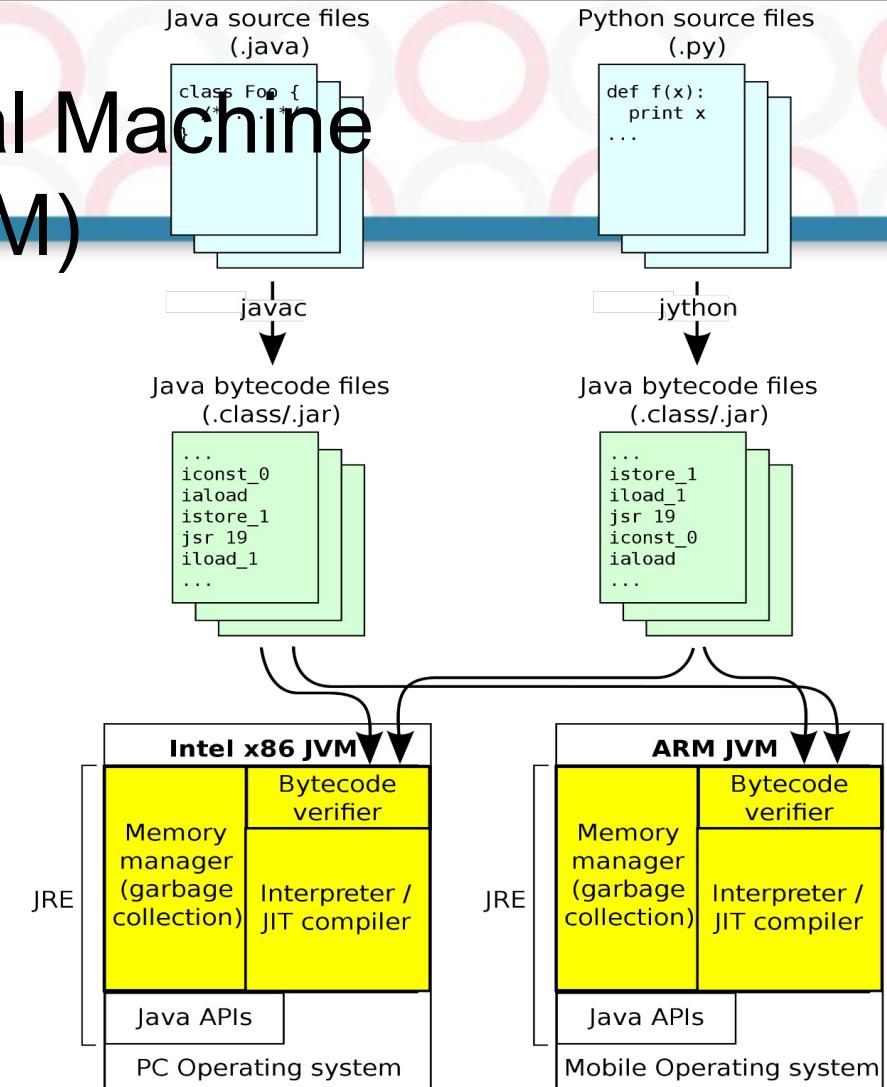
# Key Features of Java Language

- **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: IntelliJ IDEA, Eclipse

# Java Virtual Machine (JVM)



Source: [http://en.wikipedia.org/wiki/File:Java\\_virtual\\_machine\\_architecture.svg](http://en.wikipedia.org/wiki/File:Java_virtual_machine_architecture.svg)

# Java Application Stack

**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

Level of Optimization ↓

# Classes, Objects and References

- **Class** - set of objects that share a common structure, behaviour 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 (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;

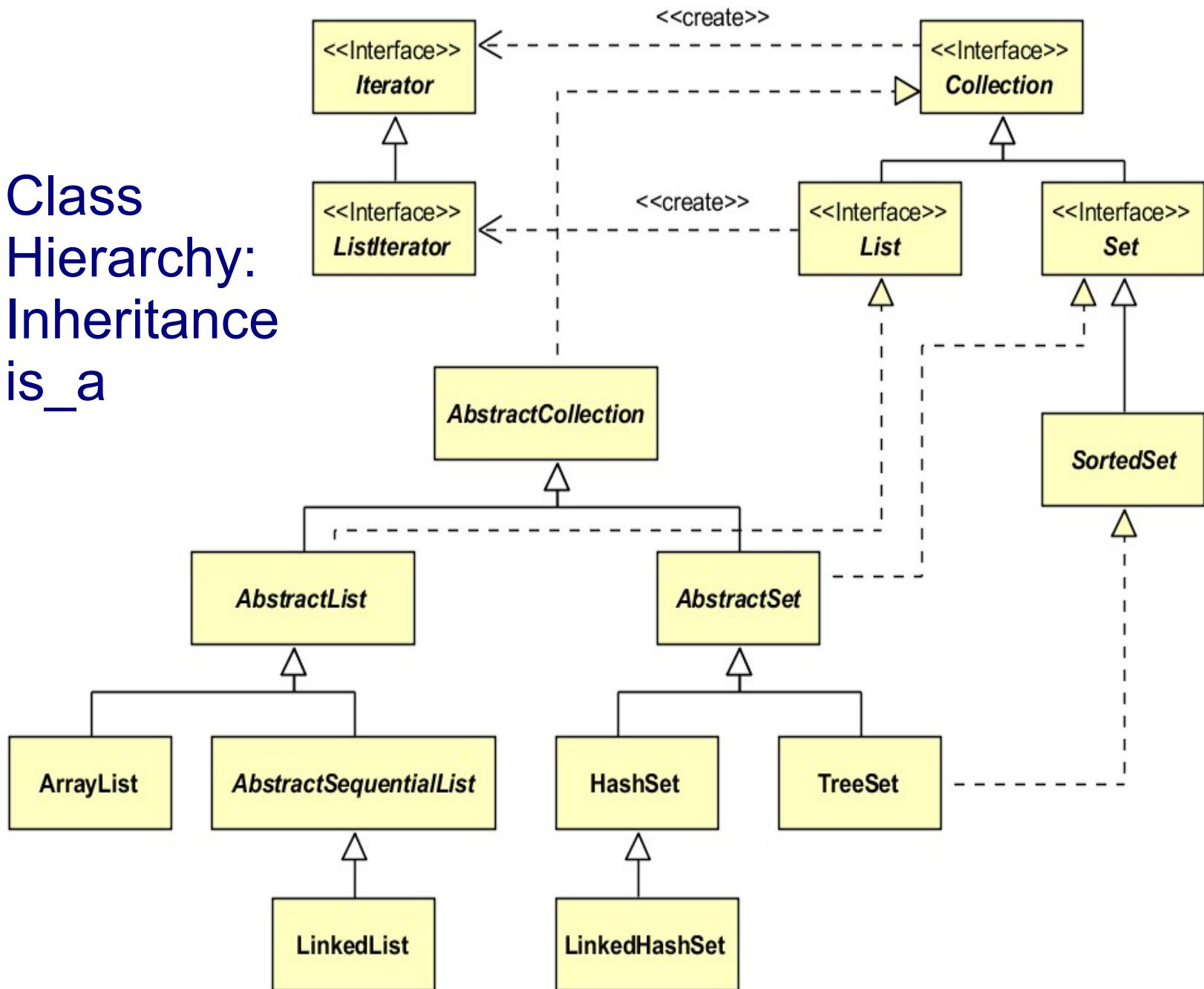
# Creating Objects

- 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";`

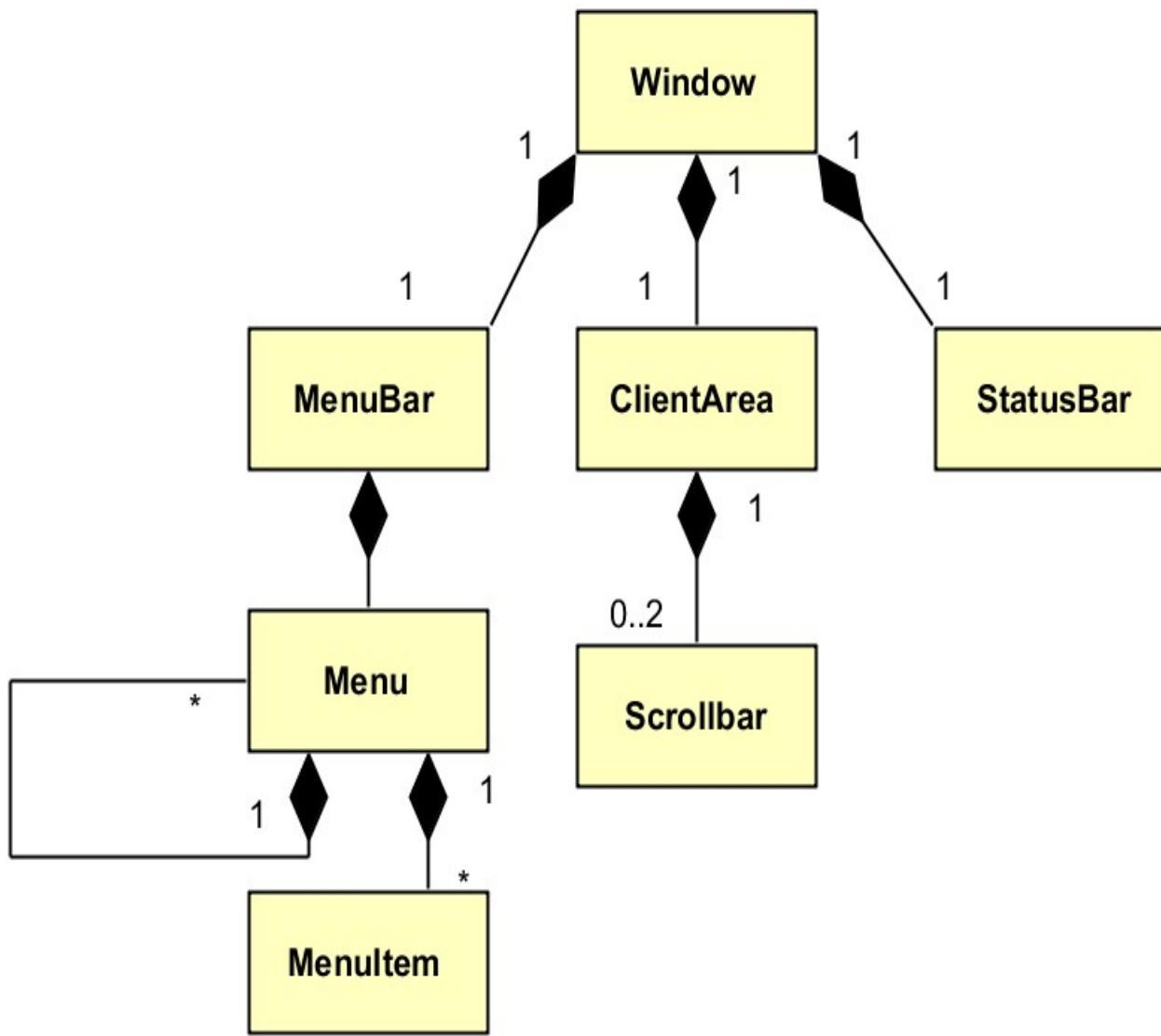
# SOLID Design Principles of OOP

1. **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.
2. **Open–closed principle** - software entities should be open for extension, but closed for modification.
3. **Liskov substitution principle** - Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
4. **Interface segregation principle** - Many client-specific interfaces are better than one general-purpose interface.
5. **Dependency inversion principle** - depend upon abstractions, not concretions.

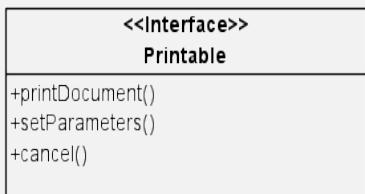
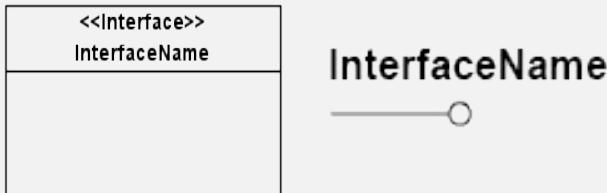
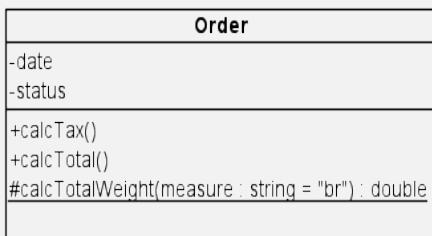
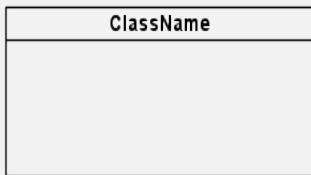
# Class Hierarchy: Inheritance is\_a



# Object Hierarchy: Composition, has\_a



# Elements of Class Diagrams

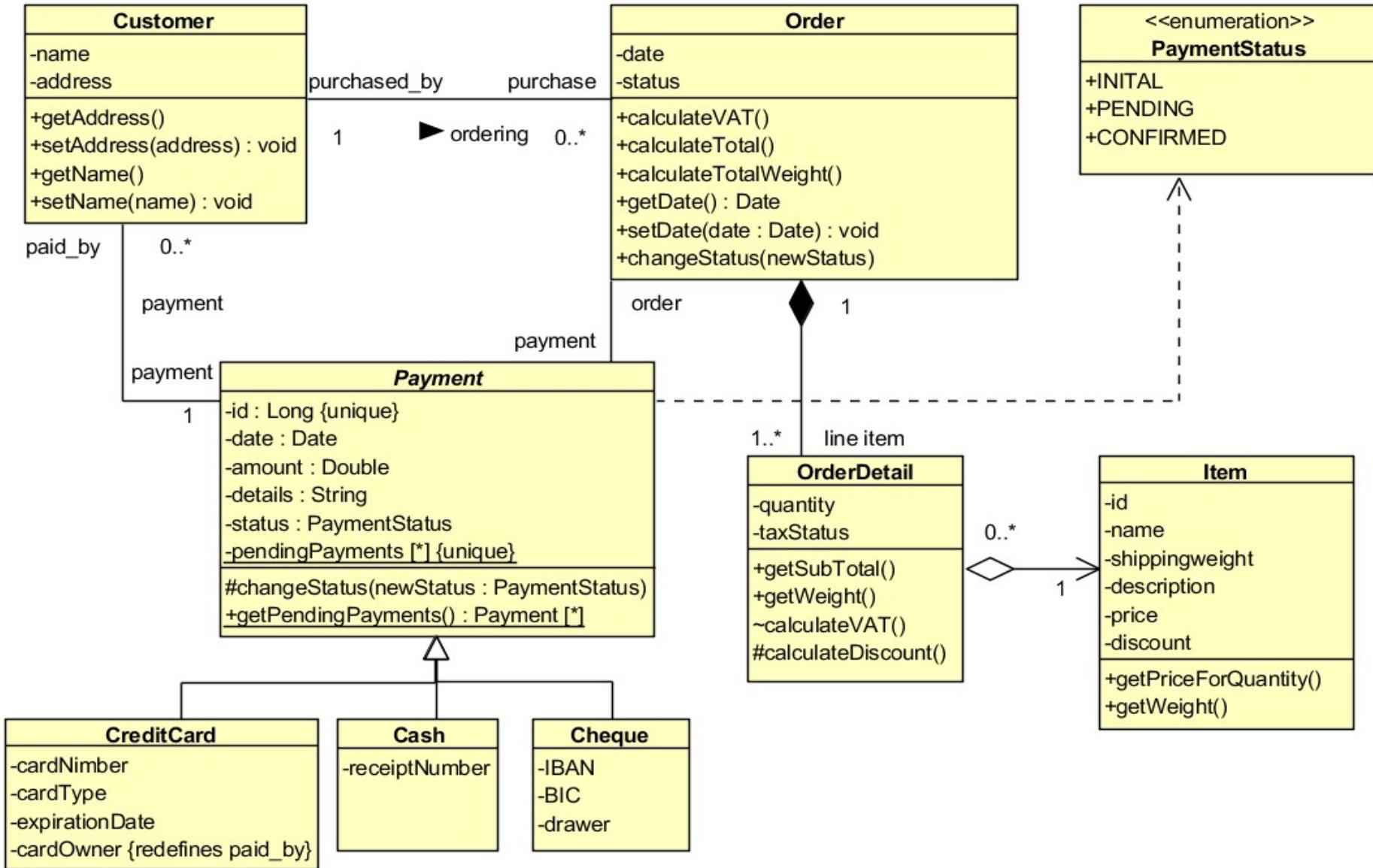


Types of connections:

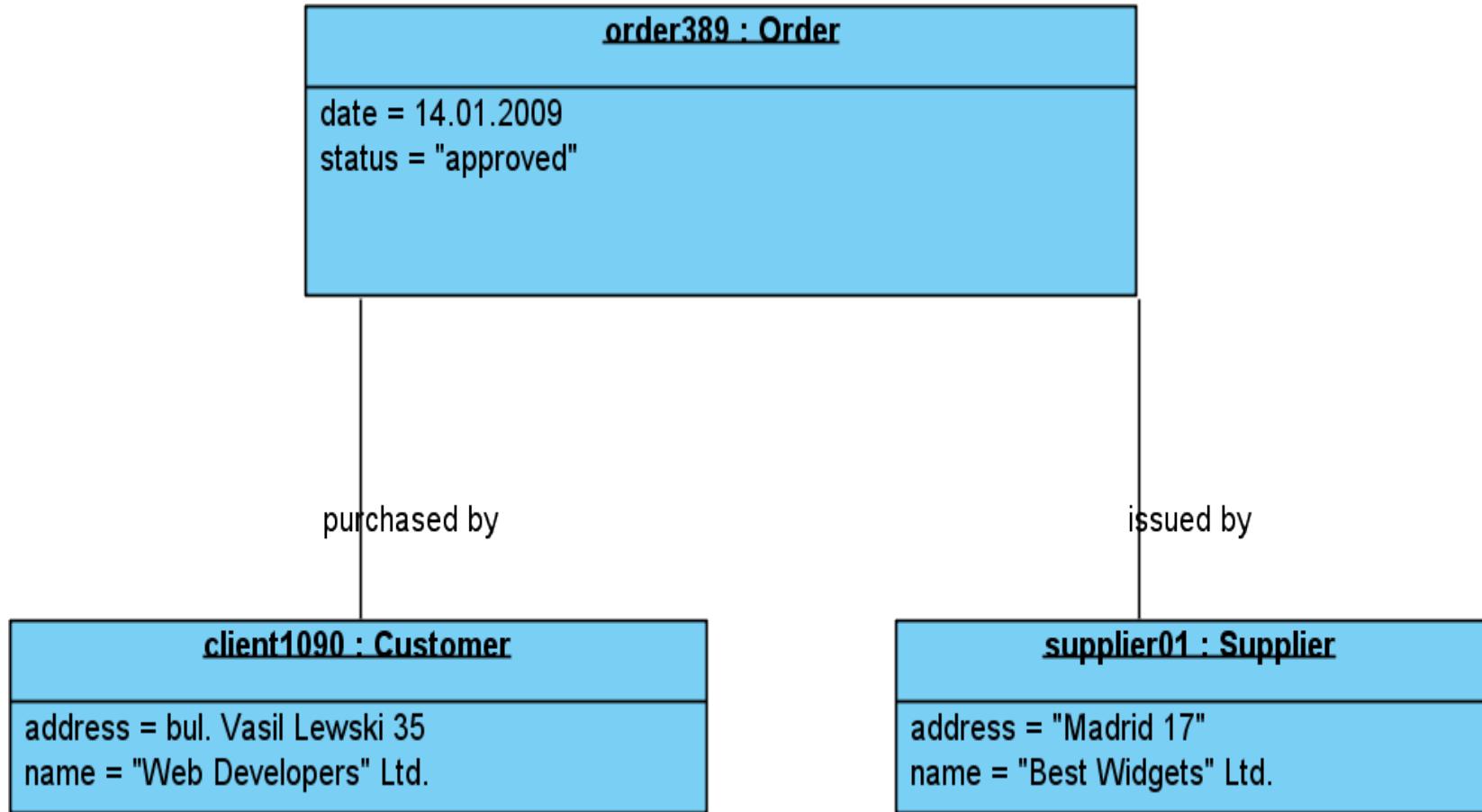
- Association
- aggregation
- composition
- dependence
- generalization
- realization



# Class Diagram



# Object Diagram



# Packages and Access Specifiers

- ❖ Packages and directories
- ❖ Importing packages – import
- ❖ Access specifiers
  - **public**
  - **private**
  - **protected**
  - Friendly access – by default within the package

# 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

# 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

# Object (Reference) Data Types

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

```
class Person {
```

```
    String name;
```

```
    int age;
```

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

```
        Return Type
```

Method Name

Arguments

```
        name = aName;
```

```
        age = anAge;
```

Method Body

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

```
}
```

```
}
```

Returning Value

# Object Constructors in Java

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

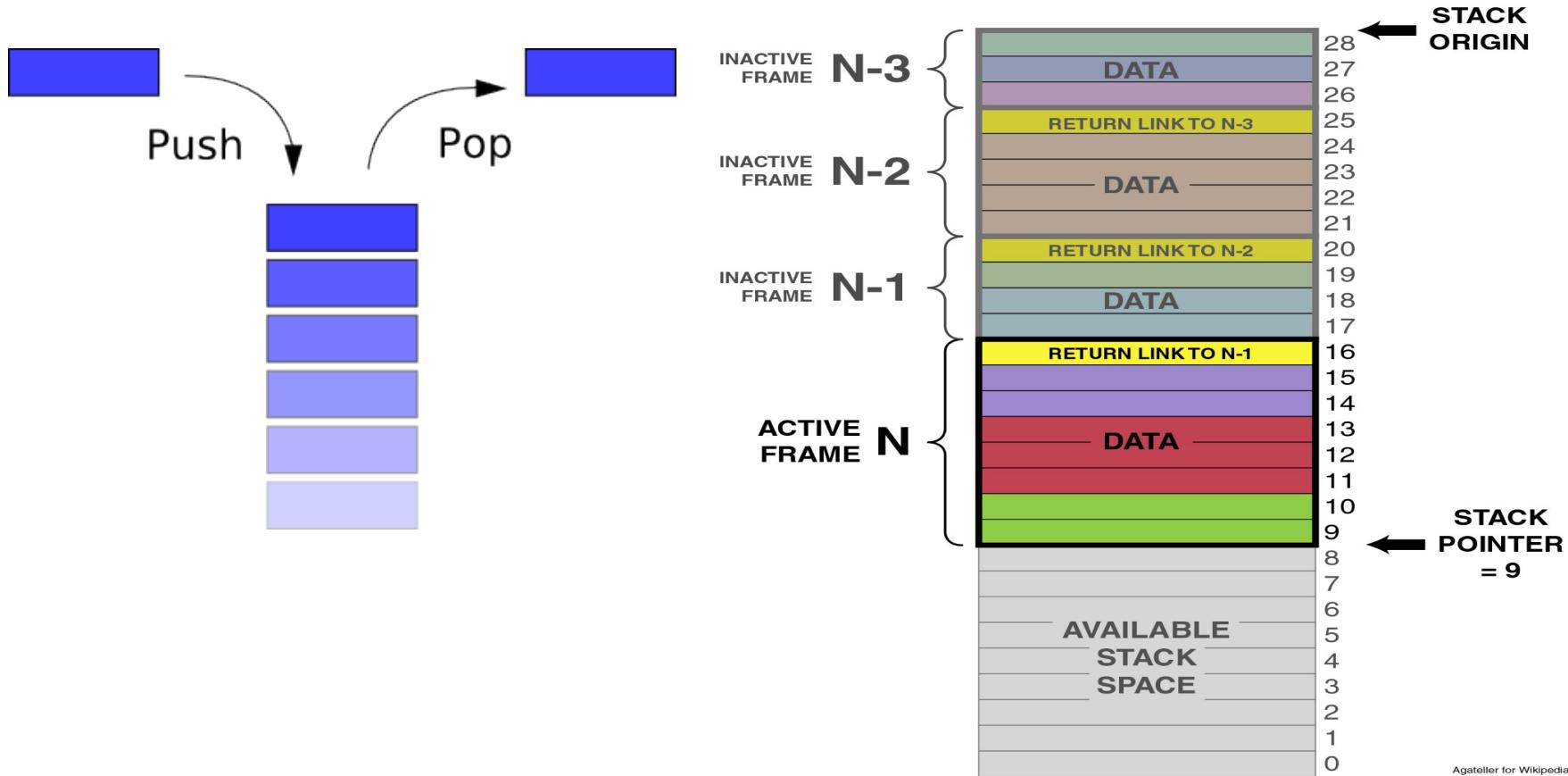
# Objects Initialization. Array initialization

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

# 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 Java 8+ Metaspace
  - Permanent Generation = class definitions.
- **Constant storage, non-RAM storage (external memory)**

# Program Stack



Agateller for Wikipedia  
Public Domain 2006



Source: <http://en.wikipedia.org/wiki/File:ProgramCallStack2.png> 26

C:\Windows\system32\cmd.exe

```
c:\CourseAdvancedJavaVerint\Temp>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=0x00000000024b8000 nid=0x1f04 waiting on condition [0x0000000000000000]
    java.lang.Thread.State: RUNNABLE

"Thread-9" #20 prio=5 os_prio=0 tid=0x000000000bea7000 nid=0x2348 waiting for monitor entry [0x000000000d14f000]
    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=0x000000000bea5800 nid=0x6ac waiting for monitor entry [0x000000000ca2e000]
    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=0x000000000bea5000 nid=0x1ffc waiting for monitor entry [0x000000000cfcf000]
    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=0x000000000bea2000 nid=0x40c waiting for monitor entry [0x000000000cd5f000]
    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=0x000000000bea0800 nid=0x1708 waiting for monitor entry [0x000000000ceae000]
    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)

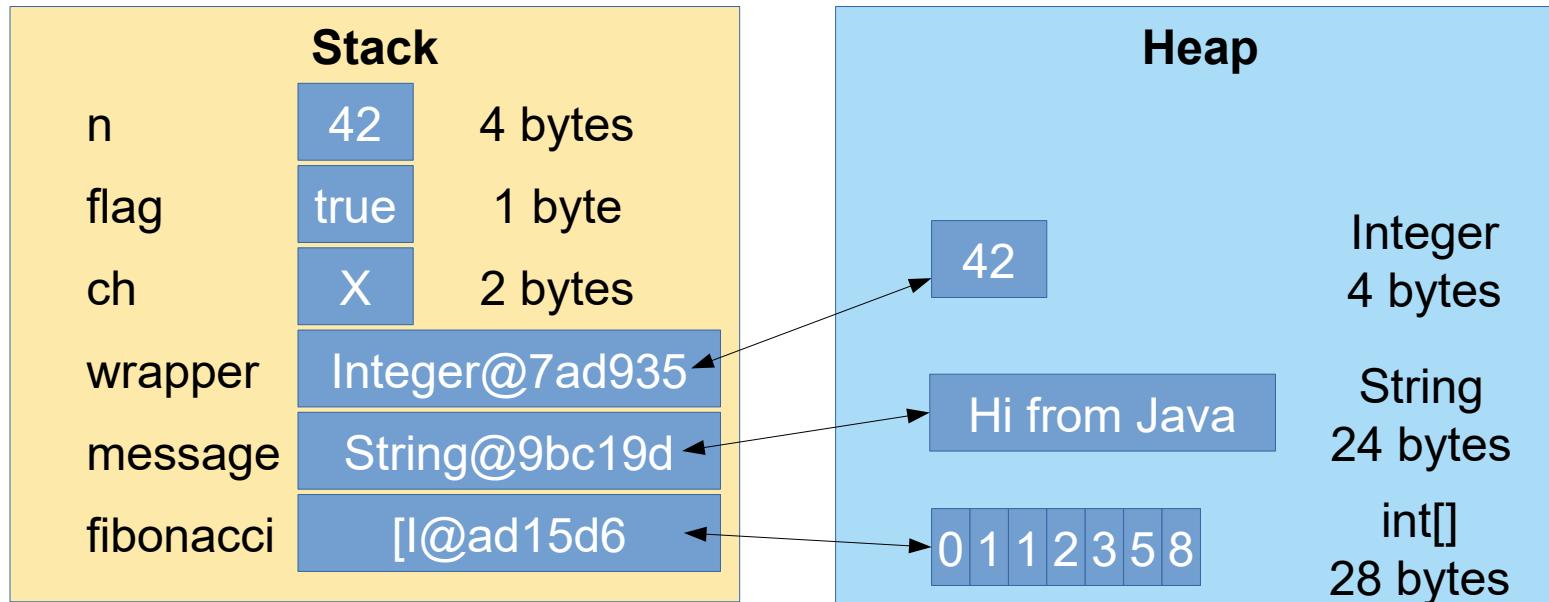
"Thread-4" #15 prio=5 os_prio=0 tid=0x000000000be9d000 nid=0xc0c waiting for monitor entry [0x000000000c7df000]
    java.lang.Thread.State: BLOCKED (on object monitor)
        at simpletest.TwoThreadsSynchronizedCounter.lambda$0(TwoThreadsSynchronizedCounter.java:14)
        - waiting to lock <0x00000000d5e660a0> (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=0x000000000be9c800 nid=0x2394 waiting for monitor entry [0x000000000cc2f000]
    java.lang.Thread.State: BLOCKED (on object monitor)
```



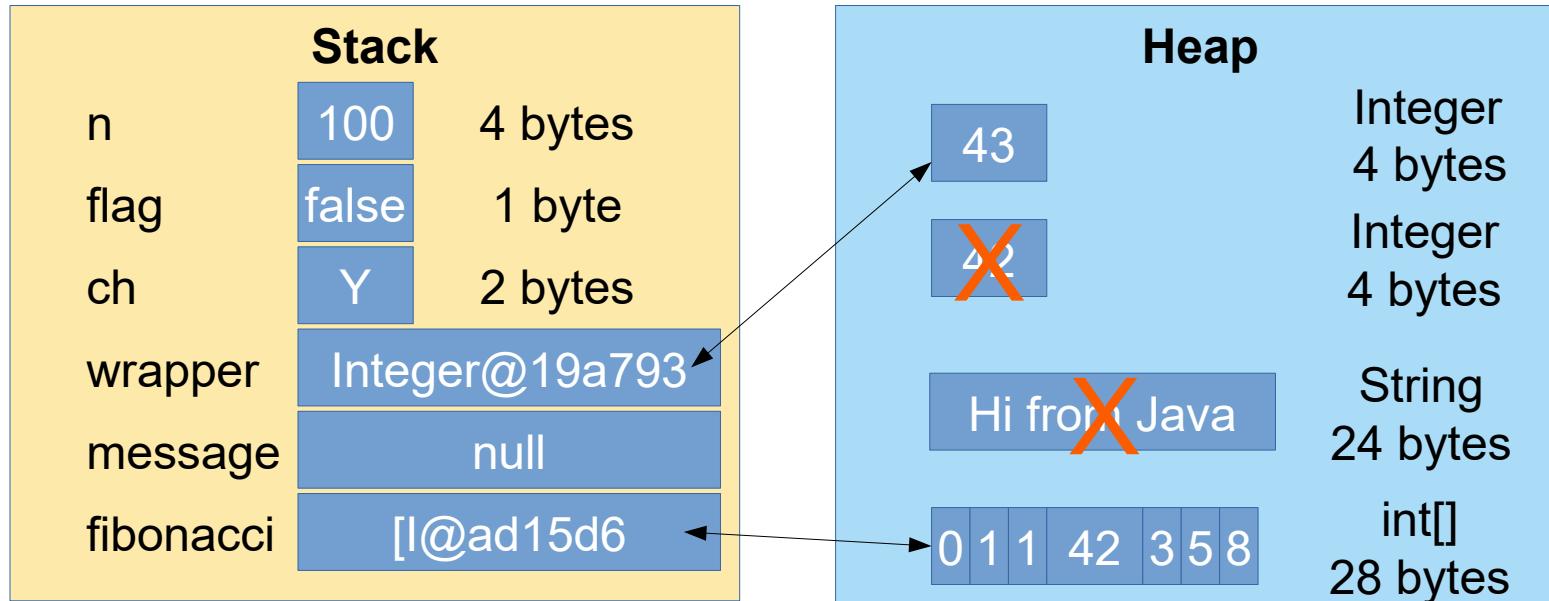
# Stack and Heap (Quick Review)

```
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  
    }  
}
```

# 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 => (x + y) + z**, because the operator **+** is left-associative
- if it was right associative, the result would be  
**+ (y + z)**

x

# 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 or by 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>;
}
```

# Enumeration Types

```
public class MyEnumeration {  
    public enum InvoiceType { SIMPLE, VAT }  
    public static void main(String[] args) {  
        for(InvoiceType it : InvoiceType.values())  
            System.out.println(it);  
    }  
}
```

Результат: SIMPLE  
VAT

# Низове

- Класът **String** предоставя **immutable** обекти – т.е. всяка операция върху низа създава нов обект в хипа
- **StringBuilder** – предоставя ефикасен откъм ресурси начин да модифициране на низове, като реализира **Reusable Design Pattern: Builder** – за постъпково изграждане на низа (основно с метод **append** и **insert**)
- Основни операции в класа **String**. Форматиран изход – метод **format()** и клас **Formatter**. Спецификатори:

**%[argument\_index\$][flags][width][.precision]conversion**

# Конверсия на типа при форматиране

- d – decimal, интегрални типове
- c – character (unicode)
- b - boolean
- s - String
- f – float, double (с десетична точка)
- e - float, double (scientific notation)
- x – шестнайсетична стойност на интегрални типове
- h – шестнайсетичен хеш код

# Регулярни изрази (1)

- Символни класове

- **.** Any character (may or may not match line terminators)
- **\d** A digit: [0-9]
- **\D** A non-digit: [^0-9]
- **\s** A whitespace character: [ \t\n\x0B\f\r]
- **\S** A non-whitespace character: [^\s]
- **\w** A word character: [a-zA-Z\_0-9]
- **\W** A non-word character: [^\w]

# Регулярни изрази (2)

- Квалификатори:
  - **X?** X, once or not at all
  - **X\*** X, zero or more times
  - **X<sup>+</sup>** X, one or more times
  - **X{n}** X, exactly n times
  - **X{n,}** X, at least n times
  - **X{n,m}** X, at least n but not more than m times
- **Greedy, Reluctant (?) & Possessive (+)** квалификатори
- **Capturing Group - (X)**

# Регулярни изрази (3)

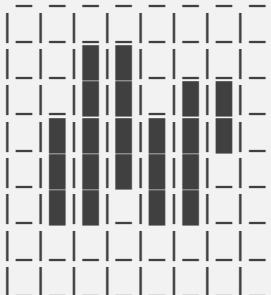
- Клас **Pattern** – основни методи:
  - **public static Pattern compile(String regex)**
  - **public Matcher matcher(CharSequence input)**
  - **public static boolean matches(String regex,**  
**CharSequence input)**
  - **public String[] split(CharSequence input, int limit)**
- Клас **Matcher** – основни методи:
  - **public boolean matches()**
  - **public boolean lookingAt()**
  - **public boolean find(int start)**
  - **public int groupCount()** и **public String group(int group)**

# Problem: Word Counting

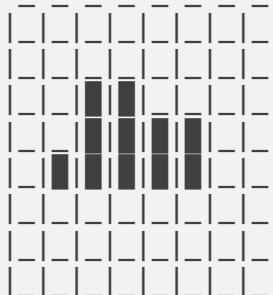
Реализирайте конзолно приложение което по подадено като аргумент от команден ред име на файл извлича top 20 ключови думи за файла на база на честотата на тяхното срещане.

# Problem: Melting Iceberg

Айсберг има форма, която може да се изобрази в таблица с N реда и N стълба,  $7 < N < 200$ , например айсбергът от фиг. 1 след един час в резултат на топенето се превръща в айсберга от фиг. 2:



фиг. 1



фиг. 2

Клетките от първия и последния ред и стълб са винаги празни. Външните клетки, които са изложени на съприкосновение с топлия въздух и вода се топят, а вътрешните не. Айсбергът се топи по следното правило: всяка клетка която има поне 2 от съседните 4 клетки (с обща страна) празни се стопява изцяло за 1 час, а останалите клетки не се топят изобщо. Напишете програма, която прочита от текстов файл размера и съдържанието на таблицата:

```
8
00000000
00**0000
00***0***0
0*****0
0*****0
0**0***0
00000000
00000000
```

В резултат програмата следва да извежда на екрана броя часове, за които айсбергът ще се разтопи изцяло. В горния пример изходът на програмата следва да бъде: 4.

# Problem: Melting Iceberg II

Реализирайте конзолно приложение за интерактивно въвеждане и редактиране на таблицата от задача 3. Приложението следва да поддържа текстово меню с възможности за:

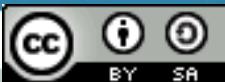
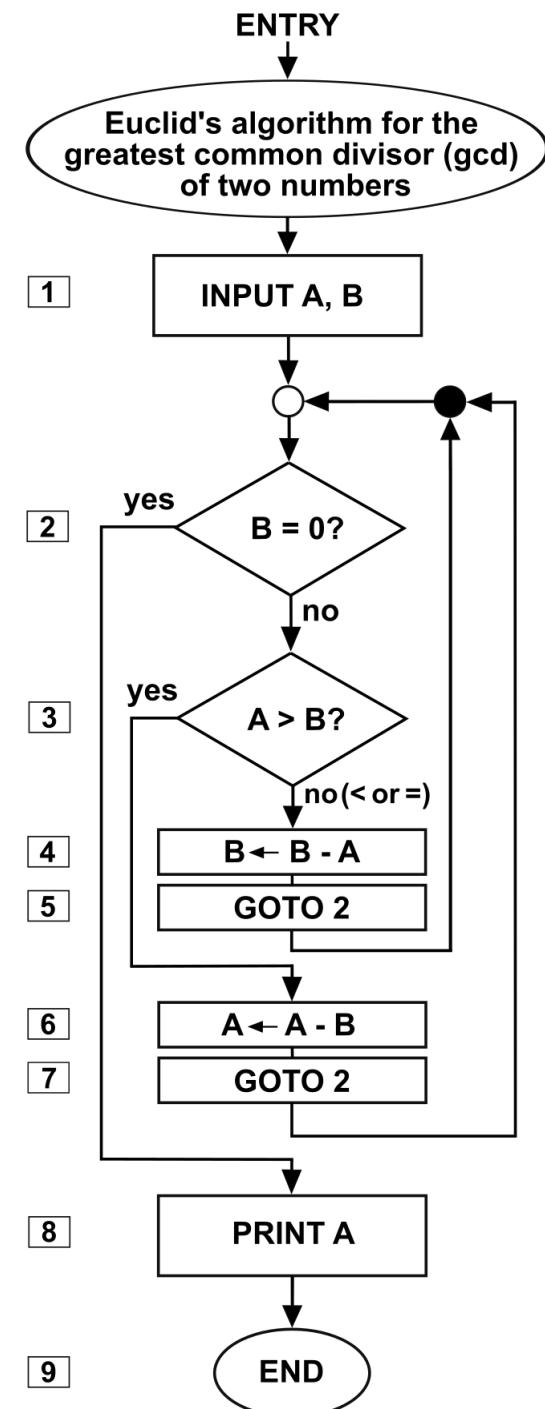
- 1) редактиране на таблицата;
- 2) създаване на нова празна таблица с възможност за редактиране;
- 3) прочитане на таблицата от текстов файл;
- 4) запис на таблицата в текстов файл;
- 5) изход от програмата.

Редактирането на таблицата трябва да стане в текстов вид, интерактивно от клавиатурата с поддържане на активен курсор (символ '#') и с натискане на '+' за запълване на клетката където е курсора и '-' за изчистване на клетката, където е курсора. Преместването на курсора става със стрелките от клавиатурата.

След натискане на всеки клавиш се извежда цялата таблица и един празен ред за разделител. Редактирането приключва с натискане на клавиша `<Enter>`, след което се връщаме в главното меню на програмата, като редактираната таблица се запомня.

# Basic Algorithms

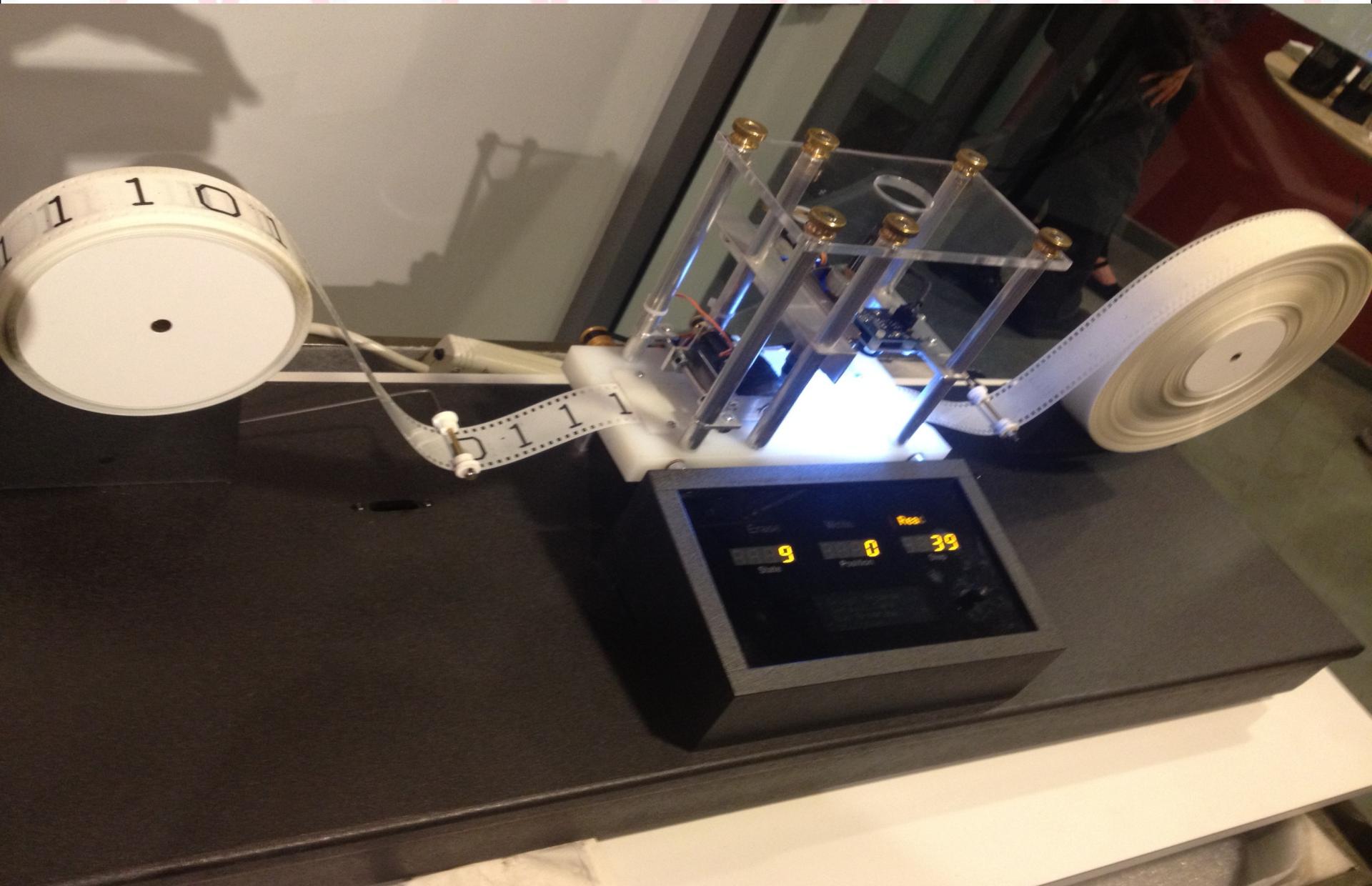
- Algorithm is a finite sequence of well-defined instructions, typically used to solve a class of specific problems or to perform a computation.
- Algorithms are used as specifications for performing calculations and data processing. By making use of artificial intelligence, algorithms can perform automated deductions (referred to as automated reasoning) and use mathematical and logical tests to divert the code through various routes (referred to as automated decision-making).



# Algorithms and Datastructures

- Minsky: "But we will also maintain, with Turing ... that any procedure which could "naturally" be called effective, can, in fact, **be realized by a (simple) machine**. Although this may seem extreme, the arguments ... in its favor are hard to refute".
- Gurevich: "... Turing's informal argument in favor of his thesis justifies a stronger thesis: **every algorithm can be simulated by a Turing machine** ... according to Savage, an **algorithm is a computational process defined by a Turing machine**".
- Typically, when an algorithm is associated with processing information, **data** can be **read from an input source**, **written to an output device** and **stored** for further processing. Stored data are regarded as part of the **internal state of the entity performing the algorithm**. In practice, the state is stored in one or more **data structures**.

# Turing Machine



# Turing Machine

A finite table of instructions that, given the state( $q_i$ ) the machine is currently in and the symbol( $a_j$ ) it is reading on the tape (symbol currently under the head), tells the machine to do the following in sequence:

- Either erase or write a symbol (replacing  $a_j$  with  $a_{j1}$ ).
- Move the head (which is described by  $d_k$  and can have values: 'L' for one step left or 'R' for one step right or 'N' for staying in the same place).
- Assume the same or a new state as prescribed (go to state  $q_{i1}$ ).



# Turing Machine

A finite table of instructions that, given the state( $q_i$ ) the machine is currently in and the symbol( $a_j$ ) it is reading on the tape (symbol currently under the head), tells the machine to do the following in sequence:

- Either erase or write a symbol (replacing  $a_j$  with  $a_{j1}$ ).
- Move the head (which is described by  $d_k$  and can have values: 'L' for one step left or 'R' for one step right or 'N' for staying in the same place).
- Assume the same or a new state as prescribed (go to state  $q_{i1}$ ).

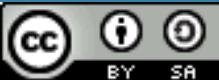


# Algorithmic Schemes: Exhaustive Search

Used when the generation of values to search is cheap:

```
for(i = 0; i < n; i++) {  
    process_value(i);  
}
```

- Permutations
- Combinations



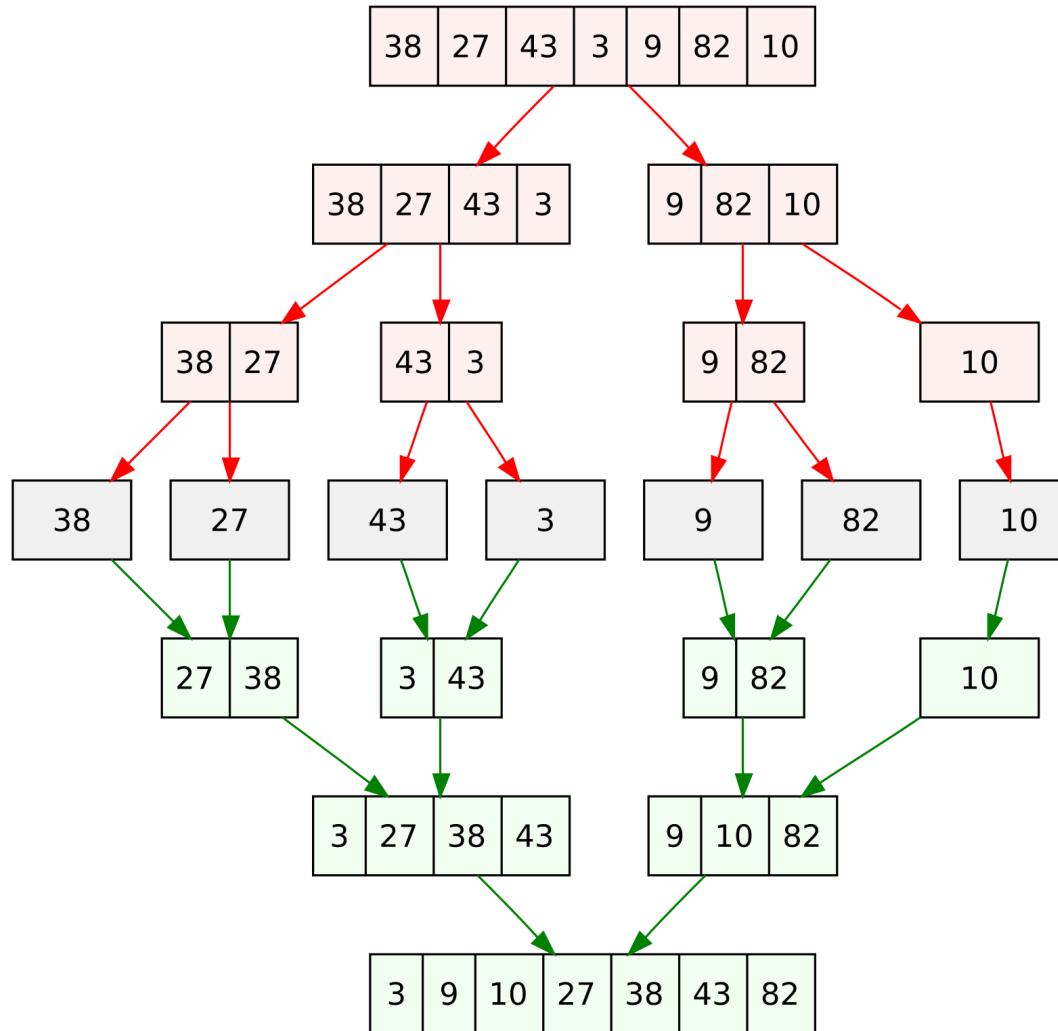
# Merging – Merge Sort

```
function merge_sort(list m) is
    // Base case. A list of zero or one elements is sorted, by definition.
    if length of m ≤ 1 then
        return m
    // Recursive case. First, divide the list into equal-sized sublists
    // consisting of the first half and second half of the list.
    // This assumes lists start at index 0.
    var left := empty list
    var right := empty list
    for each x with index i in m do
        if i < (length of m)/2 then
            add x to left
        else
            add x to right
    // Recursively sort both sublists.
    left := merge_sort(left)
    right := merge_sort(right)
    // Then merge the now-sorted sublists.
    return merge(left, right)
```

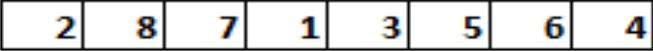
```
function merge(left, right) is
    var result := empty list
    while left is not empty and right is not empty do
        if first(left) ≤ first(right) then
            append first(left) to result
            left := rest(left)
        else
            append first(right) to result
            right := rest(right)
    // Either left or right may have elements left; consume them.
    // (Only one of the following loops will actually be entered.)
    while left is not empty do
        append first(left) to result
        left := rest(left)
    while right is not empty do
        append first(right) to result
        right := rest(right)
    return result
```



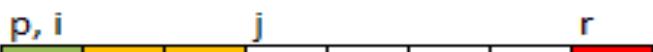
# Divide and Conquer - MergeSort

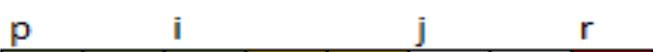


# Divide and Conquer - QuickSort

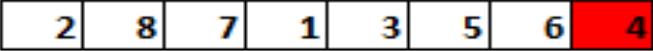
$A[8] =$    
(1)

$p, i, j$    
(3)

$p, i, j$    
(5)

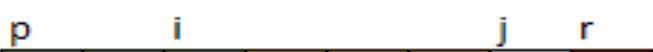
$p, i, j$    
(7)

$p, i, j$    
(9)

$i$   $p, j$    
(2)

$p, i, j$    
(4)

$p, i, j$    
(6)

$p, i, j$    
(8)

$p, i, j$    
(10)

	- pivot element
	- to be sorted
	<= pivot element
	> pivot element

# Dynamic Programming

- Dynamic Programming is a technique in computer programming that helps to efficiently solve a class of problems that have **overlapping subproblems** and **optimal substructure property**.
- If the problem can be divided into subproblems recursively, and if there are **overlapping subproblems**, then the solutions to these subproblems can be saved for future reference (memoising - such problems involve repeatedly calculating the value of the same subproblems to find the solution).
- **optimal substructure property** - if an **optimal solution** can be **constructed from optimal solutions** of its subproblems, used to determine the usefulness of **dynamic programming** and **greedy algorithms** for a problem.
- Example : **Fibonacci sequence**

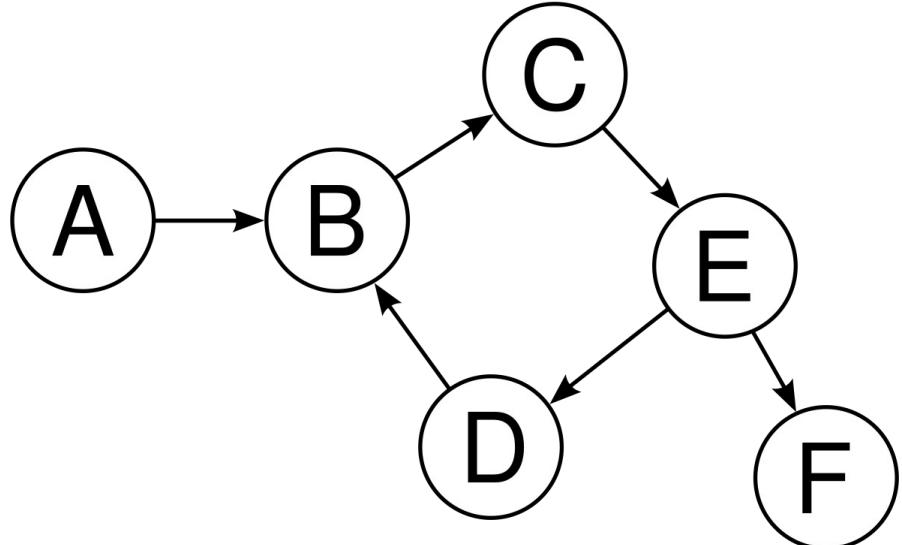
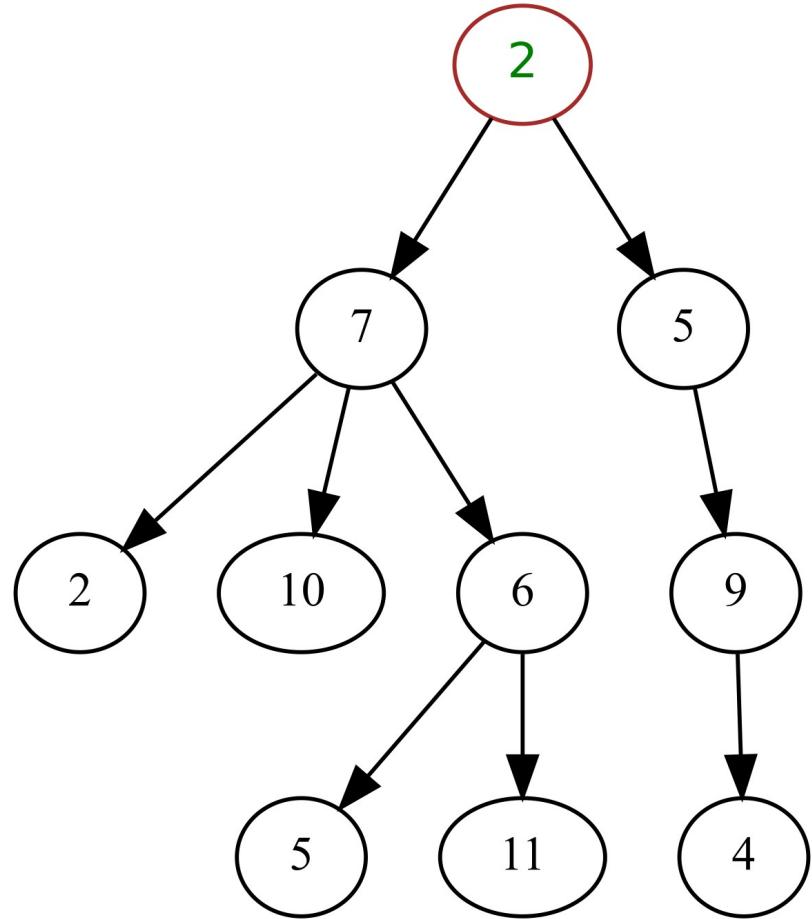


# Greedy Algorithms

- A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. Does not always find optimal result.
- The algorithm does not return to previous candidate solutions already processed, even if the choice is wrong. It works in a top-down approach.
- It always goes for the locally optimal best choice to produce the global best result.
- GA to be applicable the problem should posses the following properties:
  1. Greedy choice - the optimal solution to the problem can be found by choosing the best choice at each step without returning to the previous steps.
  2. Optimal substructure - If the optimal overall solution to the problem corresponds to the optimal solution to its subproblems.
- Example: Dijkstra's minimal spanning tree algorithm



# Trees and Graphs



# Graph Algorithms

- Depth First Search (DFS)
- Breadth First Search (BFS)
- Backtracking - a general algorithm for finding solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.
- The classic textbook example of the use of backtracking is the [eight queens puzzle](#), that asks for all arrangements of eight chess queens on a standard chessboard so that no queen attacks any other. In the common backtracking approach, the partial candidates are arrangements of  $k$  queens in the first  $k$  rows of the board, all in different rows and columns. Any partial solution that contains two mutually attacking queens can be abandoned.

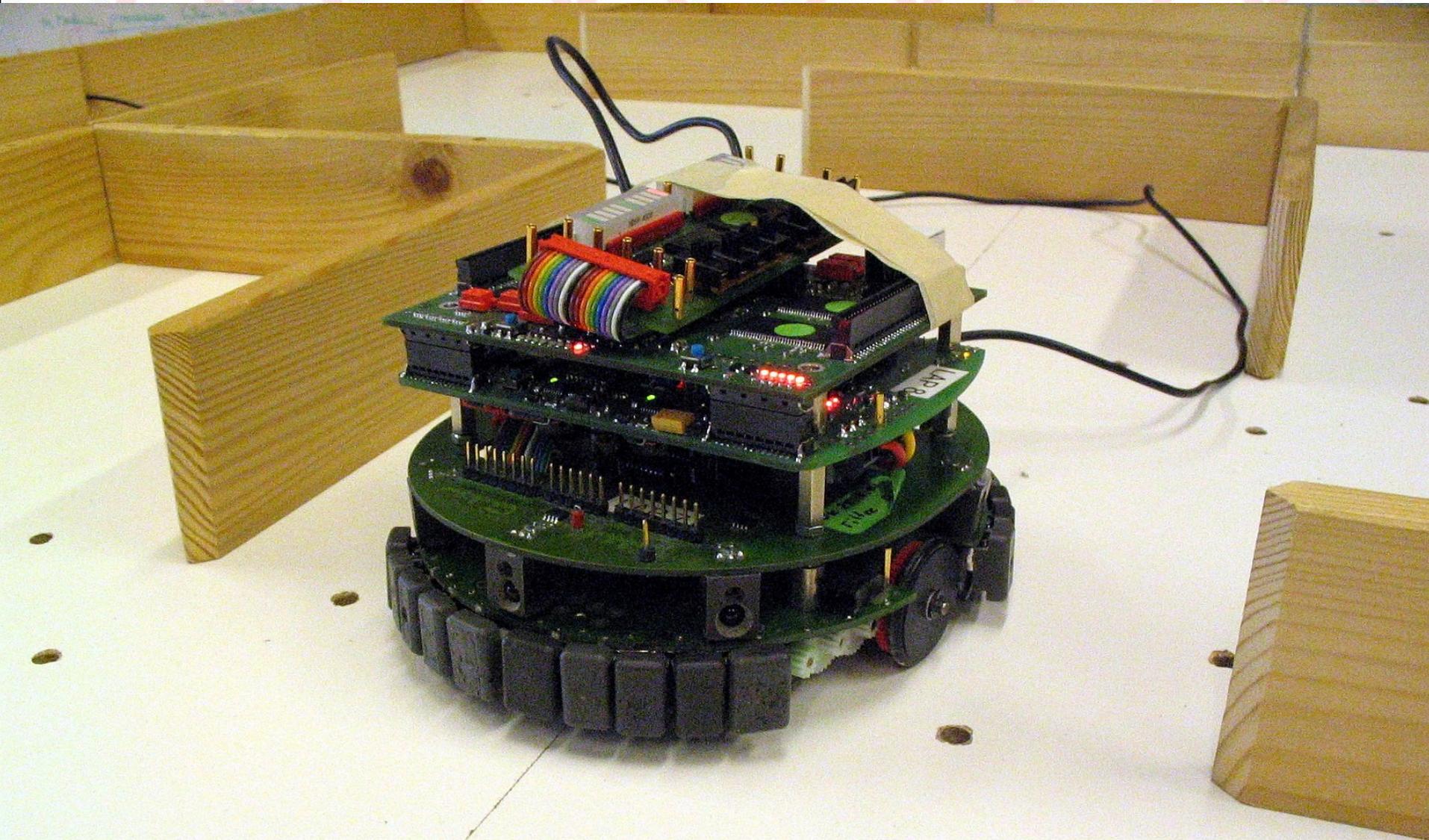


# Backtracking Algorithm

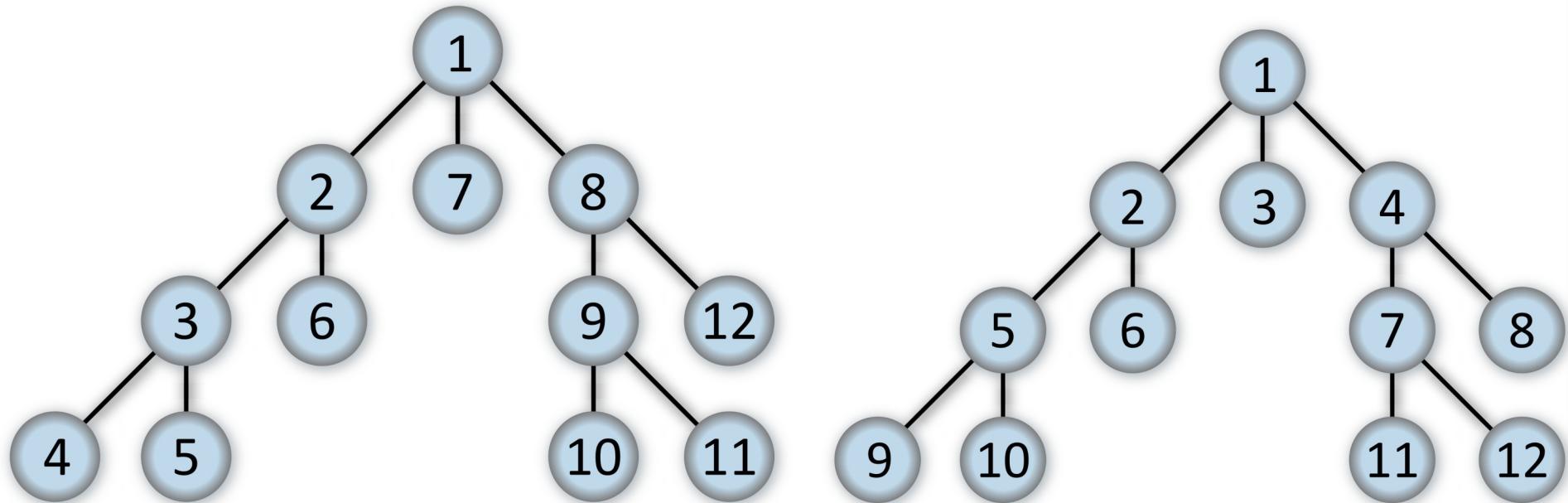
```
procedure backtrack(c) is
    if reject(P, c) then return
    if accept(P, c) then output(P, c)
    s ← first(P, c)
    while s ≠ NULL do
        backtrack(s)
        s ← next(P, s)
```



# Backtracking Algorithm – N Queens, Maze



# Depth First and Breadth First Search



# Problem: Finding Path in a Labyrinth

Да се напиша програма на езика Java, която по подаден като аргумент от команден ред аргумент – име на файл, прочита информация за топологията на лабиринт от файла (таблица с N реда и M стълба,  $5 < N, M < 200$ ) – напр. Фиг. 1 (1 – свободна клетка, 2 – стена на лабиринта, празните редове във файла се прескачат и не се обработват):

```
1,2,1,1,1,2  
1,2,2,2,1,2  
1,1,1,2,1,1  
1,2,1,2,1,1  
1,2,1,1,1,2
```

фиг. 1

Напишете програма която:

1. Въвежда начална и крайна клетка от клавиатурата и намира всички пътища между двете клетки използвайки DFS.
2. Въвежда начална и крайна клетка от клавиатурата и намира всички пътища между двете клетки използвайки BFS (намира пътищата с нарастваща дължина).

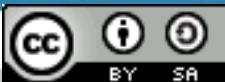
# Graph Operations

The basic operations provided by a graph data structure G usually include:

- `adjacent(G, x, y)`: tests whether there is an edge from the vertex x to the vertex y;
- `neighbors(G, x)`: lists all vertices y such that there is an edge from the vertex x to the vertex y;
- `add_vertex(G, x)`: adds the vertex x, if it is not there;
- `remove_vertex(G, x)`: removes the vertex x, if it is there;
- `add_edge(G, x, y)`: adds the edge from the vertex x to the vertex y, if it is not there;
- `remove_edge(G, x, y)`: removes the edge from the vertex x to the vertex y, if it is there;
- `get_vertex_value(G, x)`: returns the value associated with the vertex x;
- `set_vertex_value(G, x, v)`: sets the value associated with the vertex x to v.

Structures that associate values to the edges usually also provide:[1]

- `get_edge_value(G, x, y)`: returns the value associated with the edge (x, y);
- `set_edge_value(G, x, y, v)`: sets the value associated with the edge (x, y) to v.



# Graph Representations

- **Adjacency list** - vertices are stored as records or objects, and every vertex stores a list of adjacent vertices. This data structure allows the storage of additional data on the vertices. Additional data can be stored if edges are also stored as objects, in which case each vertex stores its incident edges and each edge stores its incident vertices.
- **Adjacency matrix** – a two-dimensional matrix, in which the rows represent source vertices and columns represent destination vertices. Data on edges and vertices must be stored externally. Only the cost for one edge can be stored between each pair of vertices.
- **Incidence matrix** – a two-dimensional matrix, in which the rows represent the vertices and columns represent the edges. The entries indicate the incidence relation between the vertex at a row and edge at a column.

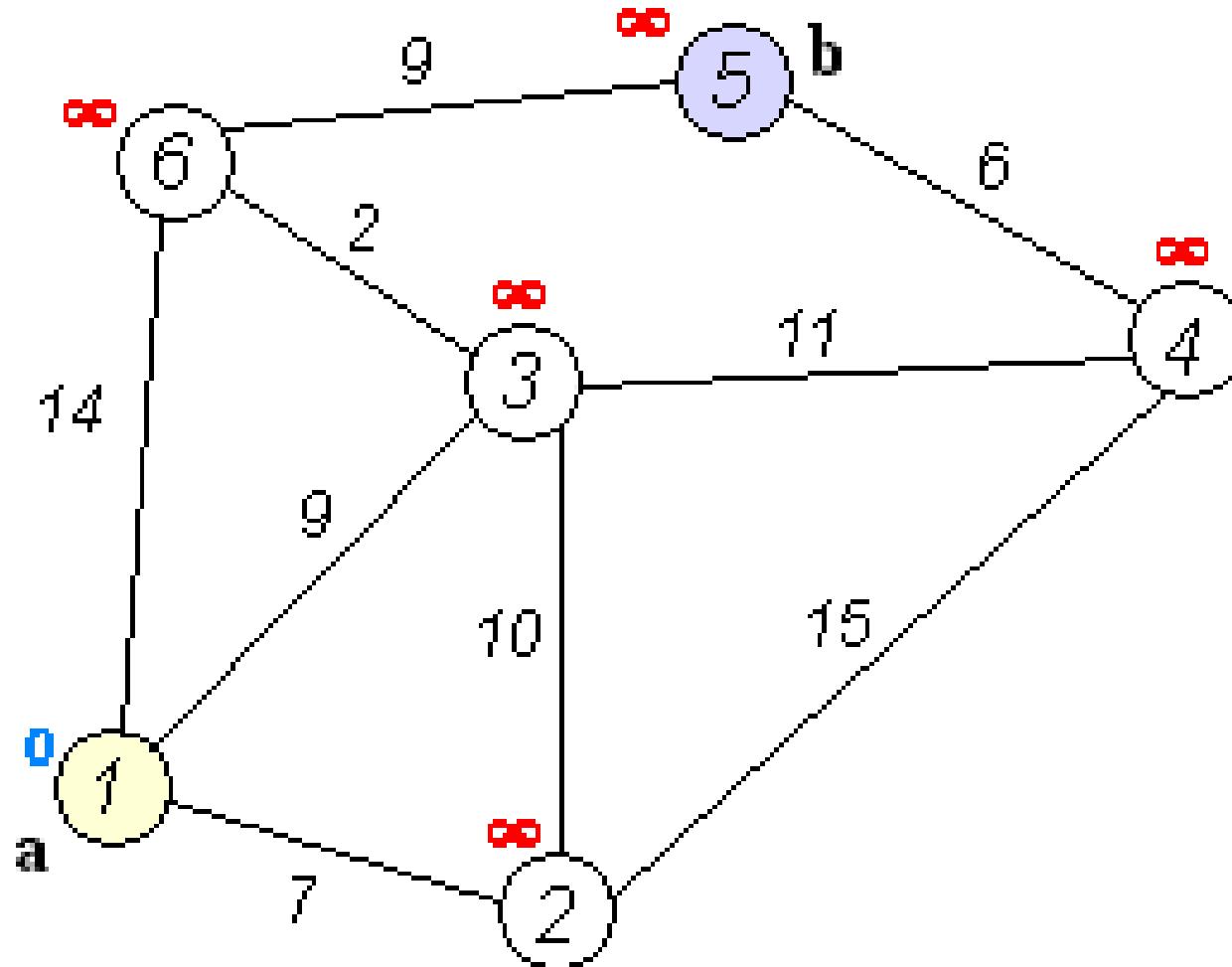


# Graph Representations

	Adjacency list	Adjacency matrix	Incidence matrix
Store graph	$O( V  +  E )$	$O( V ^2)$	$O( V  \cdot  E )$
Add vertex	$O(1)$	$O( V ^2)$	$O( V  \cdot  E )$
Add edge	$O(1)$	$O(1)$	$O( V  \cdot  E )$
Remove vertex	$O( E )$	$O( V ^2)$	$O( V  \cdot  E )$
Remove edge	$O( V )$	$O(1)$	$O( V  \cdot  E )$
Are vertices $x$ and $y$ adjacent (assuming that their storage positions are known)?	$O( V )$	$O(1)$	$O( E )$
Remarks	Slow to remove vertices and edges, because it needs to find all vertices or edges	Slow to add or remove vertices, because matrix must be resized/copied	Slow to add or remove vertices and edges, because matrix must be resized/copied



# Dijkstra's algorithm



# Dijkstra's algorithm - Pseudocode

```
1 function Dijkstra(Graph, source):
2
3     for each vertex v in Graph.Vertices:
4         dist[v] ← INFINITY
5         prev[v] ← UNDEFINED
6         add v to Q
7         dist[source] ← 0
8
9     while Q is not empty:
10        u ← vertex in Q with min dist[u]
11        remove u from Q
12
13        for each neighbor v of u still in Q:
14            alt ← dist[u] + Graph.Edges(u, v)
15            if alt < dist[v]:
16                dist[v] ← alt
17                prev[v] ← u
18
19    return dist[], prev[]
```



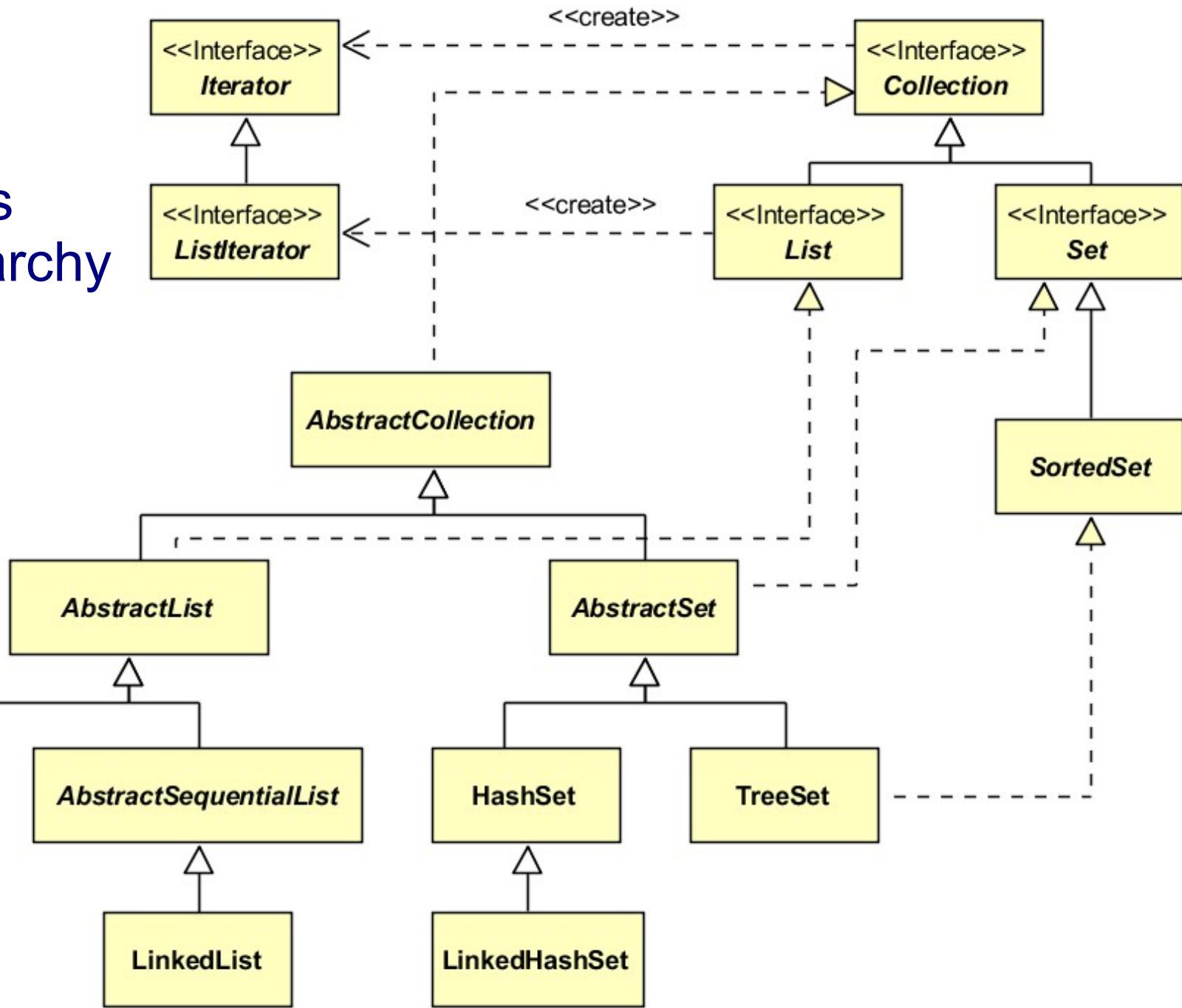
# Arrays. Comapring and Sorting

- Arrays and working with them
- Utility methods of the class **Arrays**:
  - equals()
  - fill()
  - copyOf() и copyOfRange()
  - binarySearch()
  - sort()
- Comparing objects – interfaces **Comparable** and **Comparator**

# Container Classes and Interfaces. Iterators.

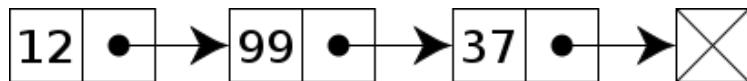
- ❖ Колекции – интерфейс **Collection**
- ❖ Списъци – интерфейс **List**, реализации – **ArrayList**, **LinkedList**, ...
- ❖ Множества – интерфейс **Set**, реализации – **HashSet**, **TreeSet**, ...
- ❖ Асоциативни списъци – интерфейс **Map**, реализации – **HashMap**, **TreeMap**, **LinkedHashMap**, **WeakHashMap**, ...
- ❖ Обхождане на колекция с итератор.
- ❖ Реализиране на структури от данни стек, опашка, дек – интерфейси **Queue** и **Dequeue**. Реализации: **ArrayDeque**

# Class Hierarchy

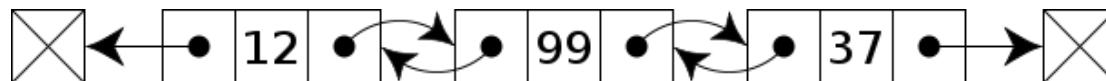


# Data Structures

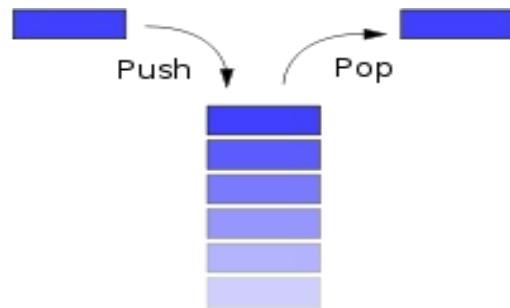
- Linked list:



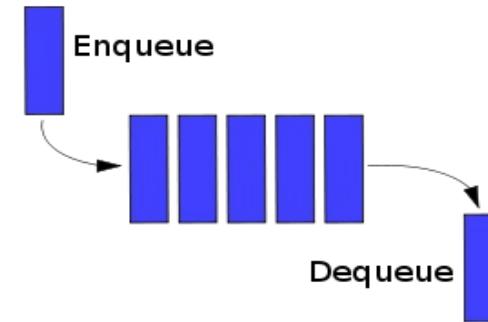
- Doubly-linked list:



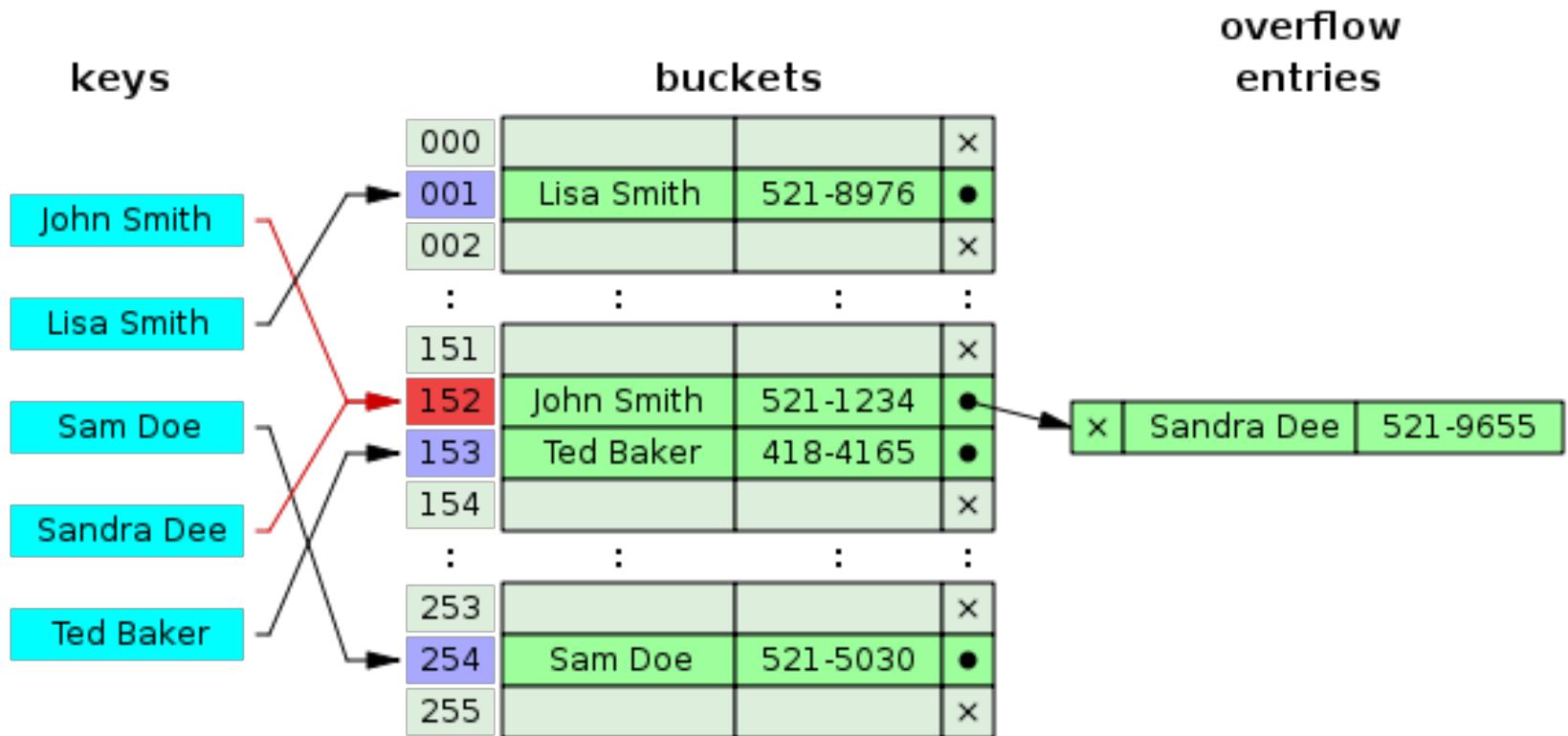
- Stack:



- Queue:



# Hashinng. Hash-Functions. Hash Tables



# Garbage Collection – Main Concepts

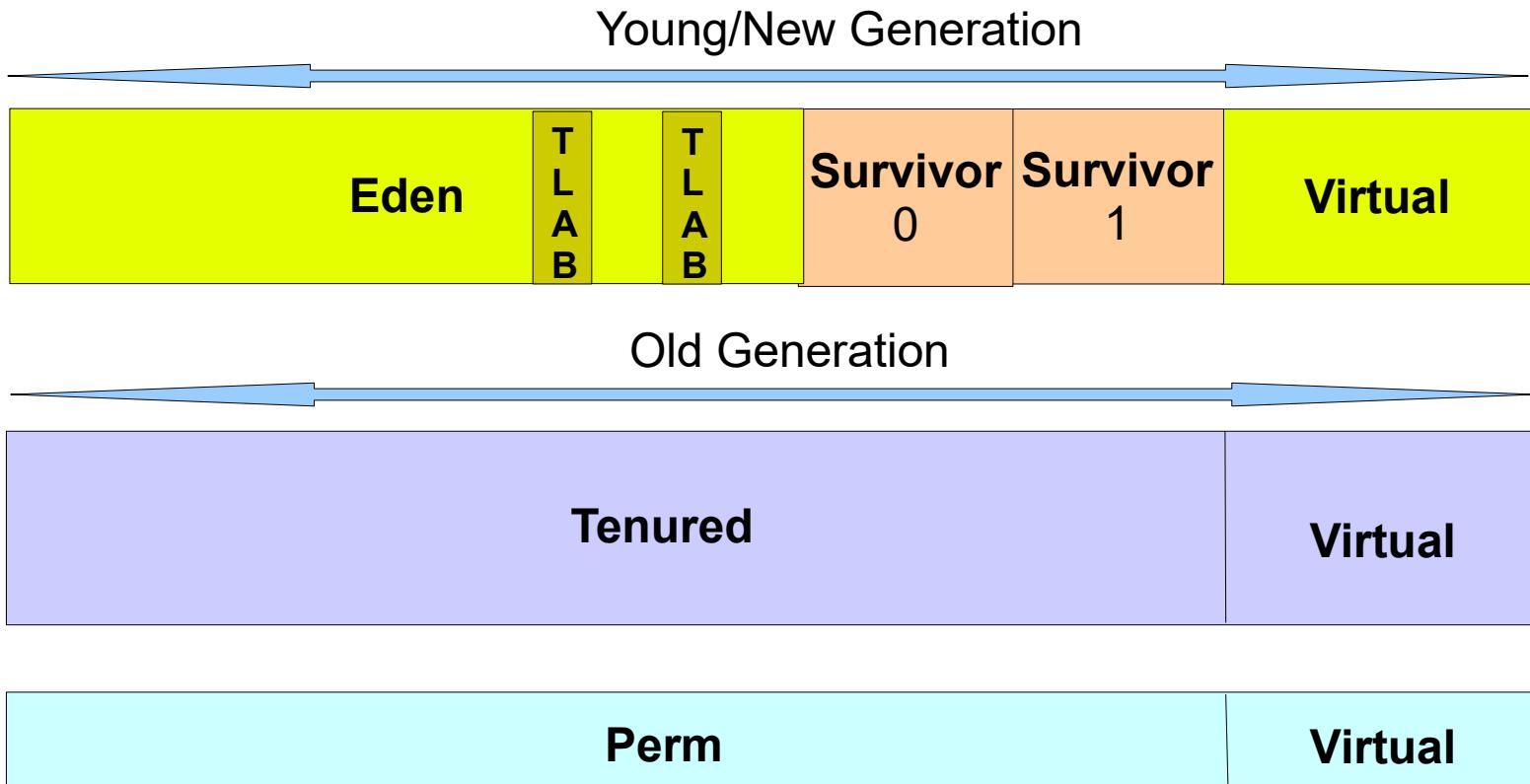
- Garbage collection and finalization – method **finalize()**
- Client and Server VMs ( $\neq$  JIT Compilers & Defaults), x86, x64
- Generational Garbage Collection – **Young, Old & Permanent**  
(in Java 8 → **Metaspace**) – Weak generational hypothesis:
  - Most of the objects become unreachable soon;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

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

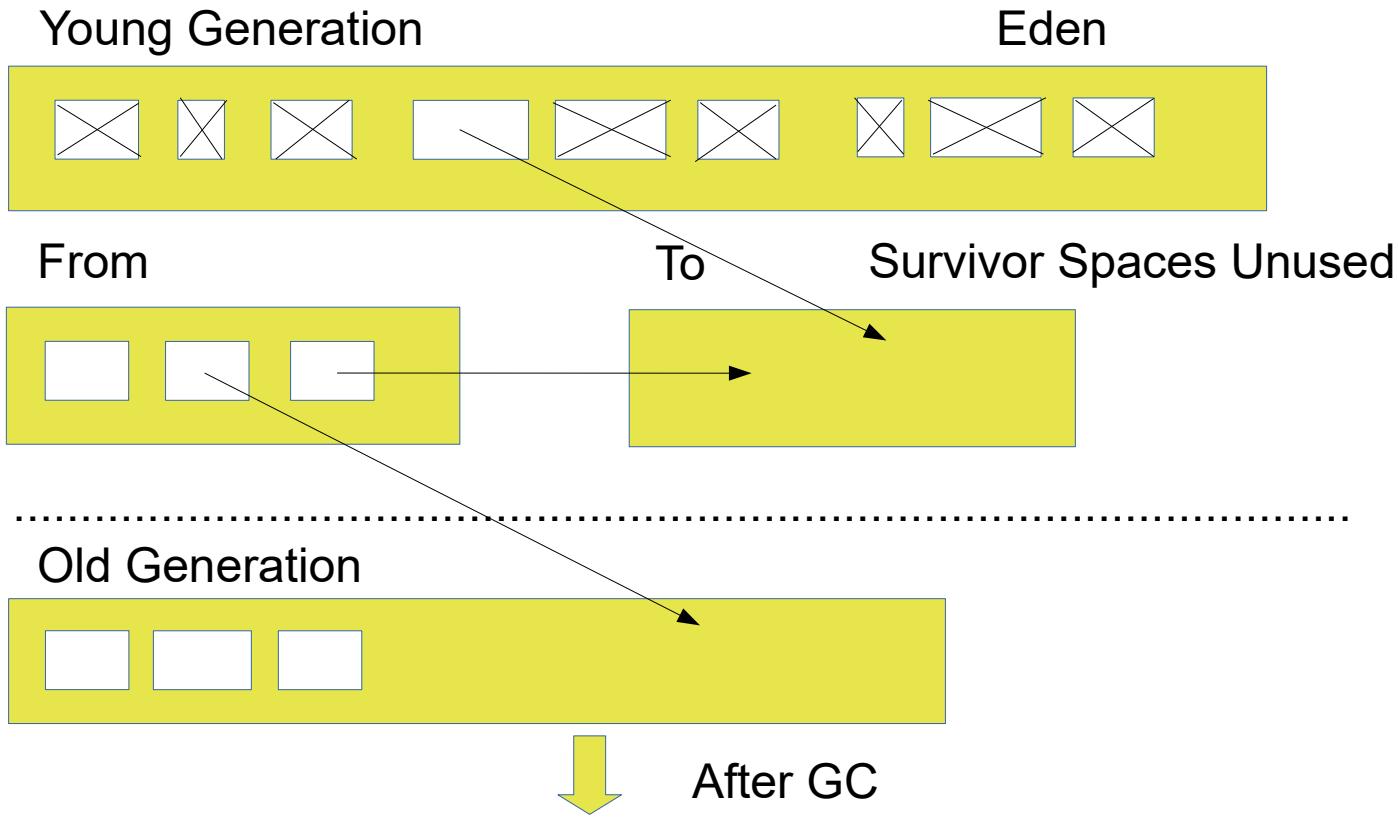
- Tuning for **Lower Latency**

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

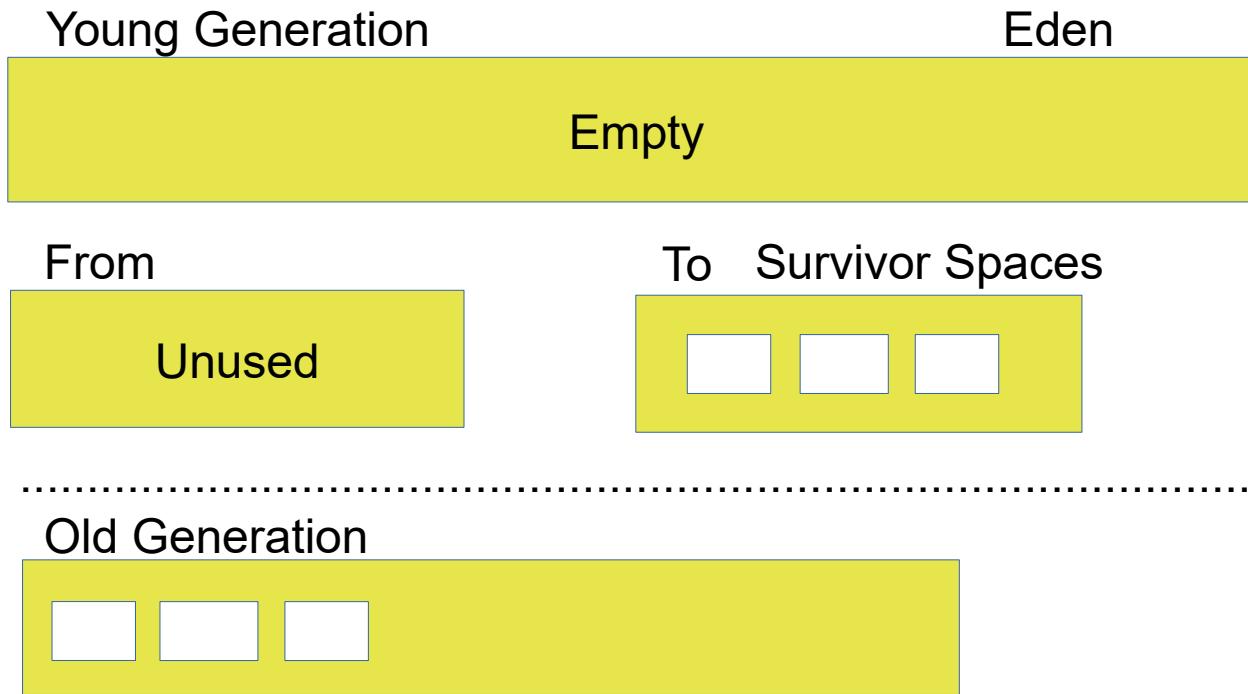
# Garbage Collection – Main Concepts



# Before GC



# After GC



# 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** – threshold and ages of New generation
- XX:+PrintGCDetails**
- XX:+PrintGCTimeStamps**

# 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**

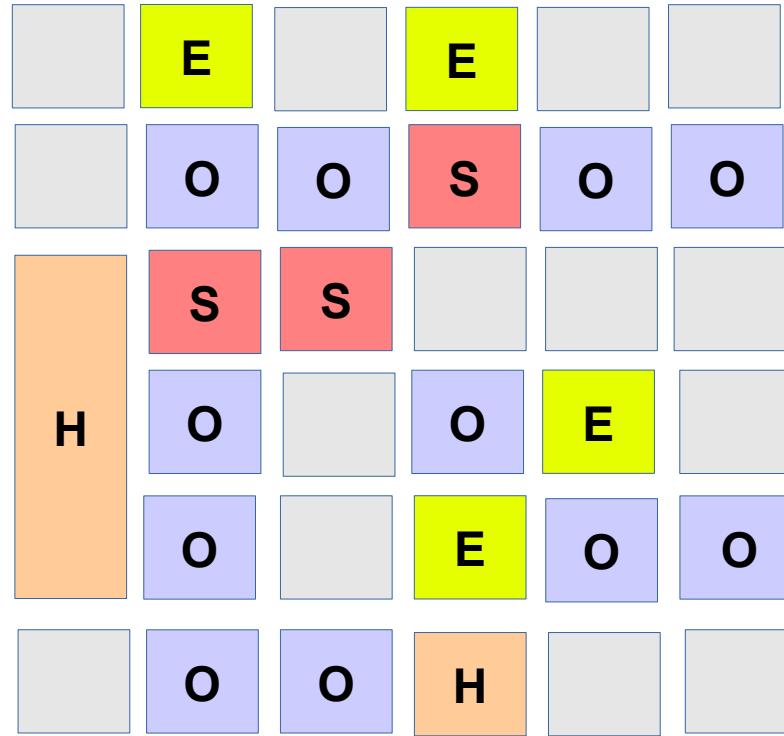
**-XX:+CMSParallelRemarkEnabled**

**-XX:CMSInitiatingOccupancyFraction=<N>**

**-XX:+UseCMSInitiatingOccupancyOnly**

G1 **-XX:+UseG1GC**

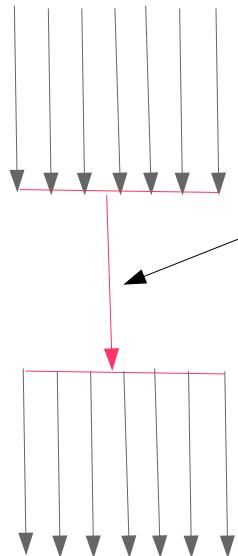
# Garbage First G1 Partially Concurrent Collector



<b>E</b>	Eden
<b>S</b>	Survivor
<b>O</b>	Old
<b>H</b>	Humongous
	Unused

# CMS GC (-XX: +UseConcMarkSweepGC)

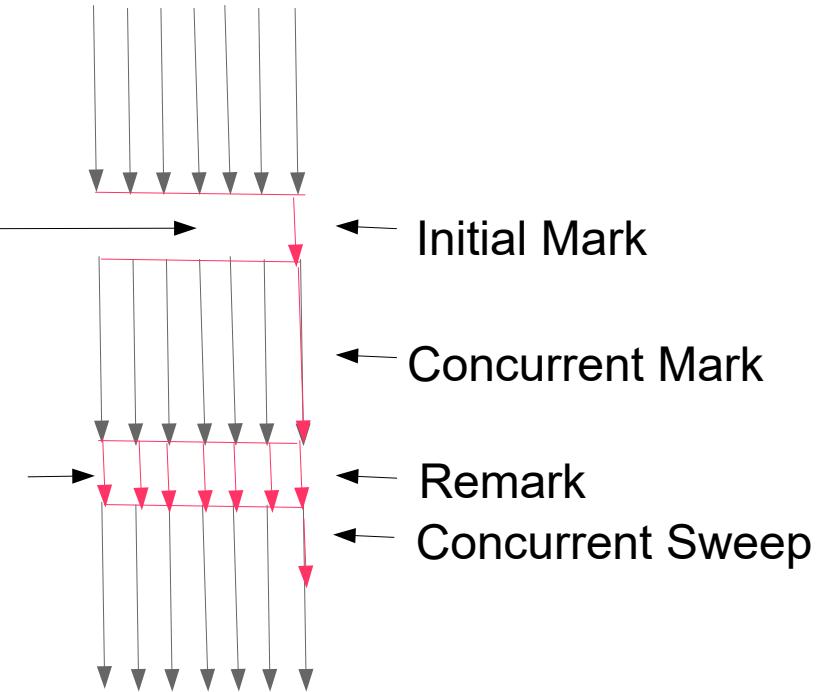
Serial Mark-Sweep-Compact  
Collector



Stop-the-world pause

Stop-the-world  
pause

Concurrent Mark-Sweep  
Collector



# 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 (**Ivmid** - 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**

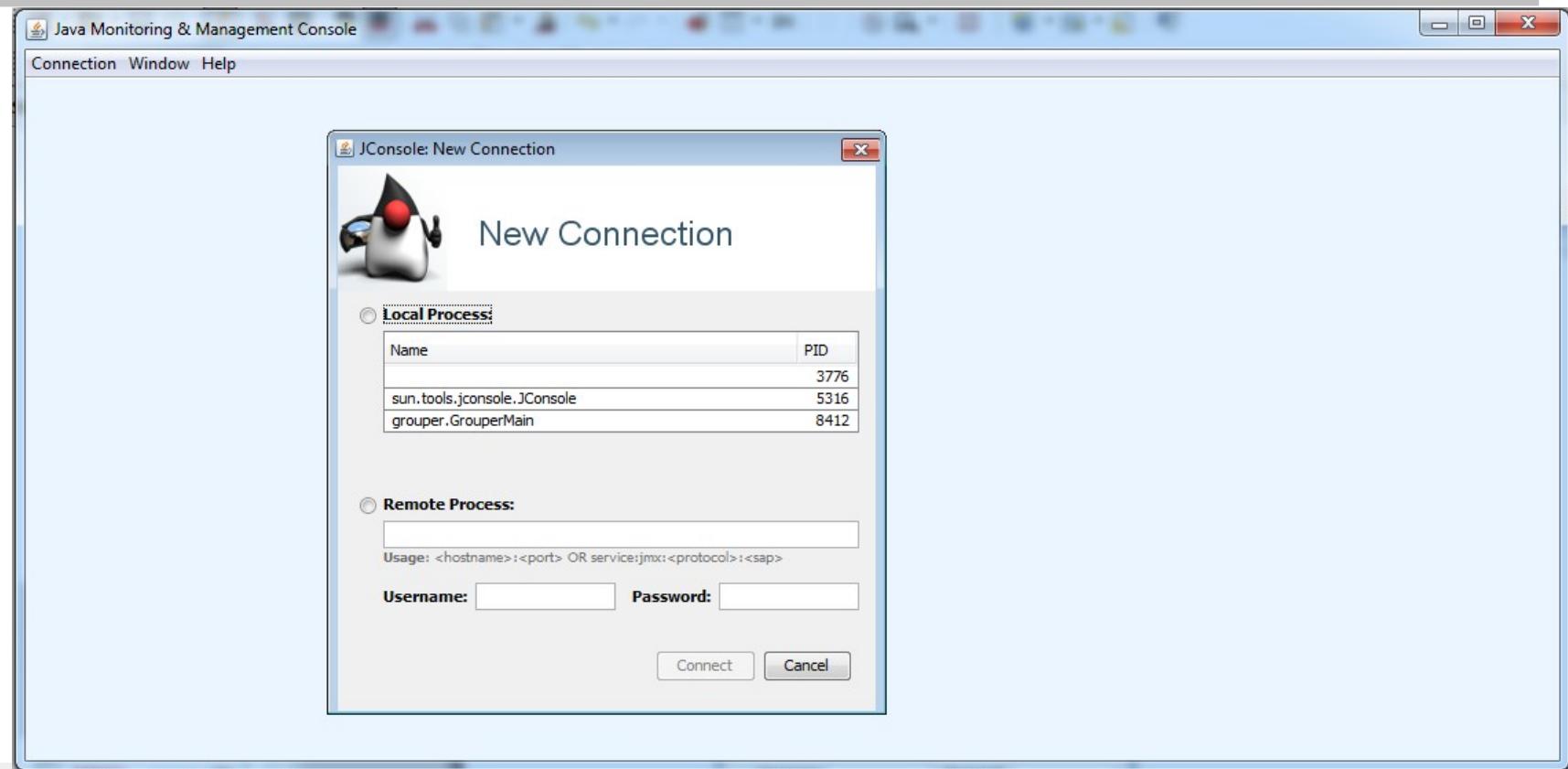
# Java Command Line Monitoring/Tuning Tools -II

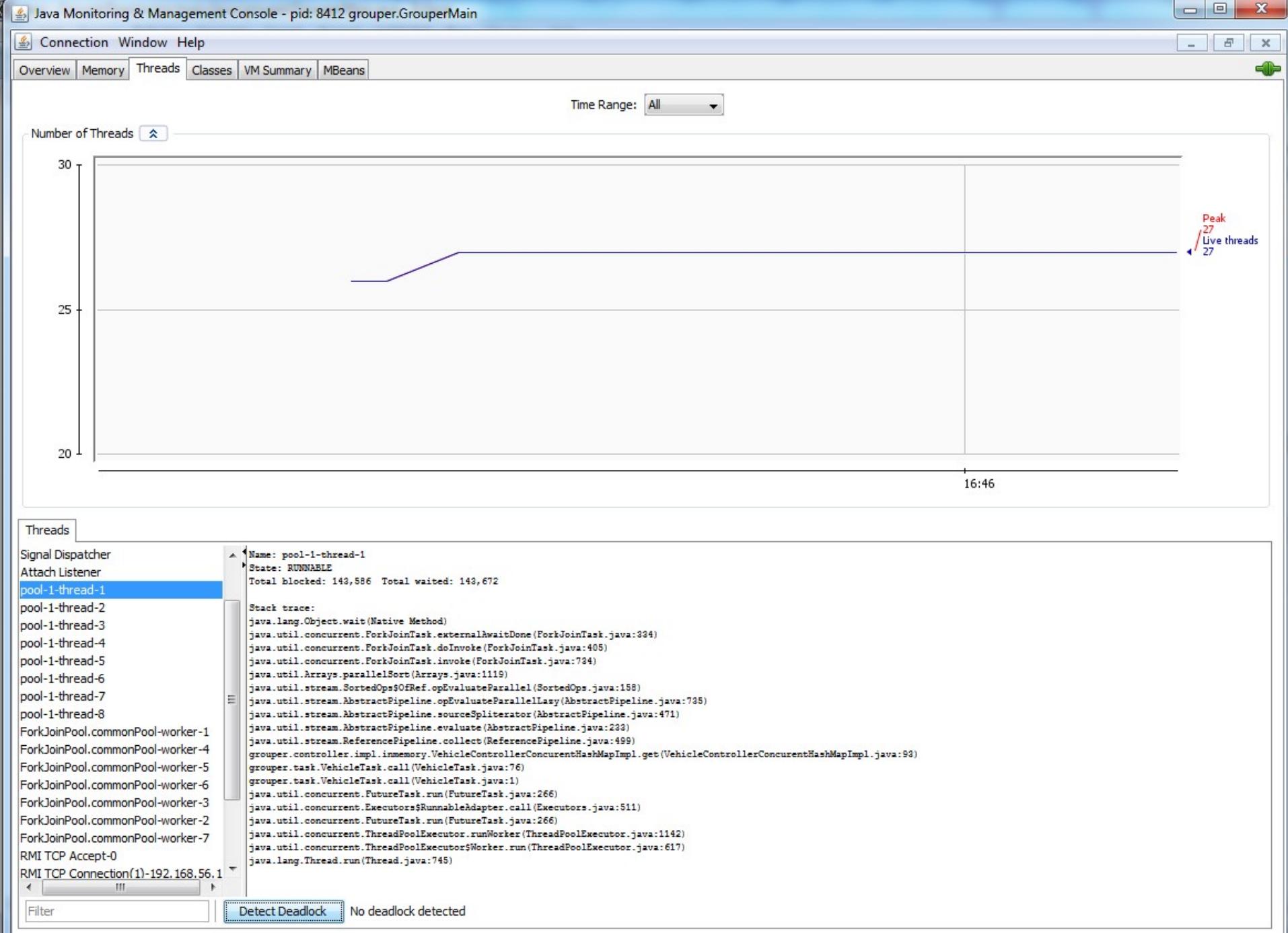
- **jstat & jstatd** – provide information about GC and class loading activities, useful for automated scripting (**jstatd** = RMI deamon):

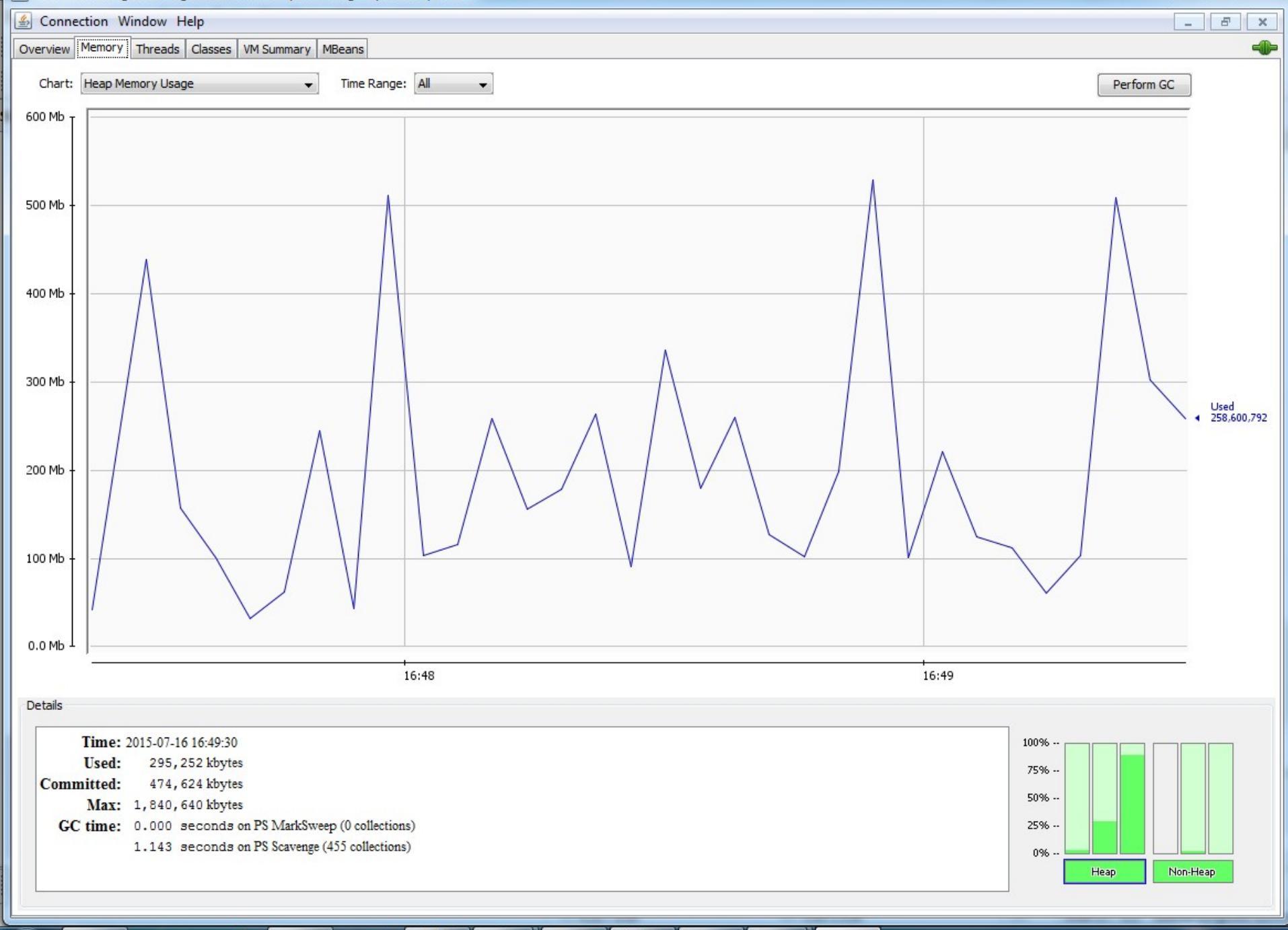
jstat [ generalOption | outputOptions vmid [interval[s|ms] [count]]] Ex: **jstat -gc -t -h20 4572 2s**

- 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;
  - gcnew, -gcnewcapacity, -gcold, -gcoldcapacity, -gcpermcapacity**
    - Young/Old/Permanent generation stats
  - printcompilation** - HotSpot compilation method statistics

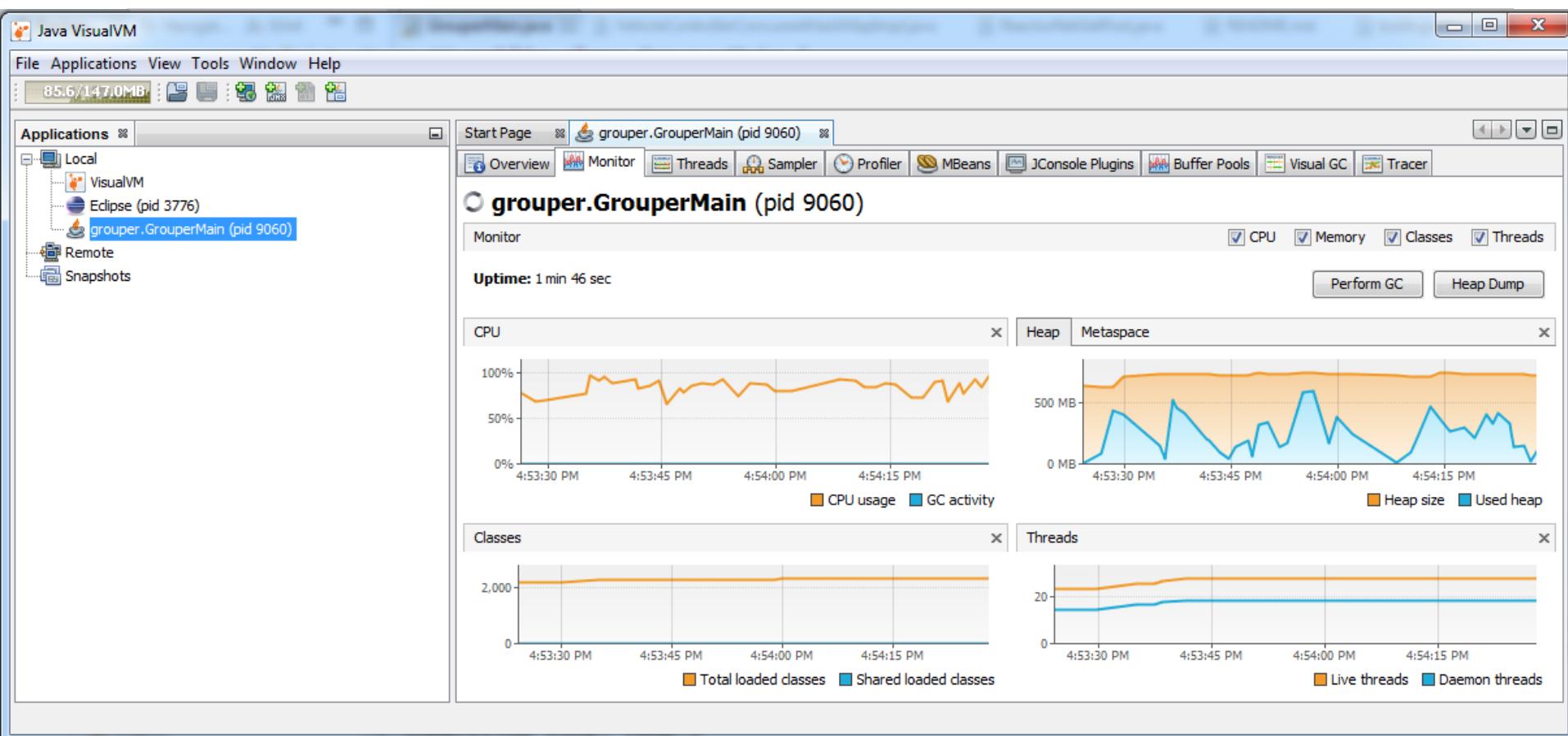
## Java GUI tools – JConsole







## Java GUI tools – jvisualvm



Java VisualVM

File Applications View Tools Window Help

81.0 / 117.0MB

Applications Local Remote Snapshots

Start Page grouper.GrouperMain (pid 8332)

Overview Monitor Threads Sampler Profiler MBeans JConsole Plugins Buffer Pools Visual GC Tracer

### grouper.GrouperMain (pid 8332)

Threads

Live threads: 27 Daemon threads: 18

Threads visualization Threads inspector

Thread Dump

#### Timeline

View: All threads

Name	5:23:00 PM	5:23:05 PM	5:23:10 PM	5:23:15 PM	Running	Total
RMI TCP Connection(1)-192.168.5					112,250 ms (65.2%)	172,238 ms
RMI TCP Accept-0					297,342 ms (100%)	297,342 ms
ForkJoinPool.commonPool-worker-1					265,586 ms (89.3%)	297,342 ms
ForkJoinPool.commonPool-worker-2					278,134 ms (93.5%)	297,342 ms
ForkJoinPool.commonPool-worker-3					263,854 ms (88.7%)	297,342 ms
ForkJoinPool.commonPool-worker-4					277,000 ms (93.2%)	297,342 ms
ForkJoinPool.commonPool-worker-5					269,869 ms (90.8%)	297,342 ms
ForkJoinPool.commonPool-worker-6					273,238 ms (91.9%)	297,342 ms
ForkJoinPool.commonPool-worker-7					268,046 ms (90.1%)	297,342 ms
pool-1-thread-8					138,788 ms (46.7%)	297,342 ms
pool-1-thread-7					165,915 ms (55.8%)	297,342 ms
pool-1-thread-6					156,789 ms (52.7%)	297,342 ms
pool-1-thread-5					182,833 ms (61.5%)	297,342 ms
pool-1-thread-4					143,730 ms (48.3%)	297,342 ms
pool-1-thread-3					157,020 ms (52.8%)	297,342 ms
pool-1-thread-2					160,665 ms (54%)	297,342 ms
pool-1-thread-1					175,579 ms (59%)	297,342 ms

Running Sleeping Wait Park Monitor

#### Threads inspector

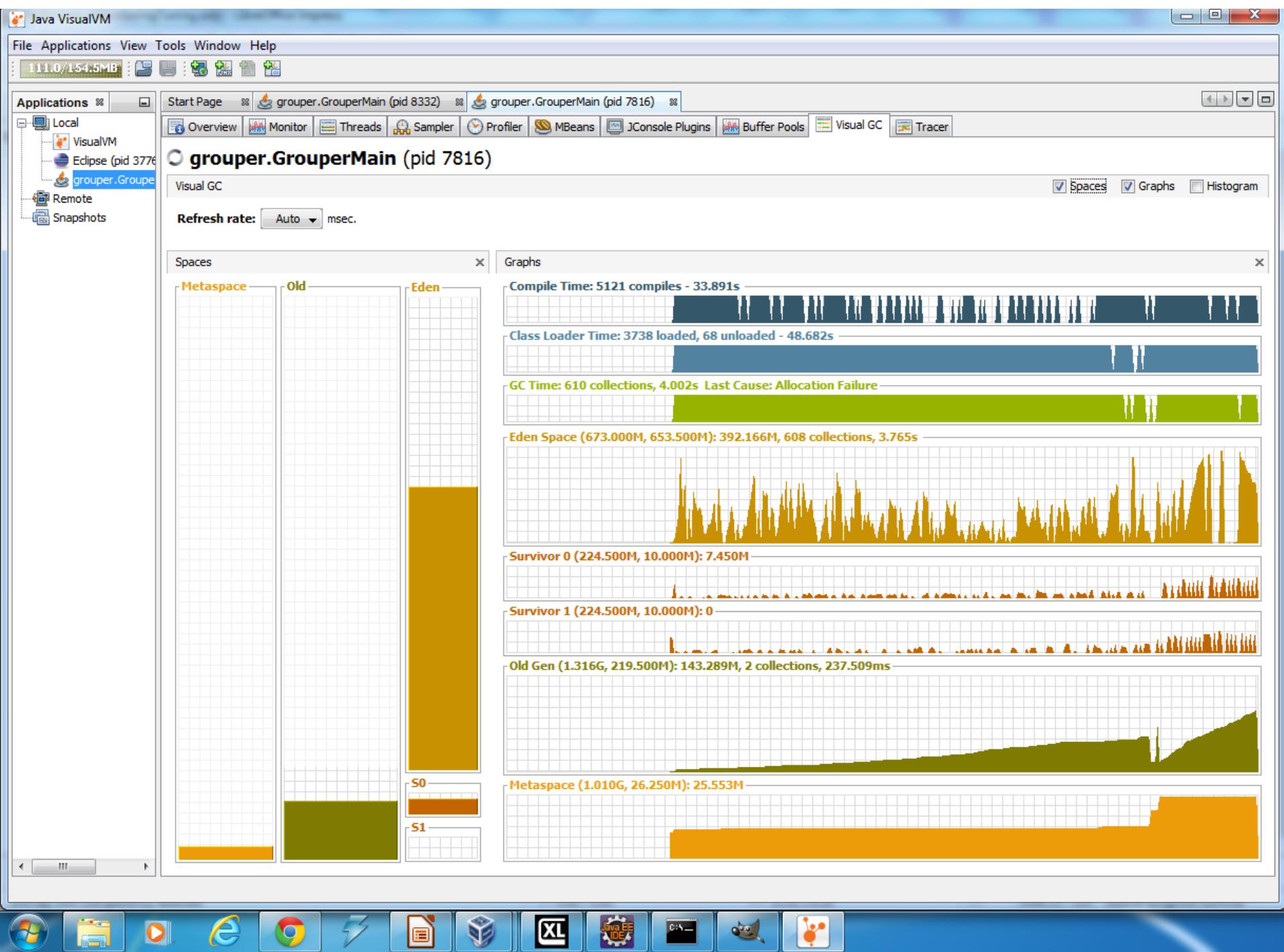
Attach Listener Finalizer ForkJoinPool.commonPool-worker-1 ForkJoinPool.commonPool-worker-2

2015-07-16 17:23:14

```
"ForkJoinPool.commonPool-worker-1" - Thread t@20
    java.lang.Thread.State: RUNNABLE
        at java.util.concurrent.CountedCompleter.tryComplete(CountedCompleter.java:583)
        at java.util.ArraysParallelSortHelpers$FJObjects$Merger.compute(ArraysParallelSortHelpers.java:227)
        at java.util.concurrent.CountedCompleter.exec(CountedCompleter.java:731)
        at java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:289)
```

Refresh

Windows Taskbar icons: File Explorer, Internet Explorer, Google Chrome, FileZilla, Java, Microsoft Word, Java EE IDE, Eclipse, MySQL Workbench, Oracle Database, Oracle VM VirtualBox.



# Thank's for Your Attention!



Trayan Iliev

CEO of IPT – Intellectual Products  
& Technologies

<http://iproduct.org/>

<https://github.com/iproduct>

<https://twitter.com/trayaniliev>

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