# Lecture 1

# Java SE – Programming

presentation

**Java Programming – Software App Development**

**Assoc. Prof. Cristian Toma Ph.D.**

D.I.C.E/D.E.I.C – Department of Economic Informatics & Cybernetics
www.dice.ase.ro

# cristian.toma@ie.ase.ro – Business Card



# **Agenda** for Lecture 1 – Summary of JSE

JVM – Java Virtual Machine, Java SE – loops, methods, array

# **JVM** & Java SE Fundamentals Intro

# 1.1 Java Virtual Machine Summary

**Compilers & Interpreters Questions:**

What is a virtual machine? How many types of JVM do you know?

What is the difference of stack based vs. registered based JVM and what are features of JIT compiler?

Should have a virtual machine associated one or more state machines?
What are advantages and disadvantages of virtual machines – native compilers vs. interpreters?

**Hello World sample** program in Oracle VM Box – Linux, MacOS and RaspberryPi

Command line compiling with JDK 9.0 | 8.0 – all lectures samples are command line based – and Eclipse OXYGEN projects in Linux Ubuntu 16 LTS. The sources are full in Git and alternatively is a Docker image for the lectures in the command line.

Linux Ubuntu 16 VM download:
**http://acs.ase.ro/java** ~ 15 GB HDD, 2 CPU cores, 4 GB RAM / use Oracle VM Box File->Import OVA file - All seminars will be in **/home/stud/javase/labs** directory and the lectures are in **/home/stud/javase/lectures** : ***user*=**stud / ***pass*=**stud

# 1.1 Java Object Oriented Programming Summary

**You had graduated from lectures of Java Fundamentals – let's talk:**

What is a class?

What is a package of classes in Java?

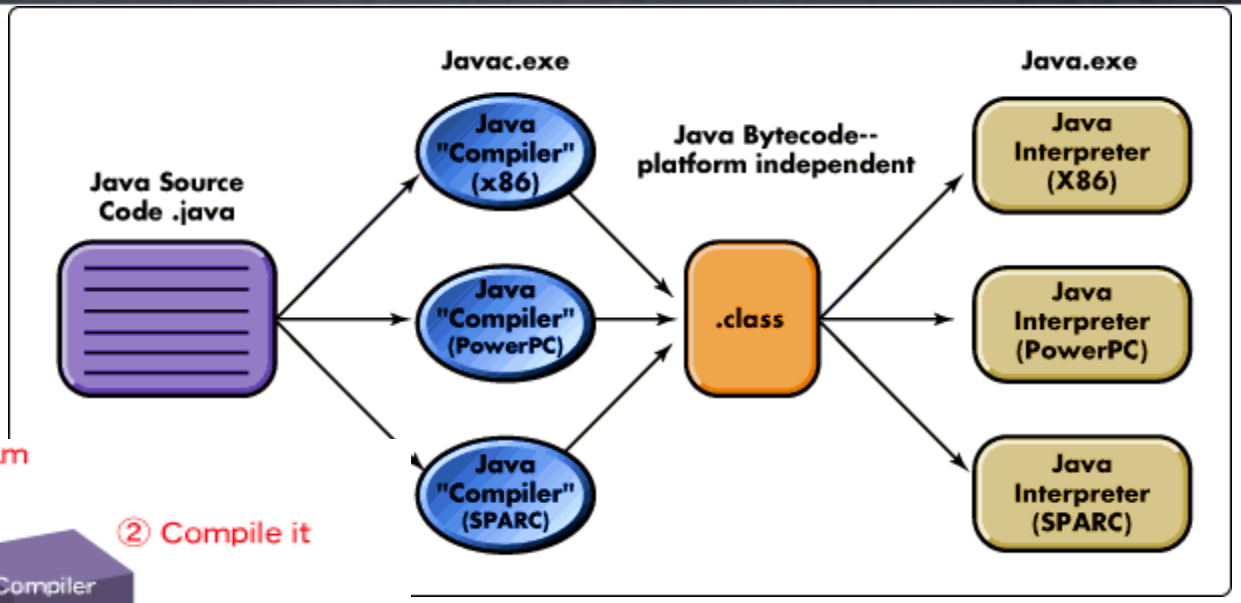
What is an object / instance?

How many bytes has an object inside JVM?

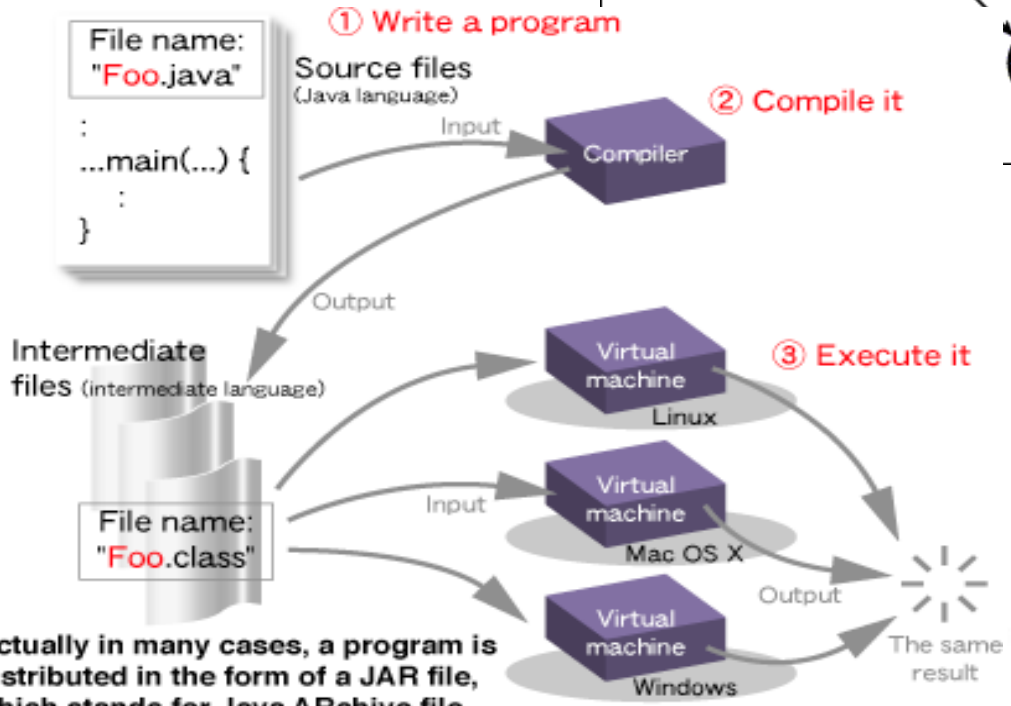
Why do we need clone, equals and hash methods?

Demo & memory model for the ***Certificate*** Java class

# 1.1 Java Overview Summary



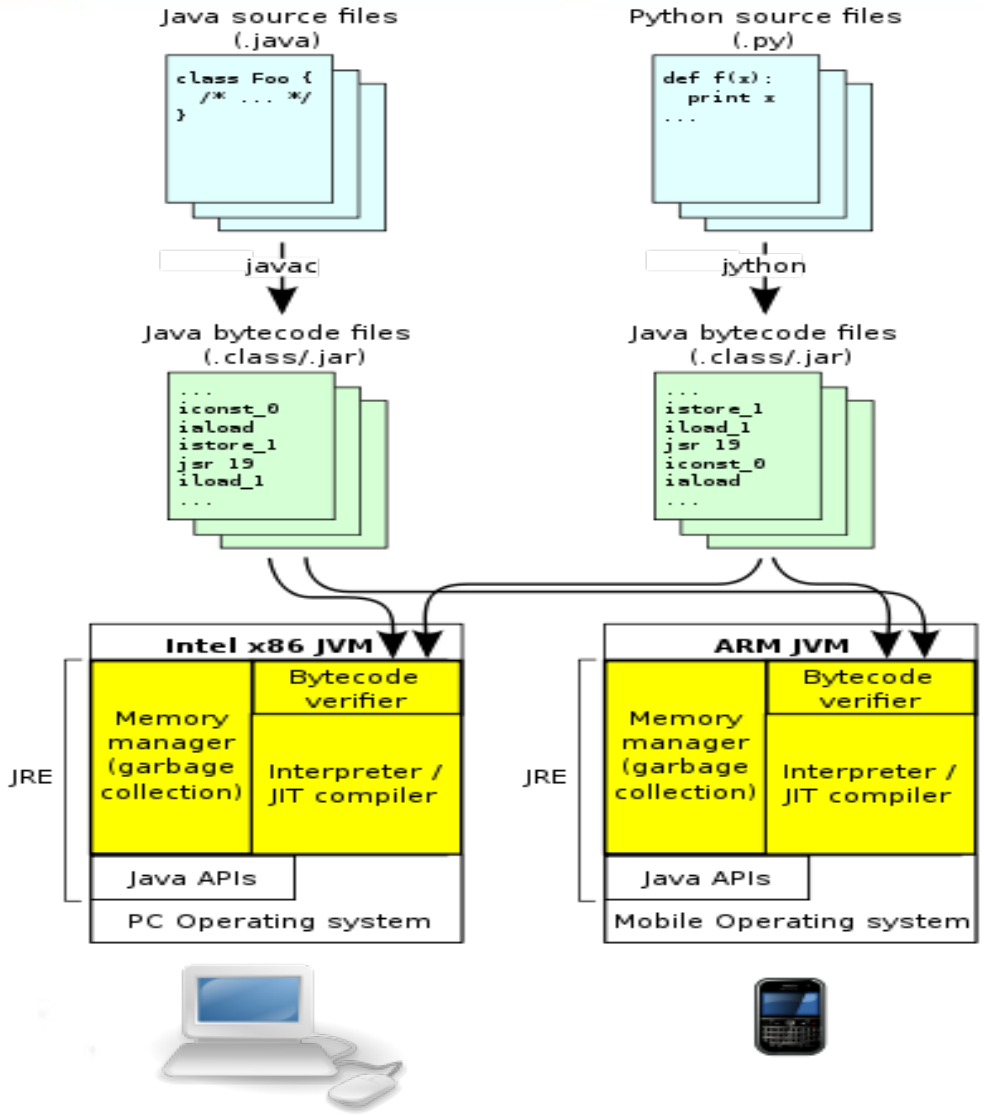

http://support.novell.com/techcenter/articles/ana19970701.html

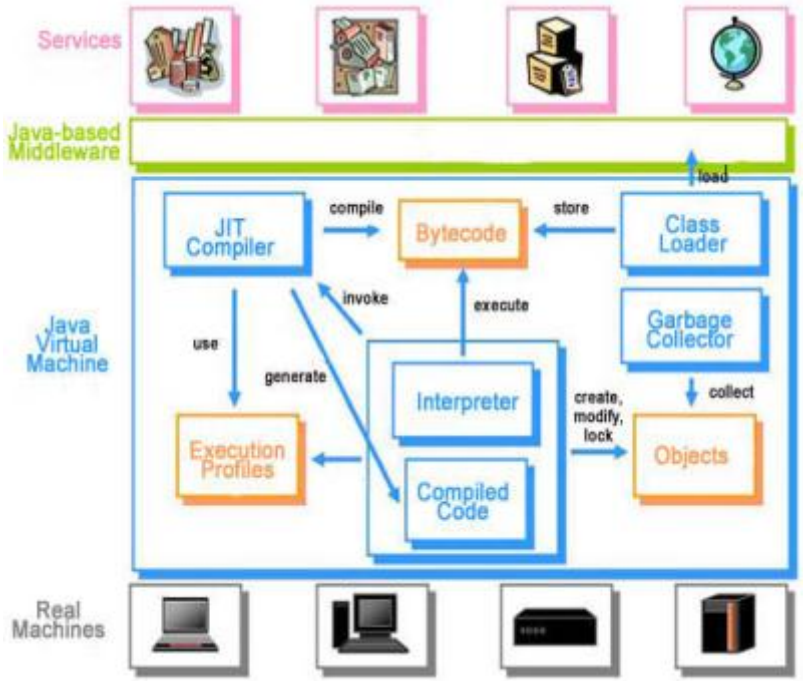



- **Actually in many cases, a program is distributed in the form of a JAR file, which stands for Java ARchive file.**

http://www.media-art-online.org/java/help/how-it-works.html

# 1.1 Java Virtual Machine Summary

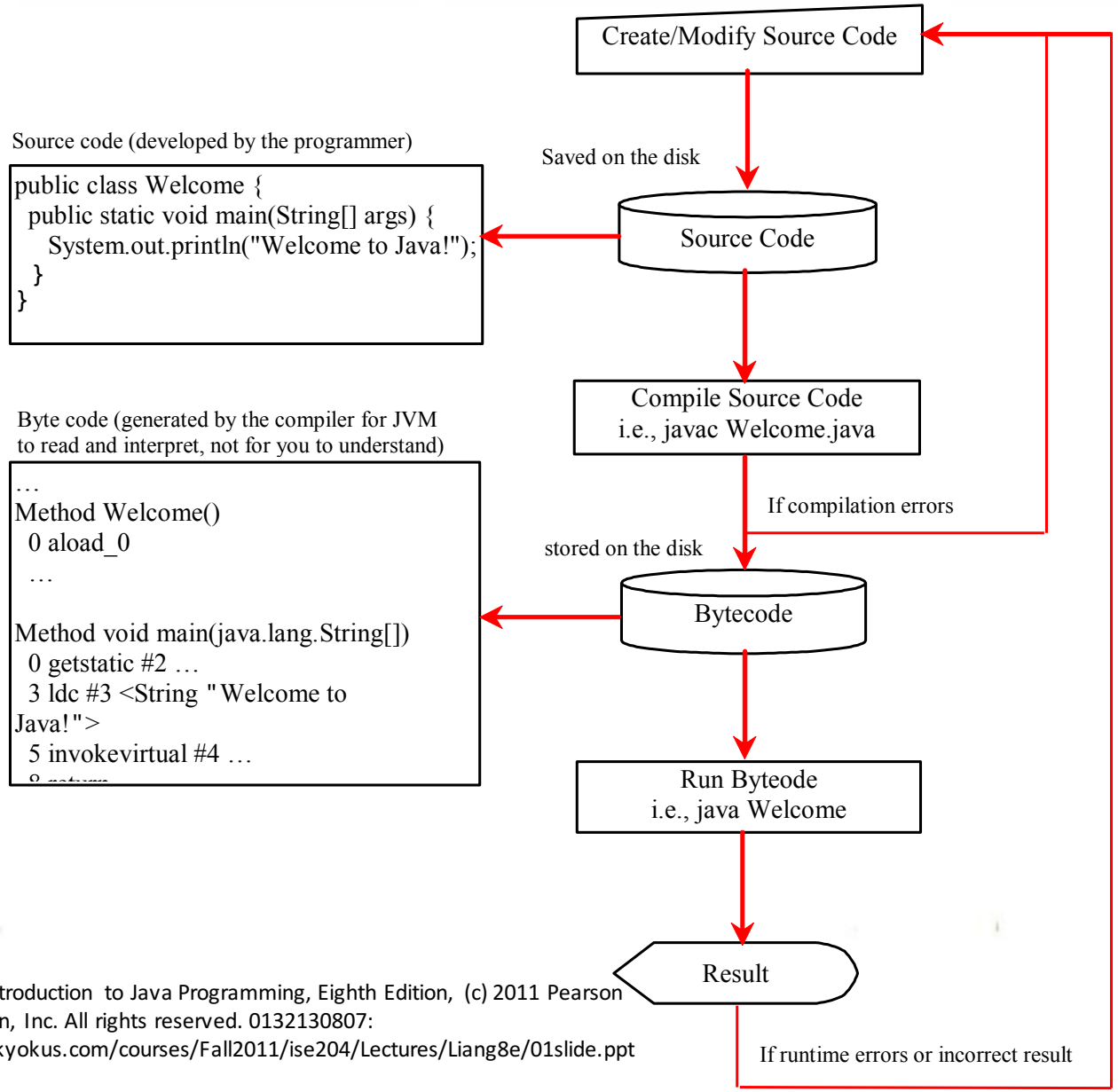



http://en.wikipedia.org/wiki/Java_virtual_machine

http://www.research.ibm.com/compsci/project_spotlight/plansoft/index.html

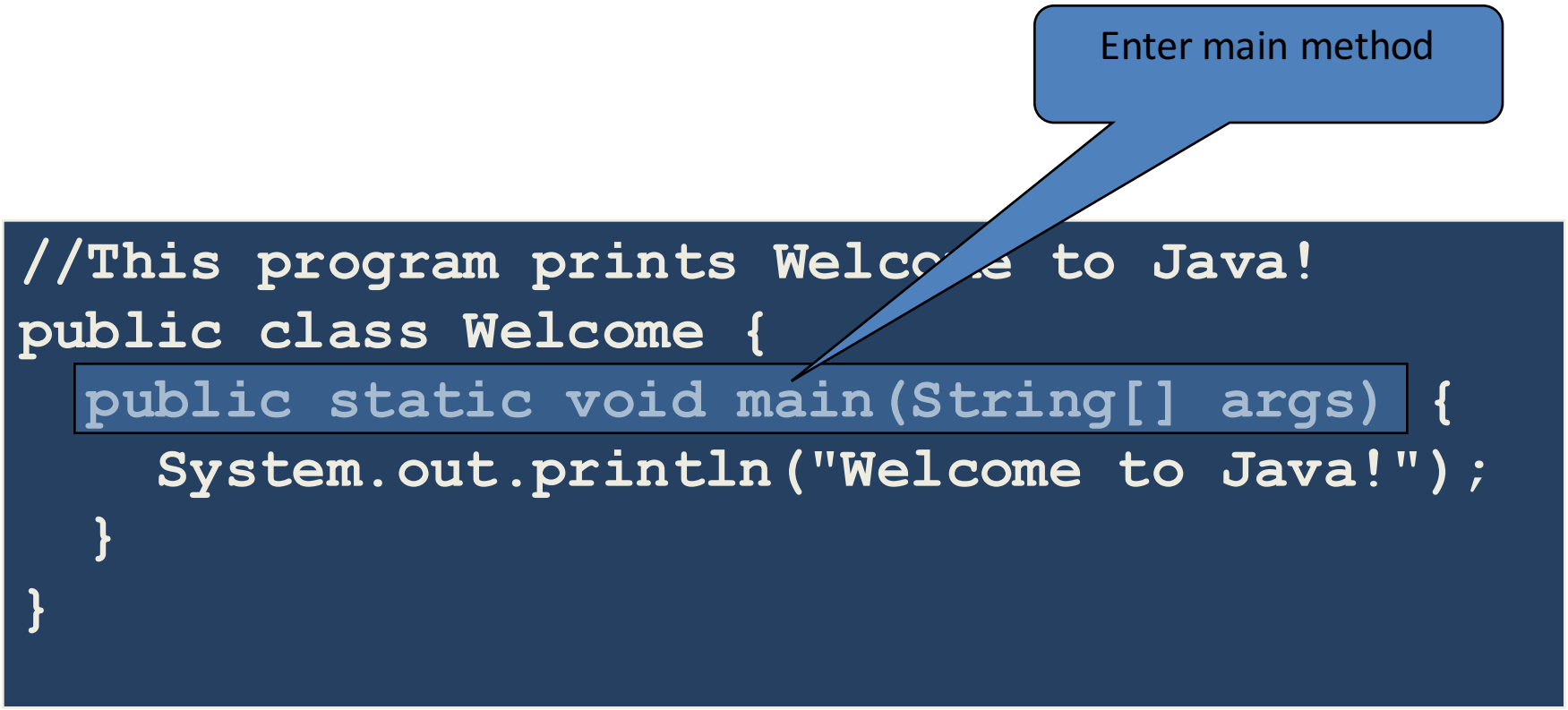
# 1.1 Java Creating, Compiling, and Running Programs



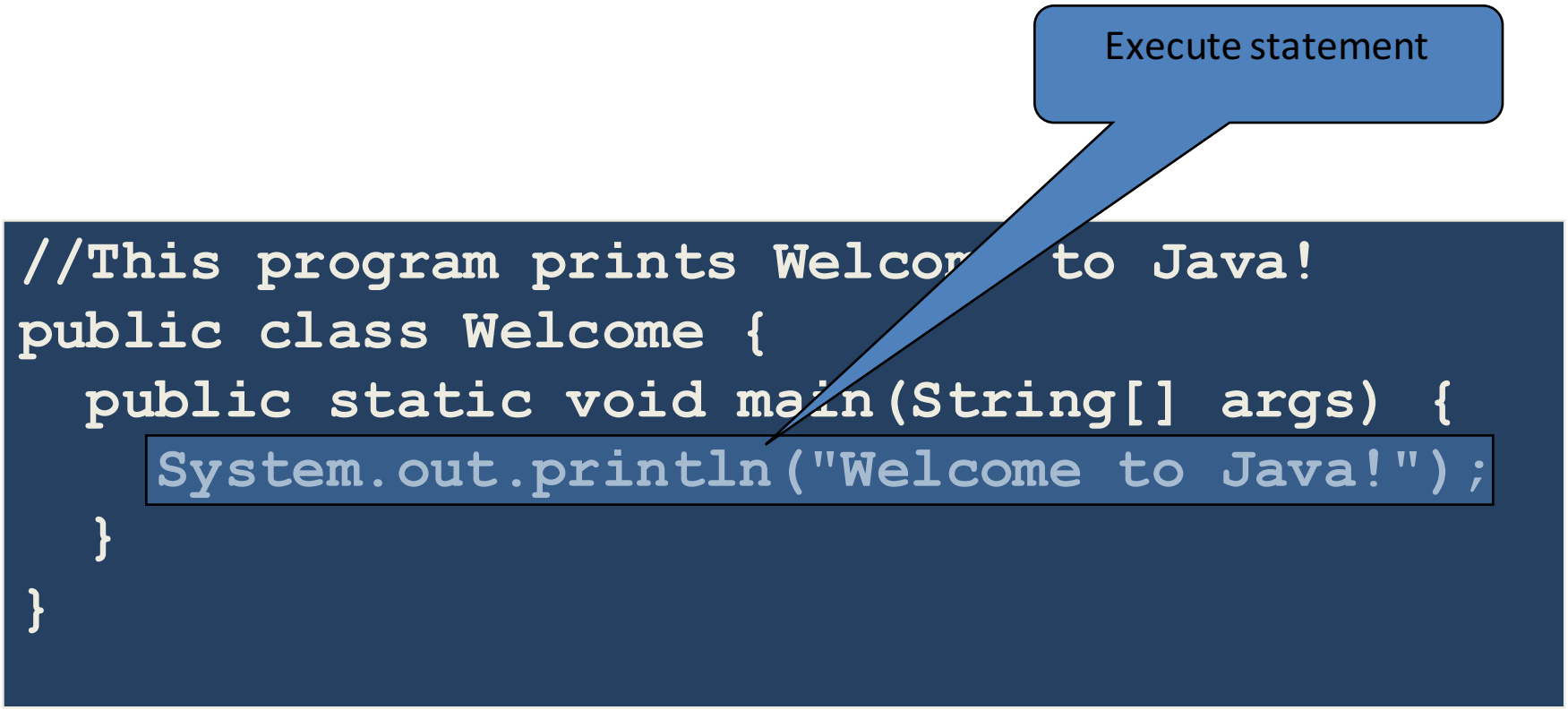

**DEMO**

**Compile in command prompt HelloWorld + show in GHex Ubuntu source and byte-code files in hex + Show Project in Eclipse + copy byte-code on Raspberry Pi and run**

# 1.1 Java Trace a Program Execution

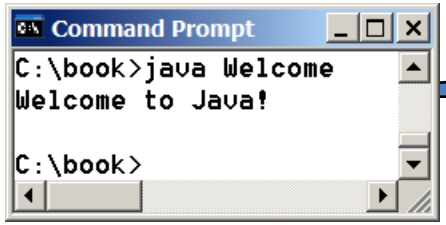


# 1.1 Java Trace a Program Execution



# 1.1 Java Trace a Program Execution

```
//This program prints Welcome to Java!
public class Welcome {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```




print a message to the console

# The `boolean` Type and Operators

Often in a program you need to compare two values, such as whether i is greater than j. Java provides six comparison operators (also known as relational operators) that can be used to compare two values. The result of the comparison is a Boolean value: true or false.

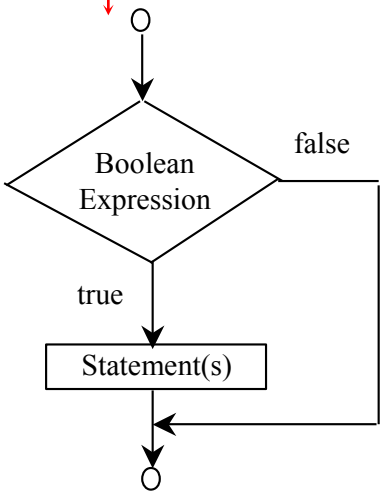
```
boolean b = (1 > 2);
```

# Comparison Operators

| *Operator* | *Name* |
|---|---|
| < | less than |
| <= | less than or equal to |
| > | greater than |
| >= | greater than or equal to |
| == | equal to |
| != | not equal to |

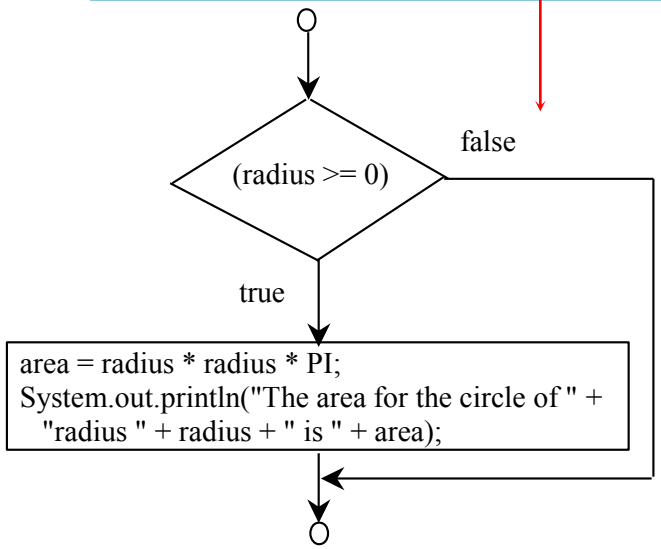
# One-way `if` Statements

```
if (boolean-expression) {
  statement(s);
}
```




(A)

```
if (radius >= 0) {
  area = radius * radius * PI;
  System.out.println("The area"
    + " for the circle of radius "
    + radius + " is " + area);
}
```




(B)

# Note

```
if i > 0 {
  System.out.println("i is positive");
}
```

(a) Wrong

```
if (i > 0) {
  System.out.println("i is positive");
}
```

(b) Correct

```
if (i > 0) {
  System.out.println("i is positive");
}
```

(a)

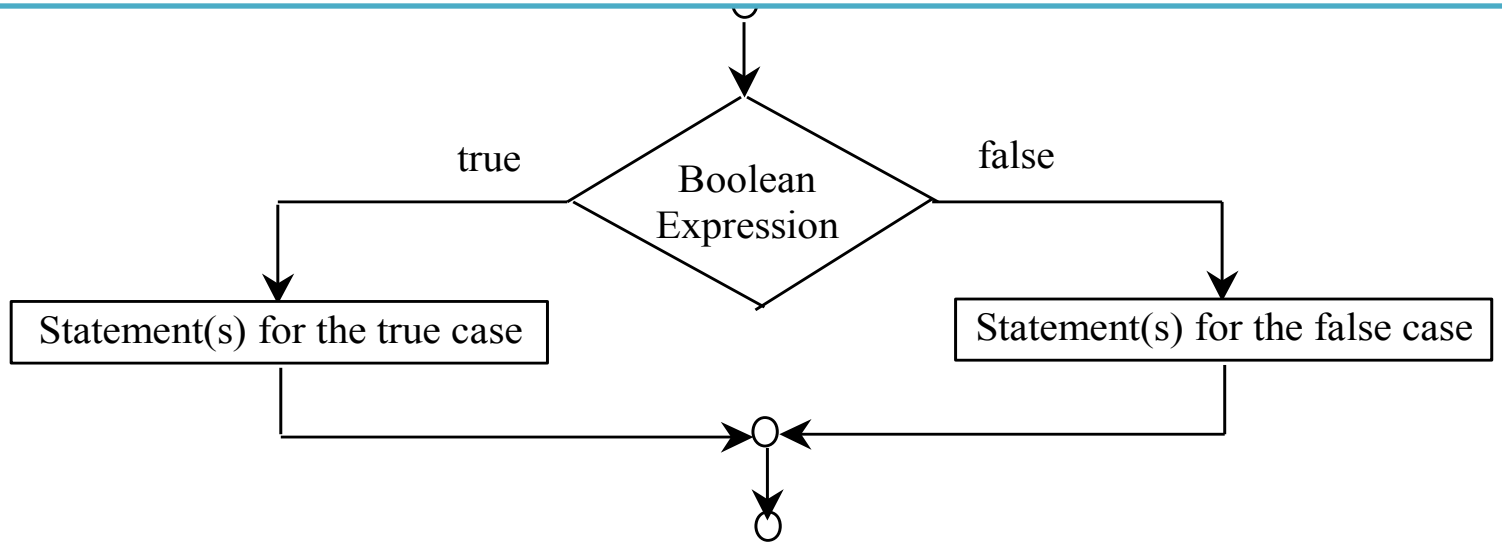
Equivalent

```
if (i > 0)
  System.out.println("i is positive");
```

(b)

# The Two-way `if` Statement

```
if (boolean-expression) {
  statement(s)-for-the-true-case;
}
else {
  statement(s)-for-the-false-case;
}
```



# `if...else` Example

```
if (radius >= 0) {
  area = radius * radius * 3.14159;

  System.out.println("The area for the “
    + “circle of radius " + radius +
    " is " + area);
}
else {
  System.out.println("Negative input");
}
```

# Multiple Alternative if Statements

```
if (score >= 90.0)
  grade = 'A';
else
  if (score >= 80.0)
    grade = 'B';
  else
    if (score >= 70.0)
      grade = 'C';
    else
      if (score >= 60.0)
        grade = 'D';
      else
        grade = 'F';
```

Equivalent

```
if (score >= 90.0)
  grade = 'A';
else if (score >= 80.0)
  grade = 'B';
else if (score >= 70.0)
  grade = 'C';
else if (score >= 60.0)
  grade = 'D';
else
  grade = 'F';
```

# Trace if-else statement

Suppose score is 70.0

The condition is false

```
if (score >= 90.0)
  grade = 'A';
else if (score >= 80.0)
  grade = 'B';
else if (score >= 70.0)
  grade = 'C';
else if (score >= 60.0)
  grade = 'D';
else
  grade = 'F';
```

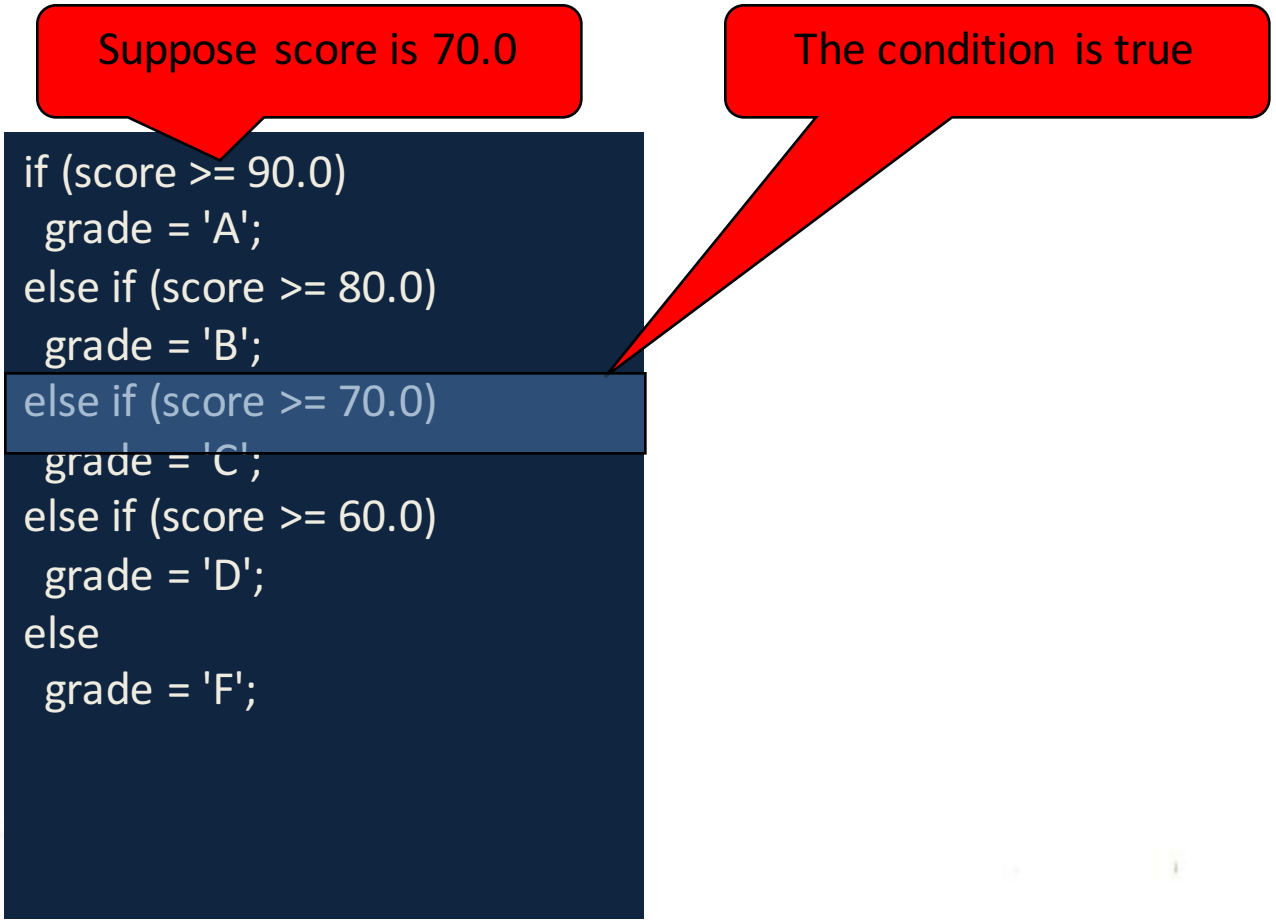
# Trace if-else statement

Suppose score is 70.0

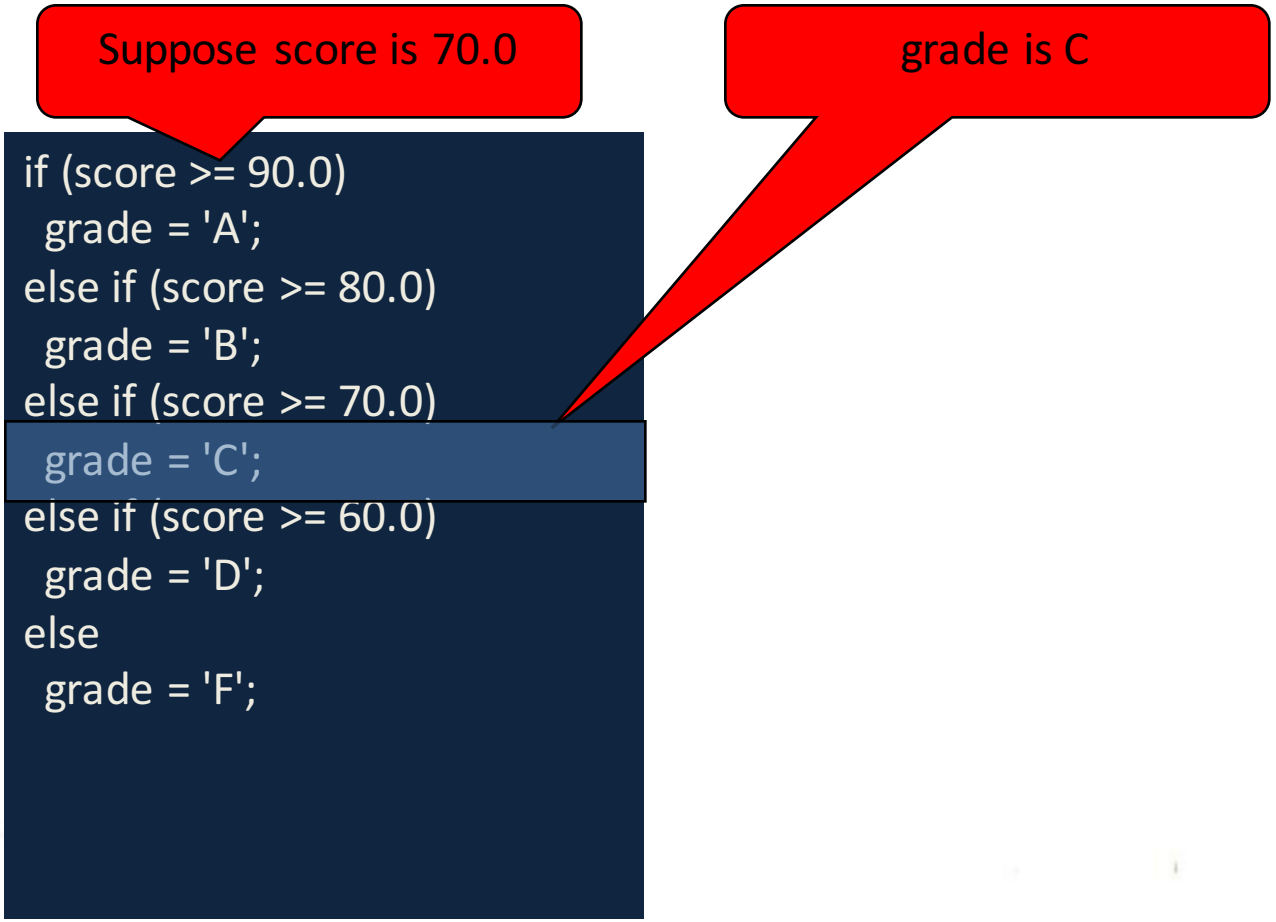
The condition is false

```
if (score >= 90.0)
  grade = 'A';
else if (score >= 80.0)
  grade = 'B';
else if (score >= 70.0)
  grade = 'C';
else if (score >= 60.0)
  grade = 'D';
else
  grade = 'F';
```

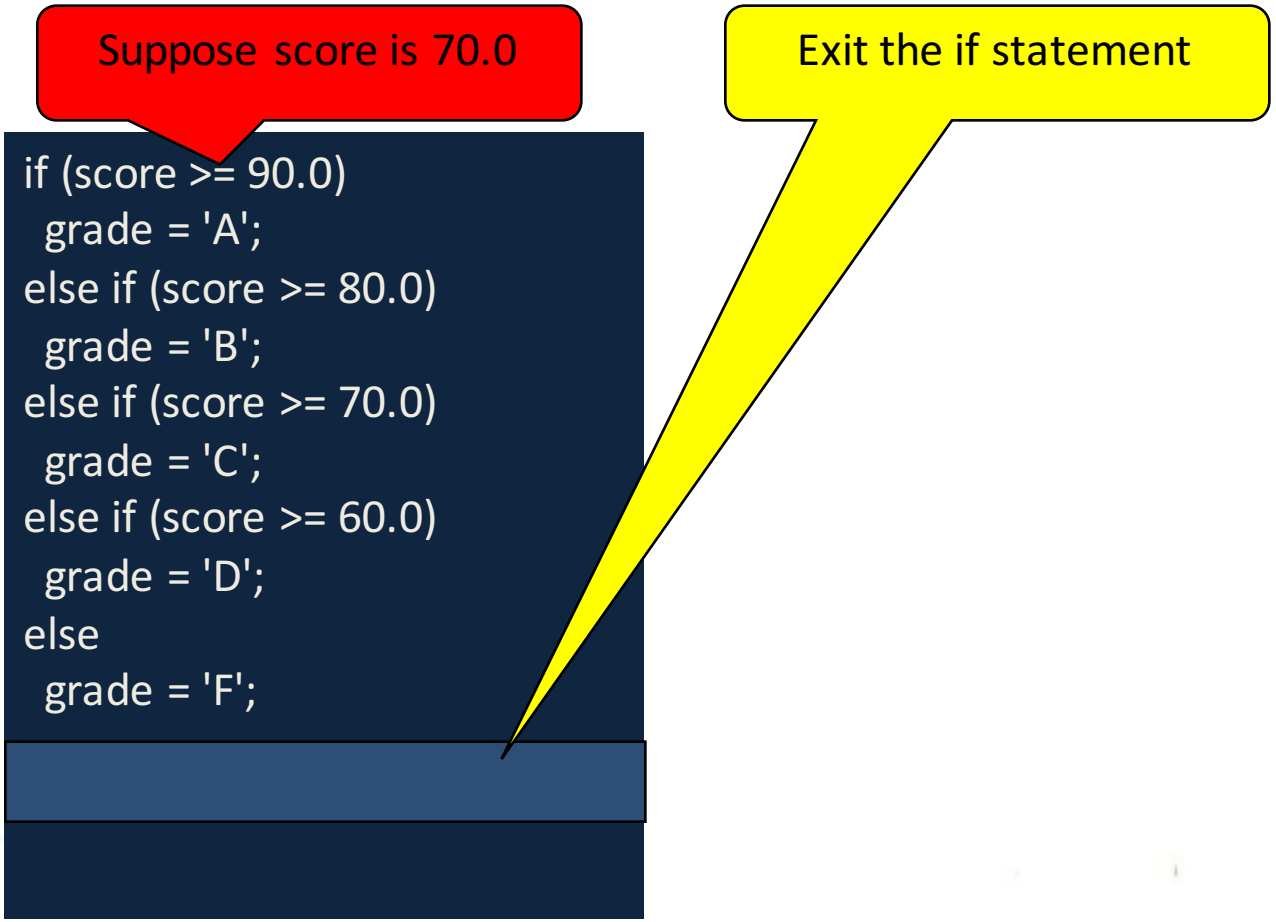
# Trace if-else statement



# Trace if-else statement



# Trace if-else statement



# Note

The else clause matches the most recent if clause in the same block.

```
int i = 1;
int j = 2;
int k = 3;

if (i > j)
  if (i > k)
    System.out.println("A");
else
    System.out.println("B");
```

(a)

Equivalent

```
int i = 1;
int j = 2;
int k = 3;

if (i > j)
  if (i > k)
    System.out.println("A");
  else
    System.out.println("B");
```

(b)

# Note, cont.

Nothing is printed from the preceding statement. To force the else clause to match the first if clause, you must add a pair of braces:

```
int i = 1;
int j = 2;
int k = 3;
if (i > j) {
  if (i > k)
    System.out.println("A");
}
else
  System.out.println("B");
```

This statement prints B.

# Common Errors

Adding a semicolon at the end of an if clause is a common mistake.

```
if (radius >= 0);
{
  area = radius*radius*PI;
  System.out.println(
    "The area for the circle of radius " +
    radius + " is " + area);
}
```

Wrong

This mistake is hard to find, because it is not a compilation error or a runtime error, it is a logic error.

This error often occurs when you use the next-line block style.

# TIP

```
if (number % 2 == 0)
  even = true;
else
  even = false;
```

(a)

Equivalent

```
boolean even
  = number % 2 == 0;
```

(b)

# CAUTION

```
if (even == true)
  System.out.println(
    "It is even.");
```

(a)

Equivalent

```
if (even)
  System.out.println(
    "It is even.");
```

(b)

# Problem: Computing Taxes

The US federal personal income tax is calculated based on the filing status and taxable income. There are four filing statuses: single filers, married filing jointly, married filing separately, and head of household. The tax rates for 2009 (2014?) are shown below.

| Marginal Tax Rate | Single | Married Filing Jointly or Qualified Widow(er) | Married Filing Separately | Head of Household |
|---|---|---|---|---|
| 10% | $0 – $8,350 | $0 – $16,700 | $0 – $8,350 | $0 – $11,950 |
| 15% | $8,351–$33,950 | $16,701 – $67,900 | $8,351 – $33,950 | $11,951 – $45,500 |
| 25% | $33,951 – $82,250 | $67,901 – $137,050 | $33,951 – $68,525 | $45,501 – $117,450 |
| 28% | $82,251 – $171,550 | $137,051 – $208,850 | $68,525 – $104,425 | $117,451 – $190,200 |
| 33% | $171,551 – $372,950 | $208,851 – $372,950 | $104,426 – $186,475 | $190,201 - $372,950 |
| 35% | $372,951+ | $372,951+ | $186,476+ | $372,951+ |

# 1.2 Java Selections – Boolean Operators, if-else, switch statements

```
import java.util.Scanner;

public class ComputeTax {
  public static void main(String[]
args) {
    // Create a Scanner
    Scanner input = new
Scanner(System.in);
    // Prompt the user to enter filing
status
    System.out.print(
      "(0-single filer, 1-married
jointly,\n" +
      "2-married separately, 3-head of
household)\n" +
      "Enter the filing status: ");
    int status = input.nextInt();
    // Prompt the user to enter
taxable income
    System.out.print("Enter the
taxable income: ");
    double income =
input.nextDouble();
    // Compute tax
    double tax = 0;
```

```
if (status == 0) { // Compute tax for single filers
    if (income <= 8350)
      tax = income * 0.10;
    else if (income <= 33950)
      tax = 8350 * 0.10 + (income - 8350) * 0.15;
    else if (income <= 82250)
      tax = 8350 * 0.10 + (33950 - 8350) * 0.15 +
        (income - 33950) * 0.25;
    else if (income <= 171550)
      tax = 8350 * 0.10 + (33950 - 8350) * 0.15 +
        (82250 - 33950) * 0.25 + (income - 82250) * 0.28;
    else if (income <= 372950)
      tax = 8350 * 0.10 + (33950 - 8350) * 0.15 +
        (82250 - 33950) * 0.25 + (171550 - 82250) * 0.28 +
        (income - 171550) * 0.35;
    else
      tax = 8350 * 0.10 + (33950 - 8350) * 0.15 +
        (82250 - 33950) * 0.25 + (171550 - 82250) * 0.28 +
        (372950 - 171550) * 0.33 + (income - 372950) * 0.35;
  }
  else if (status == 1) { // Compute tax for married file jointly
    // Left as exercise
  }
  else if (status == 2) { // Compute tax for married separately
    // Left as exercise
  }
  else if (status == 3) { // Compute tax for head of household
    // Left as exercise
  }
  else {
    System.out.println("Error: invalid status");
    System.exit(0);
  }

  // Display the result
  System.out.println("Tax is " + (int)(tax * 100) / 100.0);
}
}
```

# ASSIGNMENT 01 - Problem: Computing Taxes, cont.

```
if (status == 0) {
  // Compute tax for single filers
}
else if (status == 1) {
  // Compute tax for married file jointly
}
else if (status == 2) {
  // Compute tax for married file separately
}
else if (status == 3) {
  // Compute tax for head of household
}
else {
  // Display wrong status
}
```

# Logical Operators

| *Operator* | *Name* |
|---|---|
| **!** | not |
| **&&** | and |
| **\|\|** | or |
| **^** | exclusive or |

# Truth Table for Operator !

| p | !p | Example (assume age = 24, gender = 'M') |
|---|---|---|
| true | false | !(age > 18) is false, because (age > 18) is true. |
| false | true | !(gender != 'F') is true, because (grade != 'F') is false. |

| p | !p | Example |
|---|---|---|
| true | false | !(1 > 2) is true, because (1 > 2) is false. |
| false | true | !(1 > 0) is false, because (1 > 0) is true. |

# Truth Table for Operator &&

| p1 | p2 | p1 && p2 | Example (assume age = 24, gender = 'F') |
|---|---|---|---|
| false | false | false | (age > 18) && (gender == 'F') is true, because (age > 18) and (gender == 'F') are both true. |
| false | true | false | |
| true | false | false | (age > 18) && (gender != 'F') is false, because (gender != 'F') is false. |
| true | true | true | |

| p1 | p2 | p1 && p2 | Example |
|---|---|---|---|
| false | false | false | (3 > 2) && (5 >= 5) is true, because (3 > 2) and (5 >= 5) are both true. |
| false | true | false | |
| true | false | false | (3 > 2) && (5 > 5) is false, because (5 > 5) is false. |
| true | true | true | |

# Truth Table for Operator ||

| p1 | p2 | p1 \|\| p2 | Example (assume age = 24, gender = 'F') |
|---|---|---|---|
| false | false | false | (age > 34) \|\| (gender == 'F') is true, because (gender == 'F') is true. |
| false | true | true | |
| true | false | true | (age > 34) \|\| (gender == 'M') is false, because (age > 34) and (gender == 'M') are both false. |
| true | true | true | |

| p1 | p2 | p1 \|\| p2 | Example |
|---|---|---|---|
| false | false | false | (2 > 3) \|\| (5 > 5) is false, because (2 > 3) and (5 > 5) are both false. |
| false | true | true | |
| true | false | true | (3 > 2) \|\| (5 > 5) is true, because (3 > 2) is true. |
| true | true | true | |

# Truth Table for Operator ^

| p1 | p2 | p1 ^ p2 | Example (assume age = 24, gender = 'F') |
|---|---|---|---|
| false | false | false | (age > 34) ^ (gender == 'F') is true, because (age > 34) is false but (gender == 'F') is true. |
| false | true | true | |
| true | false | true | (age > 34) \|\| (gender == 'M') is false, because (age > 34) and (gender == 'M') are both false. |
| true | true | false | |

# Examples

```
System.out.println("Is " + number + " divisible by 2 and 3? " +
  ((number % 2 == 0) && (number % 3 == 0)));


System.out.println("Is " + number + " divisible by 2 or 3? " +
  ((number % 2 == 0) || (number % 3 == 0)));


 System.out.println("Is " + number +
   " divisible by 2 or 3, but not both? " +
   ((number % 2 == 0) ^ (number % 3 == 0)));
```

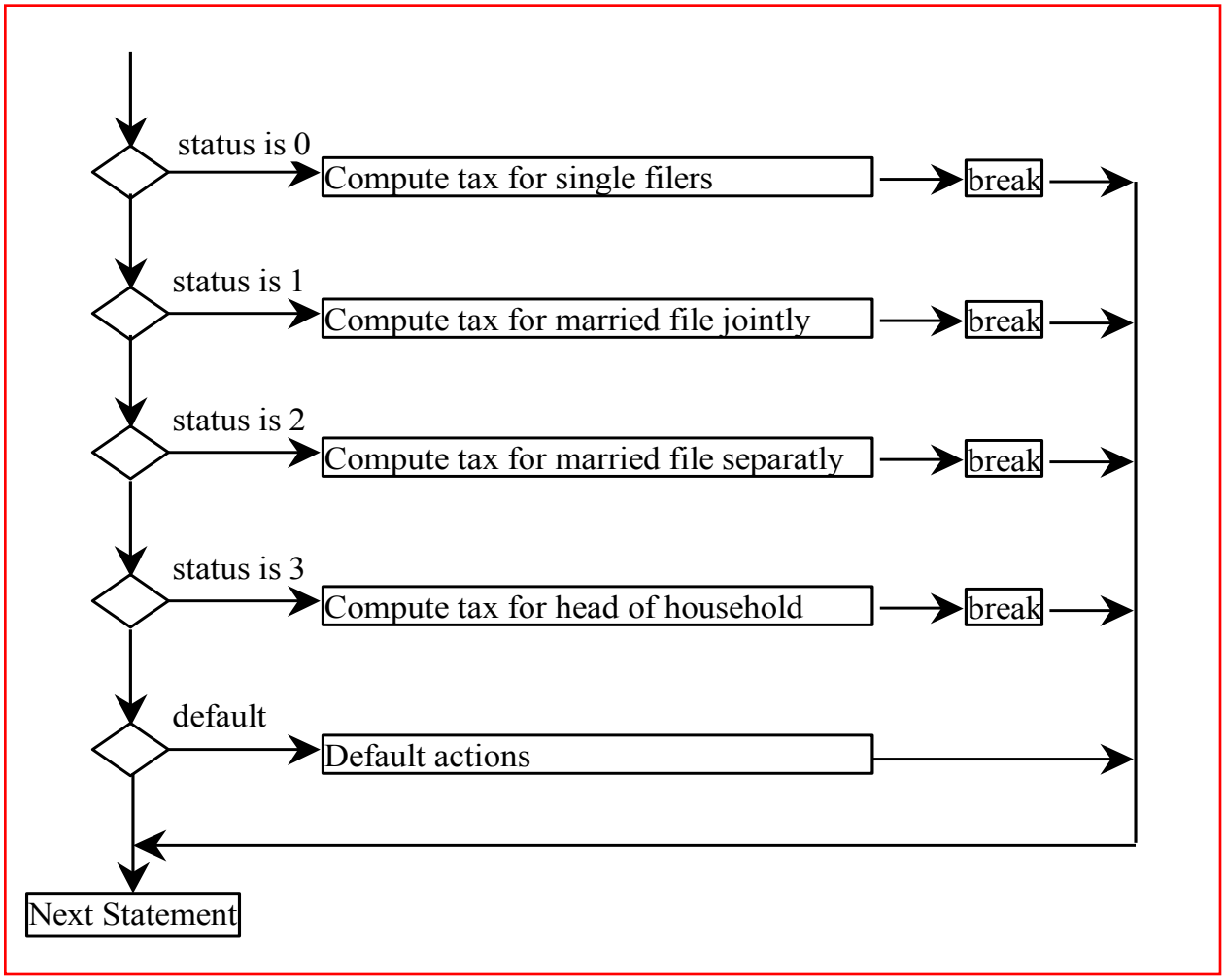
# The & and | Operators

```
If x is 1, what is x after this expression?
(x > 1) & (x++ < 10)

If x is 1, what is x after this expression?
(1 > x) && ( 1 > x++)

How about (1 == x) | (10 > x++)?
(1 == x) || (10 > x++)?
```

# `switch` Statements

```
switch (status) {
  case 0:  compute taxes for single filers;
           break;
  case 1:  compute taxes for married file jointly;
           break;
  case 2:  compute taxes for married file separately;
           break;
  case 3:  compute taxes for head of household;
           break;
  default: System.out.println("Errors: invalid status");
           System.exit(0);
}
```

# `switch` Statement Flow Chart



# `switch` Statement Rules

The switch-expression must yield a value of char, byte, short, or int type and must always be enclosed in parentheses.

The value1, ..., and valueN must have the same data type as the value of the switch-expression. The resulting statements in the case statement are executed when the value in the case statement matches the value of the switch-expression. Note that value1, ..., and valueN are constant expressions, meaning that they cannot contain variables in the expression, such as 1 + x.

```
switch (switch-expression) {
  case value1:  statement(s)1;
                break;
  case value2: statement(s)2;
                break;
  …
  case valueN: statement(s)N;
                break;
  default: statement(s)-for-default;
}
```

# `switch` Statement Rules

The keyword break is optional, but it should be used at the end of each case in order to terminate the remainder of the switch statement. If the break statement is not present, the next case statement will be executed.

The default case, which is optional, can be used to perform actions when none of the specified cases matches the switch-expression.

```
switch (switch-expression) {
  case value1:  statement(s)1;
        break;
  case value2: statement(s)2;
        break;
  …
  case valueN: statement(s)N;
        break;
  default: statement(s)-for-default;
}
```

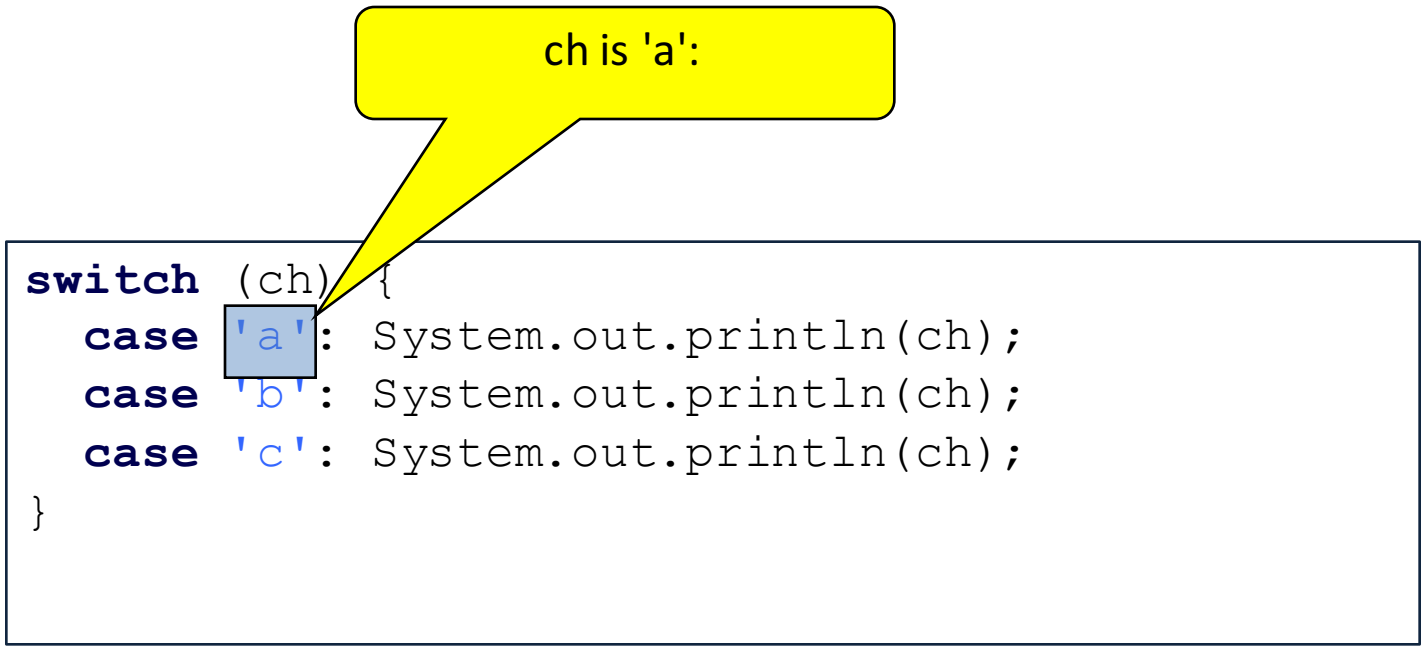
The case statements are executed in sequential order, but the order of the cases (including the default case) does not matter. However, it is good programming style to follow the logical sequence of the cases and place the default case at the end.

# Trace switch statement

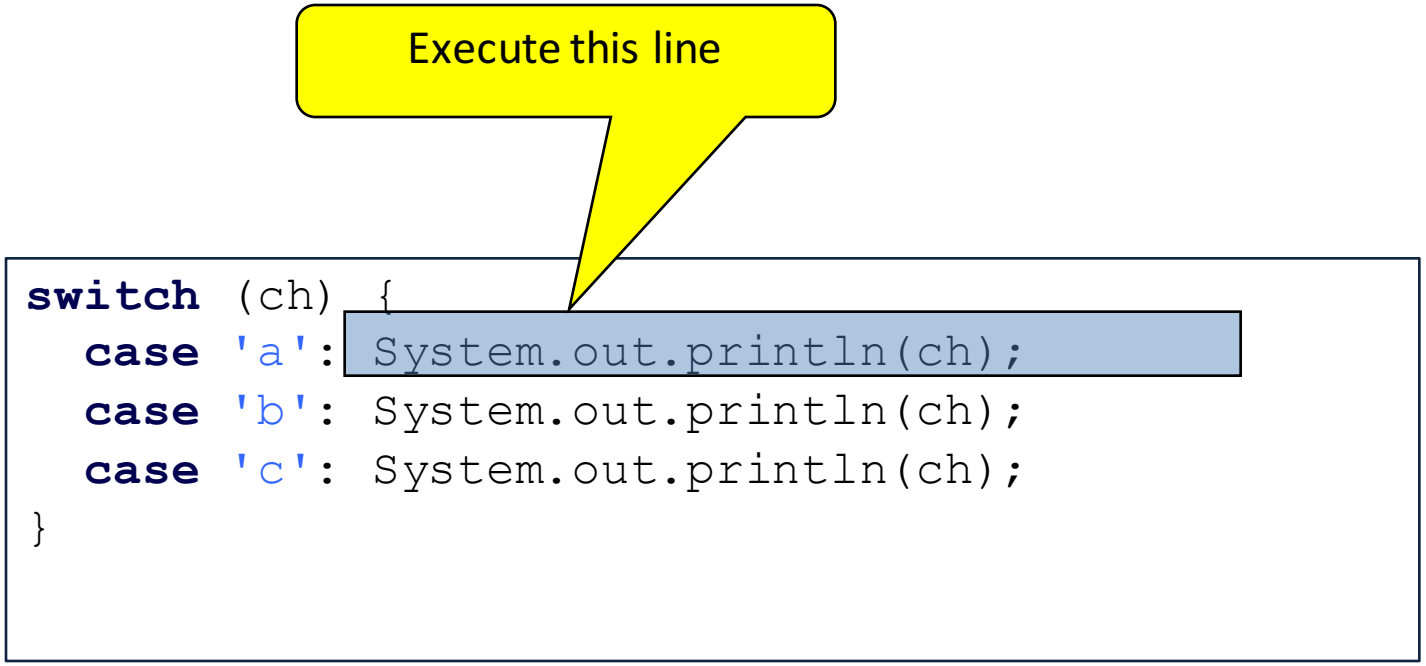
Suppose ch is 'a':

```
switch (ch) {
  case 'a': System.out.println(ch);
  case 'b': System.out.println(ch);
  case 'c': System.out.println(ch);
}
```

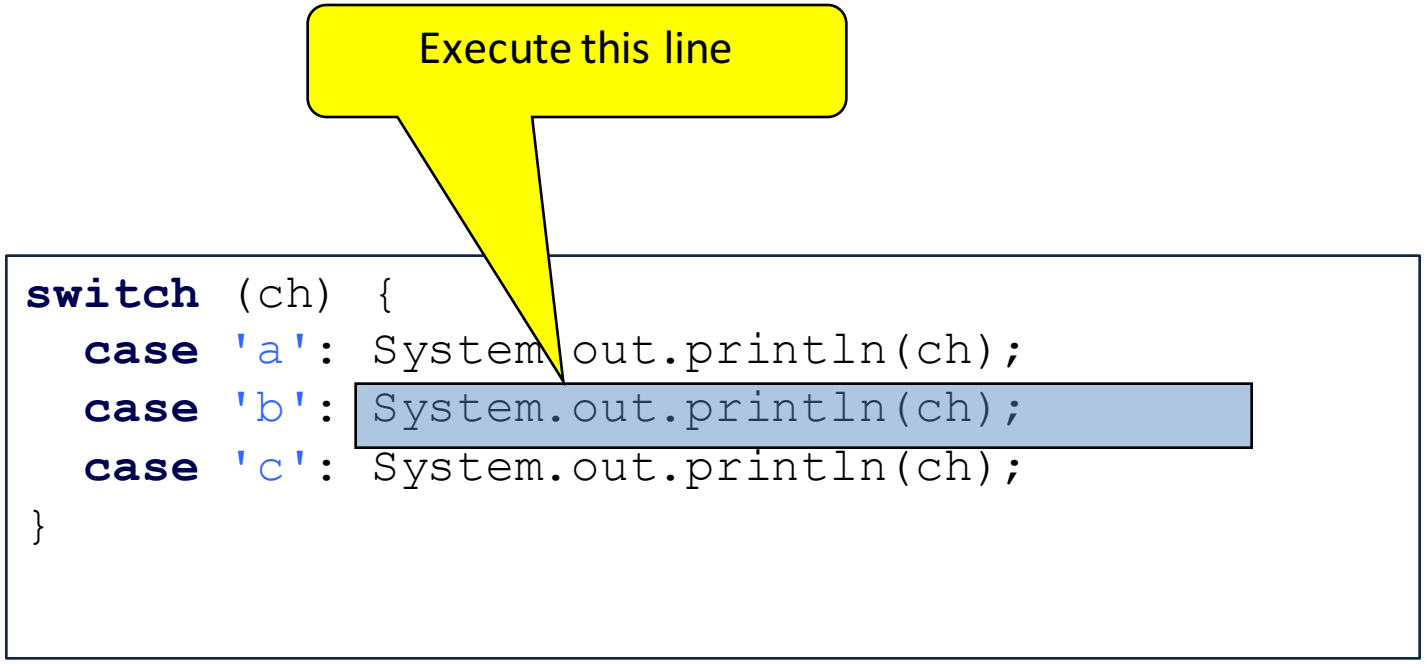
# Trace switch statement



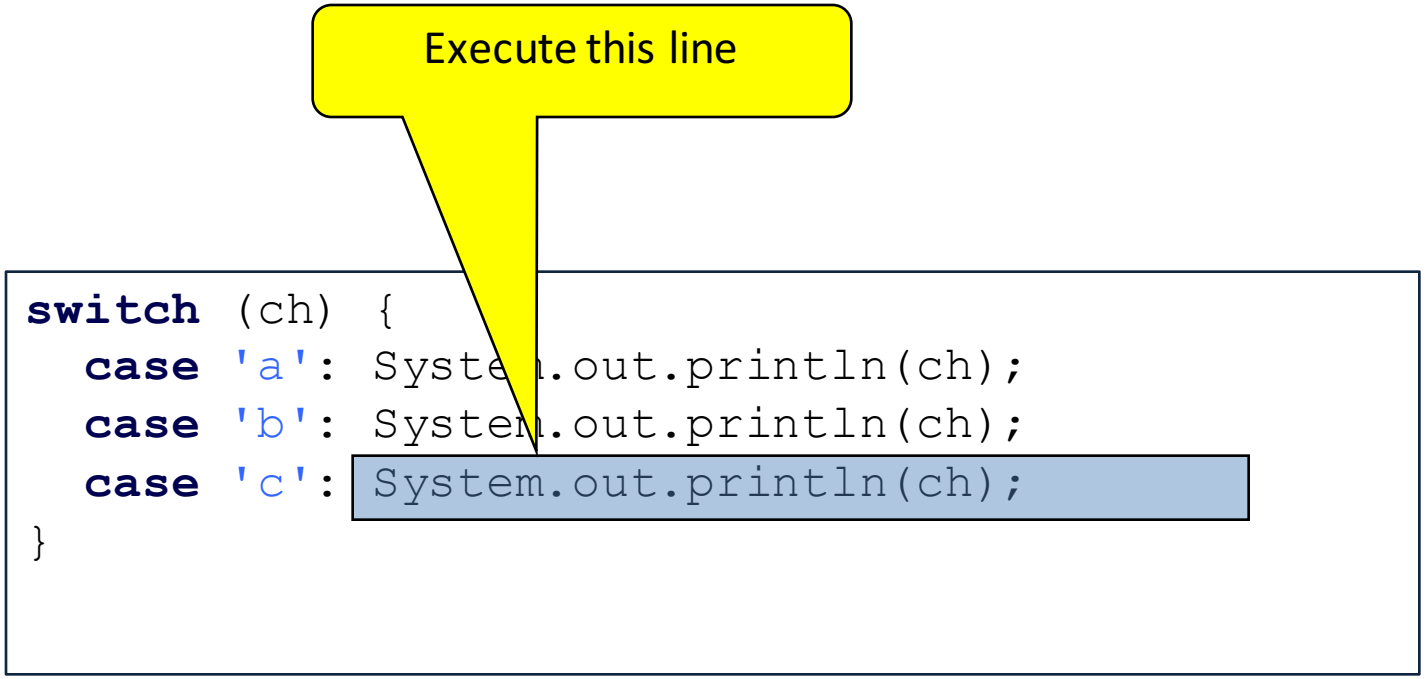
# Trace switch statement



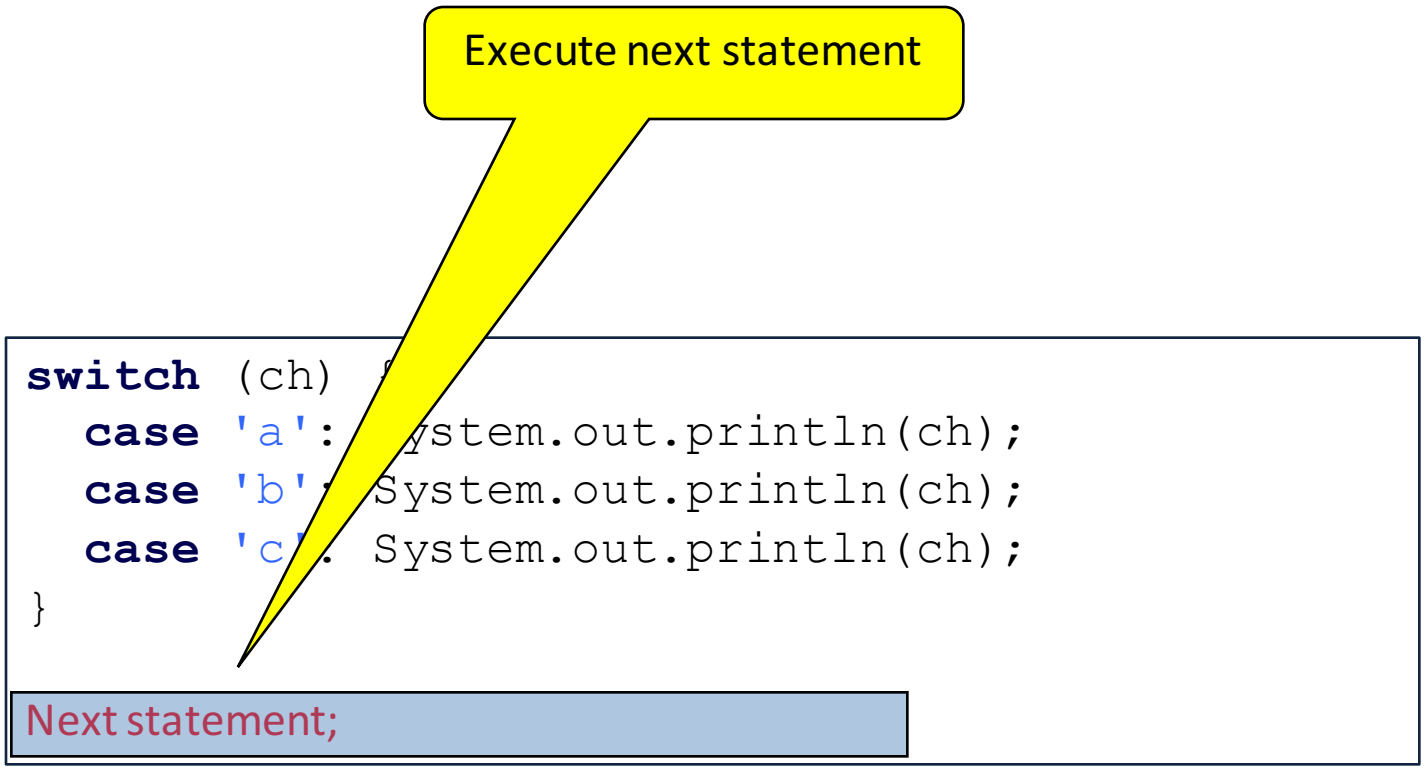
# Trace switch statement



# Trace switch statement



# Trace switch statement

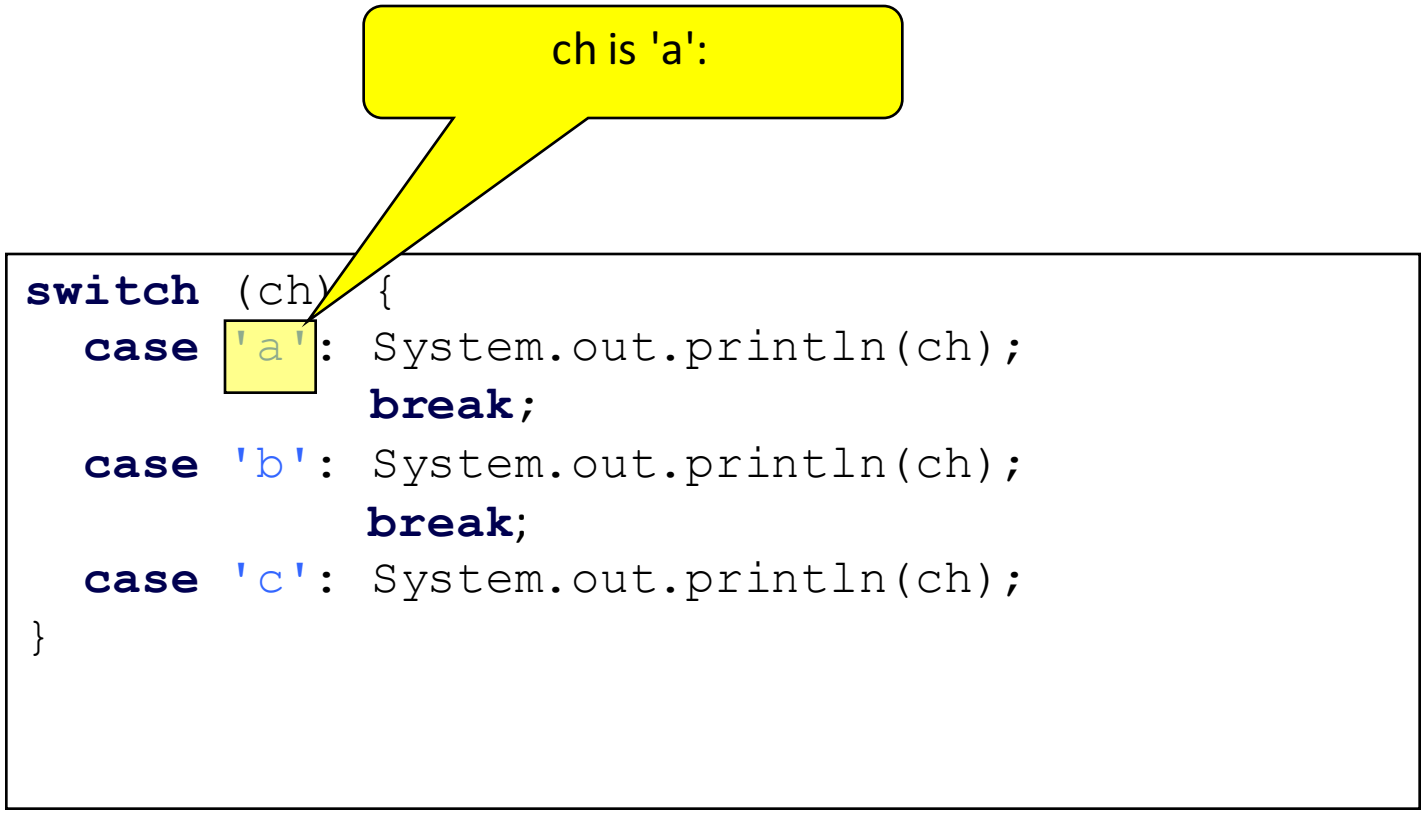


# Trace switch statement

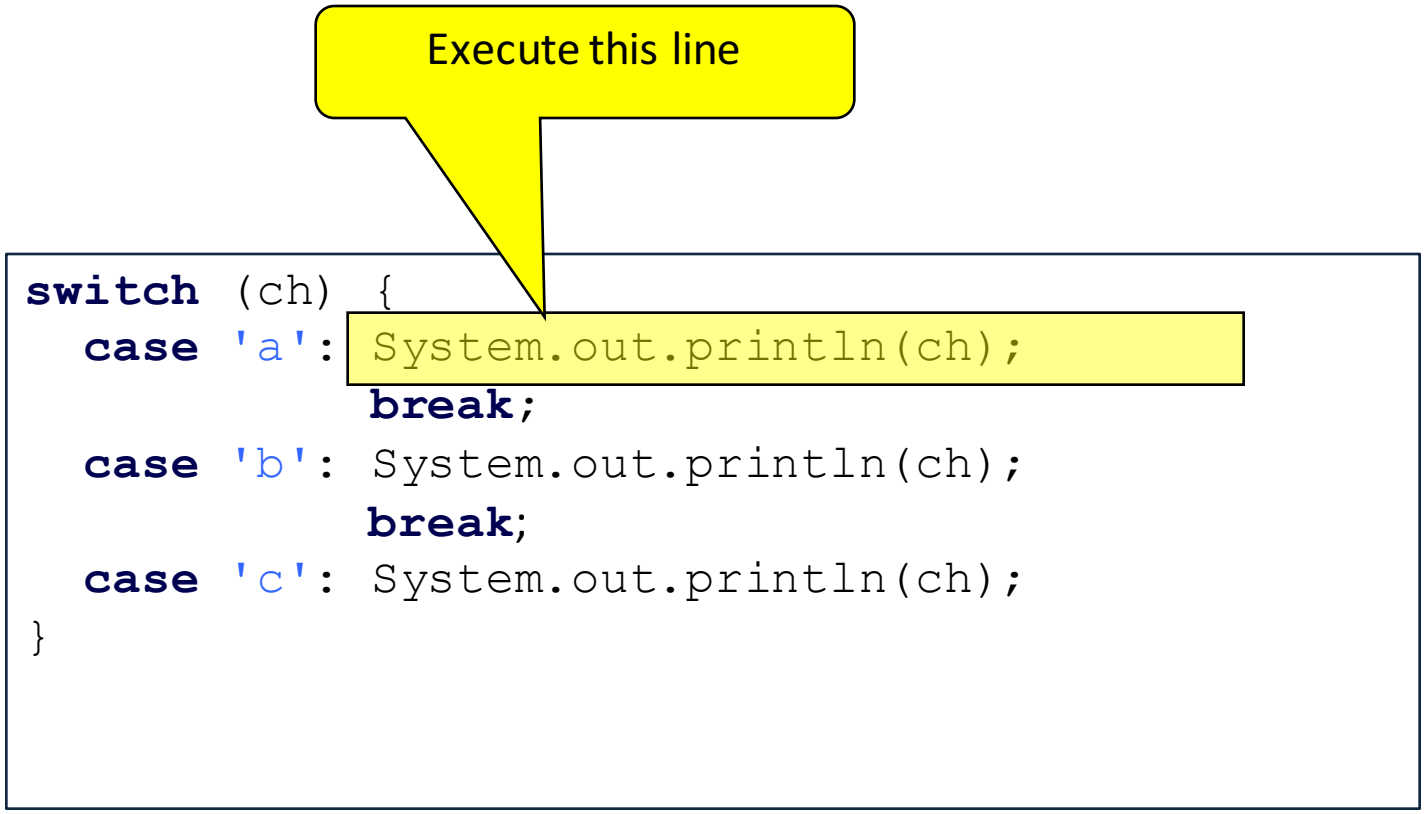
Suppose ch is 'a':

```
switch (ch) {
  case 'a': System.out.println(ch);
            break;
  case 'b': System.out.println(ch);
            break;
  case 'c': System.out.println(ch);
}
```

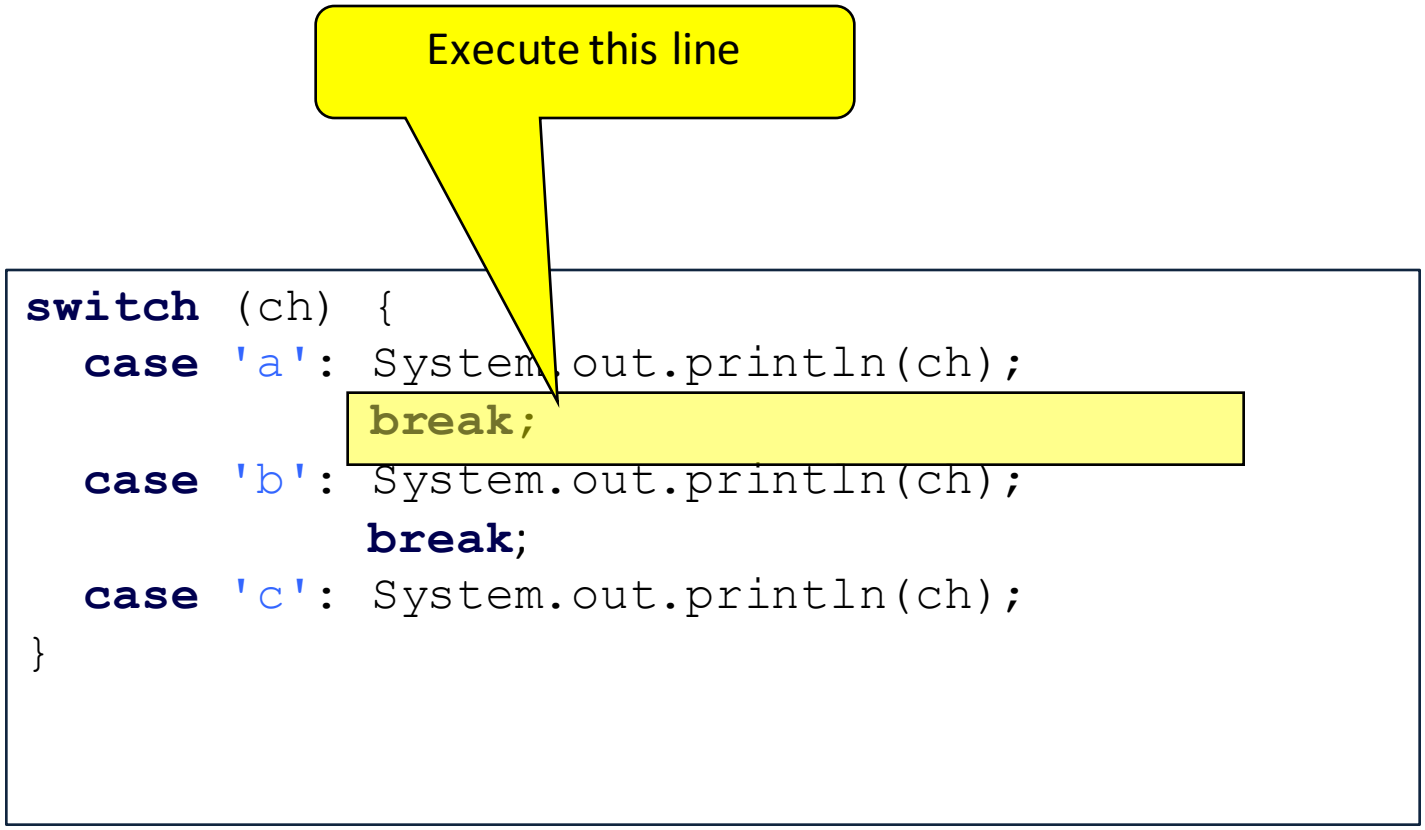
# Trace switch statement



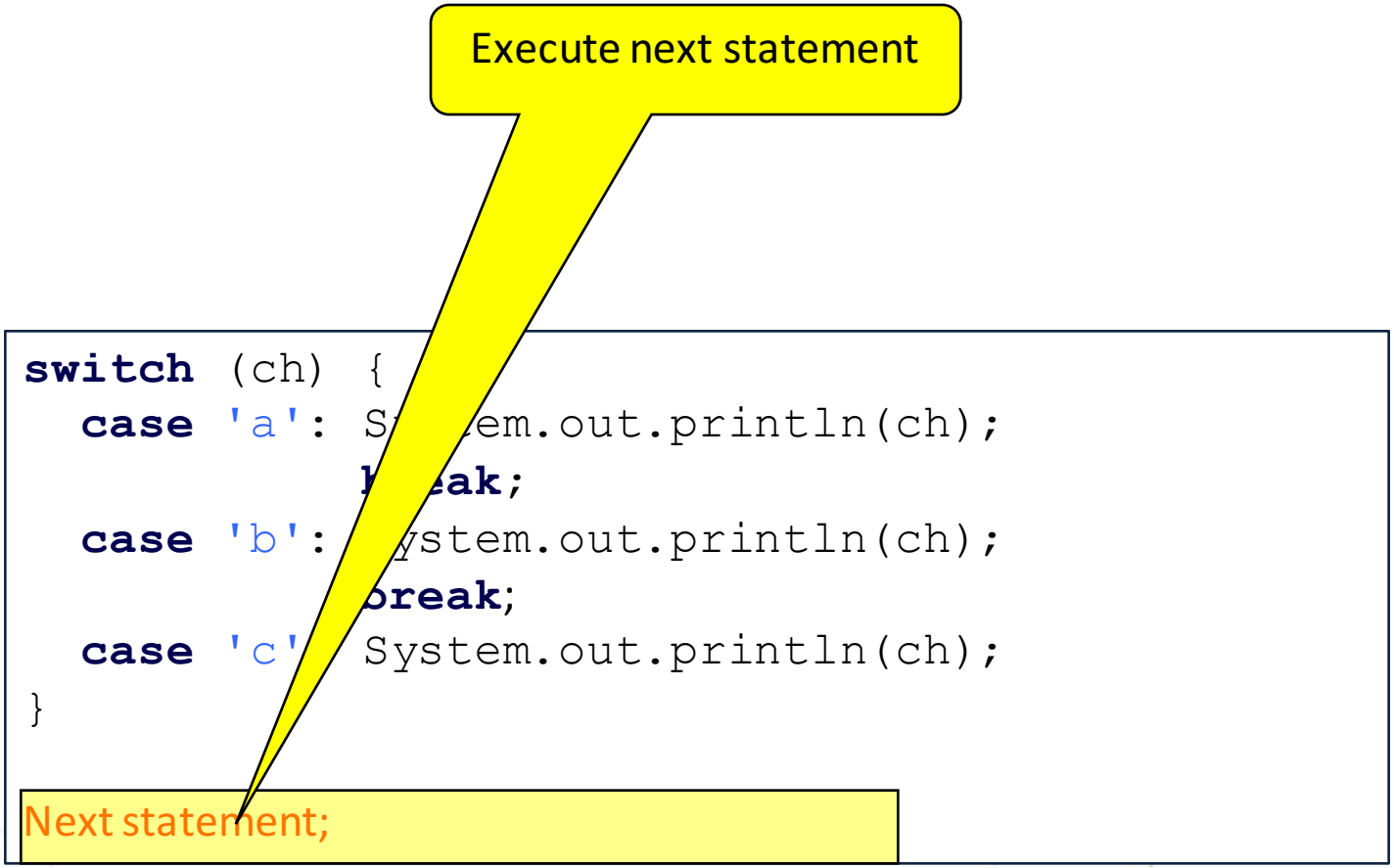
# Trace switch statement



# Trace switch statement



# Trace switch statement



# Conditional Operator

```
if (x > 0)
  y = 1;
else
  y = -1;
```

is equivalent to

```
y = (x > 0) ? 1 : -1;
```

***(boolean-expression) ? expression1 : expression2***

Ternary operator
Binary operator
Unary operator

# Conditional Operator

```
(boolean-expression) ? exp1 : exp2
```

```
if (num % 2 == 0)
  System.out.println(num + "is even");
else
  System.out.println(num + "is odd");


System.out.println(
  (num % 2 == 0)? num + "is even" :
  num + "is odd");
```

# Formatting Output

Use the printf statement.

```
System.out.printf(format, items);
```

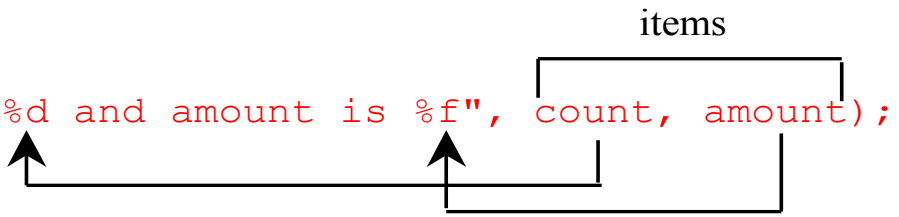
Where format is a string that may consist of substrings and format ***specifiers***. A format ***specifier*** specifies how an item should be displayed. An item may be a numeric value, character, boolean value, or a string. Each ***specifier*** begins with a percent sign.

# 1.2 Java Selections – Boolean Operators, if-else, switch statements

## Frequently-Used Specifiers

| Specifier | Output | Example |
|---|---|---|
| %b | a boolean value | true or false |
| %c | a character | 'a' |
| %d | a decimal integer | 200 |
| %f | a floating-point number | 45.460000 |
| %e | a number in standard scientific notation | 4.556000e+01 |
| %s | a string | "Java is cool" |
| %02x | a hex value | 2F |

```
int count = 5;
double amount = 45.56;
System.out.printf("count is %d and amount is %f", count, amount);
```




```
display              count is 5 and amount is 45.560000
```

# Operator Precedence

- var++, var--
- +, - (Unary plus and minus), ++var,--var
- (type) Casting
- ! (Not)
- *, /, % (Multiplication, division, and remainder)
- +, - (Binary addition and subtraction)
- <, <=, >, >= (Comparison)
- ==, !=; (Equality)
- ^ (Exclusive OR)
- && (Conditional AND) Short-circuit AND
- || (Conditional OR) Short-circuit OR
- =, +=, -=, *=, /=, %= (Assignment operator)

# Operator Precedence and Associativity

The expression in the parentheses is evaluated first. (Parentheses can be nested, in which case the expression in the inner parentheses is executed first.) When evaluating an expression without parentheses, the operators are applied according to the precedence rule and the associativity rule.

If operators with the same precedence are next to each other, their associativity determines the order of evaluation. All binary operators except assignment operators are left-associative.

# Operator Associativity

When two operators with the same precedence are evaluated, the *associativity* of the operators determines the order of evaluation. All binary operators except assignment operators are *left-associative*.

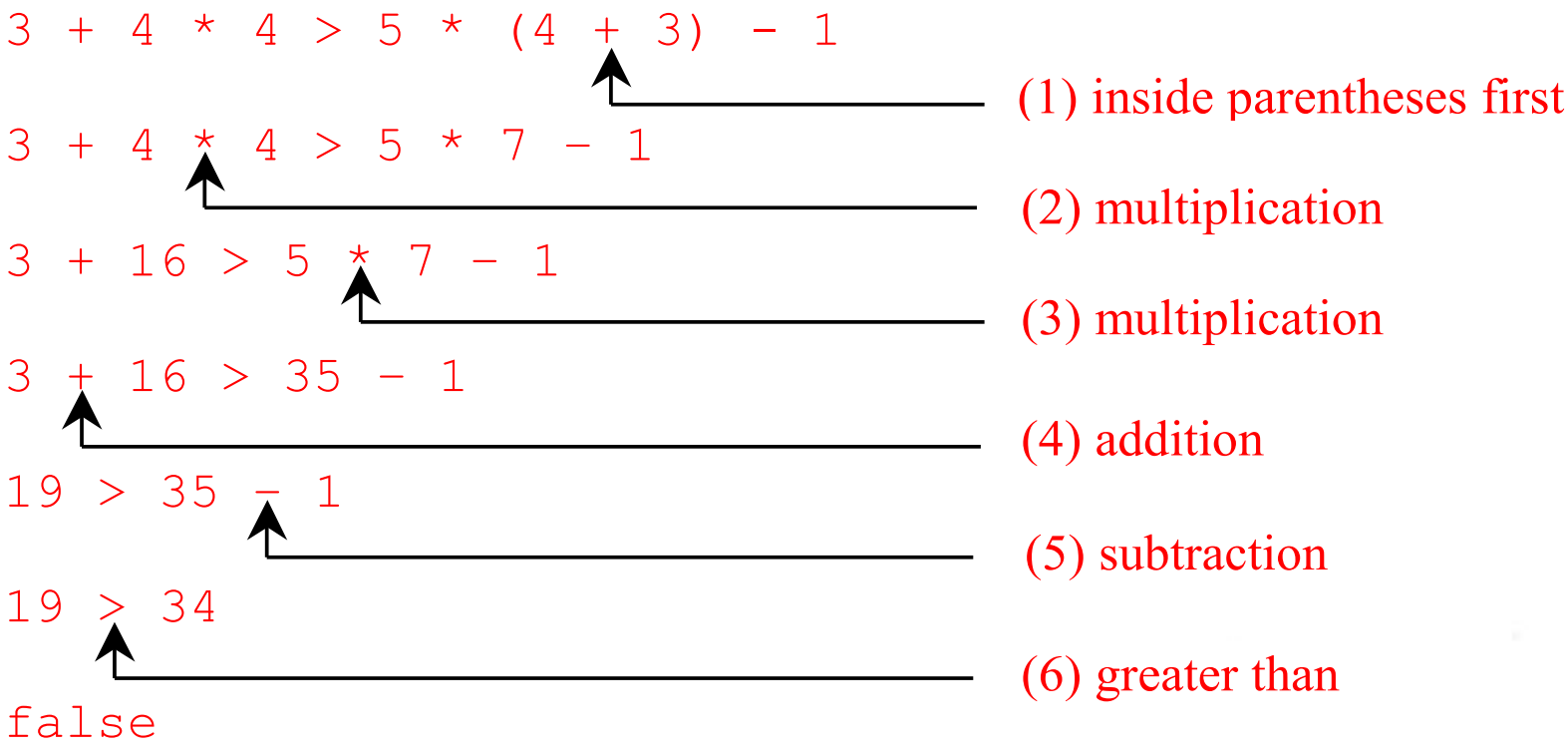
a – b + c – d is equivalent to ((a – b) + c) – d

Assignment operators are *right-associative*. Therefore, the expression

a = b += c = 5 is equivalent to a = (b += (c = 5))

# Example

Applying the operator precedence and associativity rule, the expression **3 + 4 * 4 > 5 * (4 + 3) - 1** is evaluated as follows:



# 1.3 Java Loops – for, while, … statements

Problem:

100 times

```
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
…
…
…
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
```

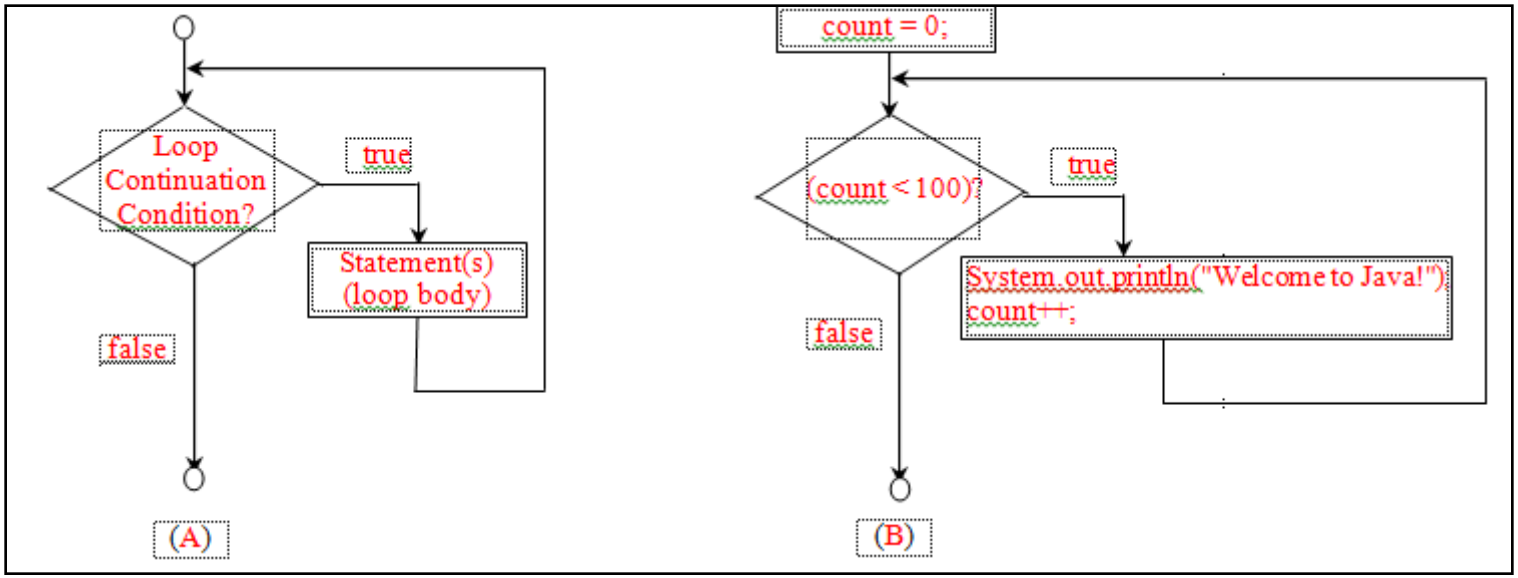
Solution:

```
int count = 0;
while (count < 100) {
  System.out.println("Welcome to Java");
  count++;
}
```

# `while` Loop Flow Chart

```
while (loop-continuation-condition) {
  // loop-body;
  Statement(s);
}
```

```
int count = 0;
while (count < 100) {
  System.out.println("Welcome  to Java!");
  count++;
}
```



# Trace while Loop

```
int count = 0;
while (count < 2) {
  System.out.println("Welcome to Java!");
  count++;
}
```

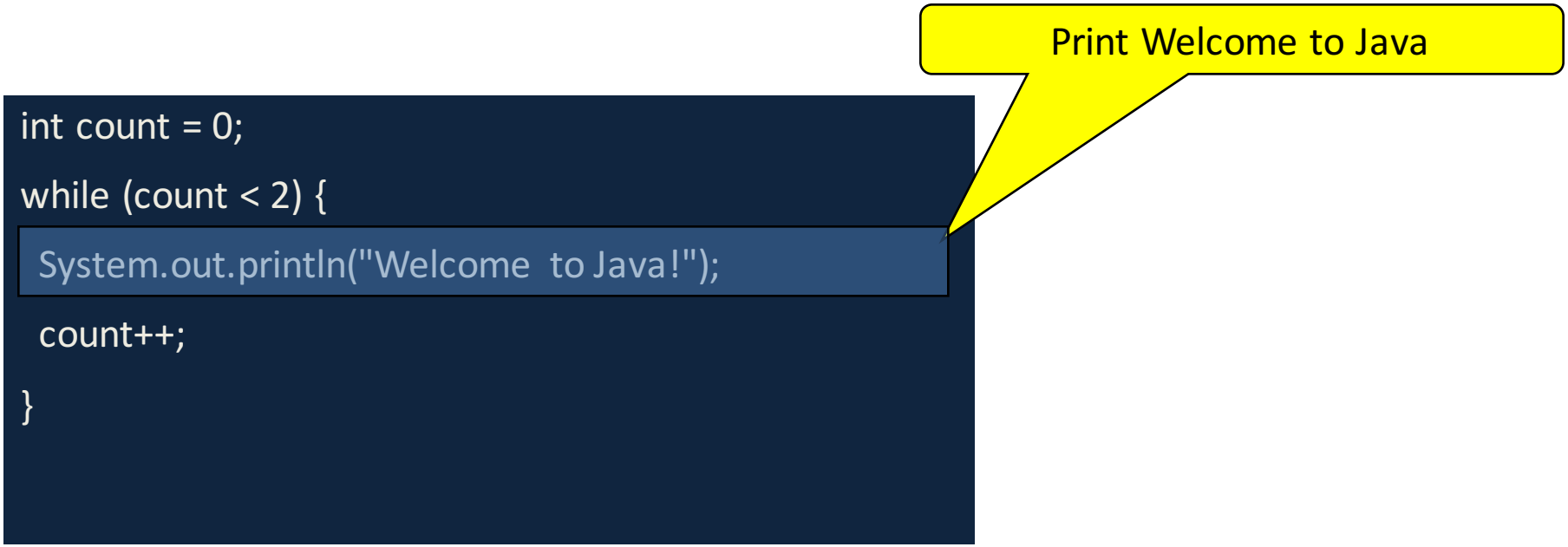
Initialize count

# Trace while Loop, cont.

(count < 2) is true

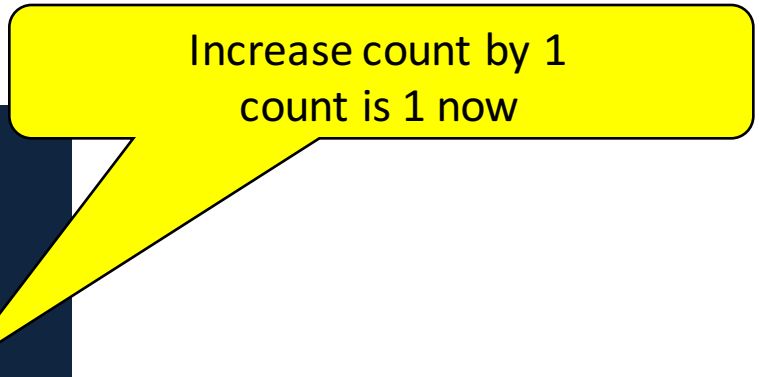
```
int count = 0;
while (count < 2) {
  System.out.println("Welcome to Java!");
  count++;
}
```

# Trace while Loop, cont.



# Trace while Loop, cont.

```
int count = 0;
while (count < 2) {
  System.out.println("Welcome to Java!");
  count++;
}
```

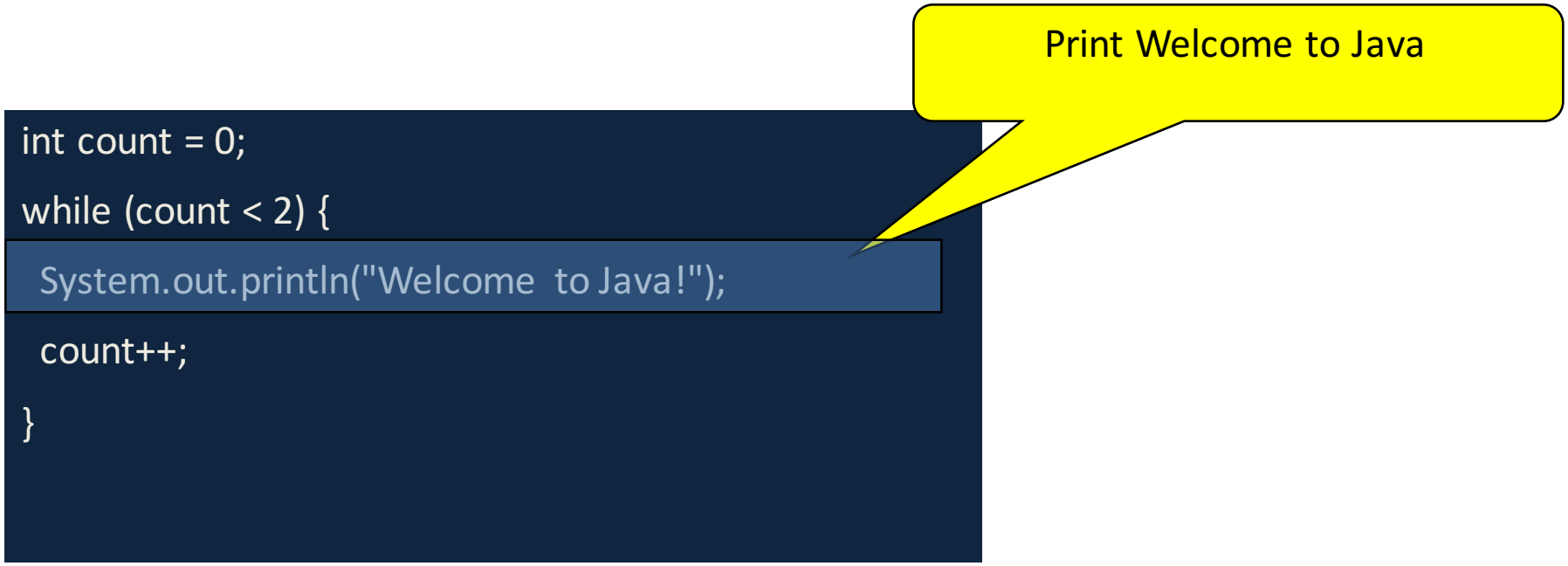


# Trace while Loop, cont.

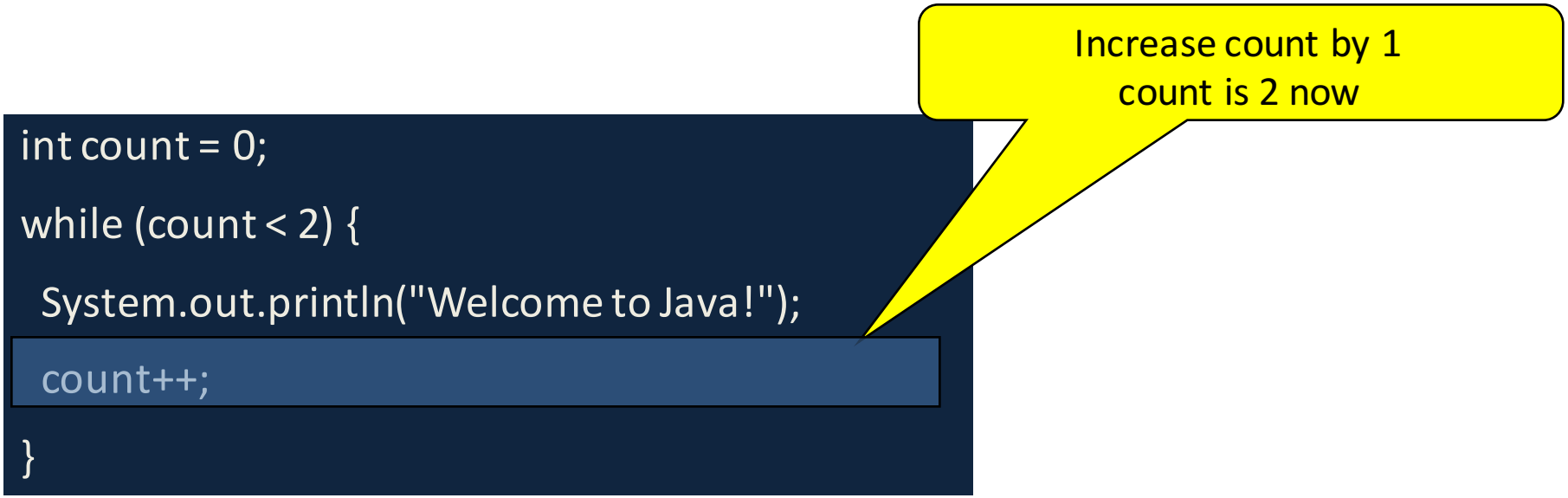
```
int count = 0;
while (count < 2) {
  System.out.println("Welcome to Java!");
  count++;
}
```

(count < 2) is still true since count is 1

# Trace while Loop, cont.



# Trace while Loop, cont.

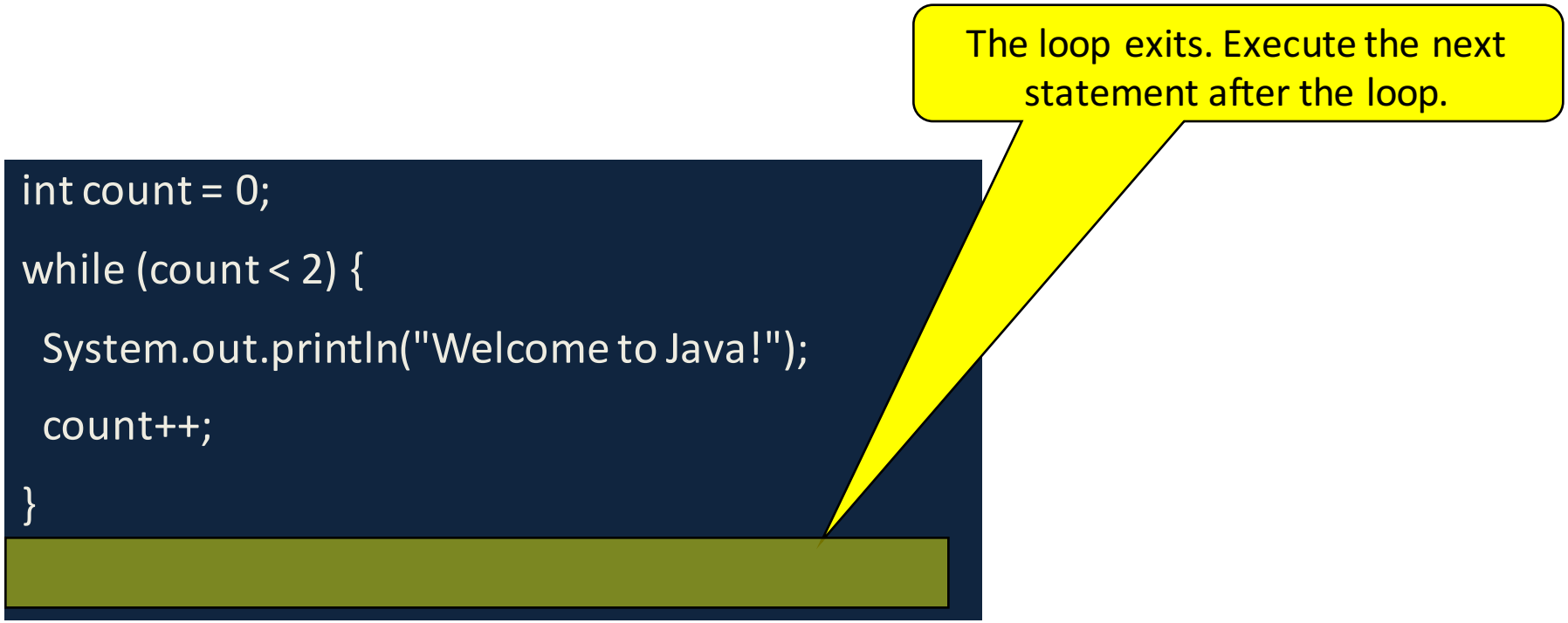


# Trace while Loop, cont.

(count < 2) is false since count is 2 now

```
int count = 0;
while (count < 2) {
  System.out.println("Welcome to Java!");
  count++;
}
```

# Trace while Loop



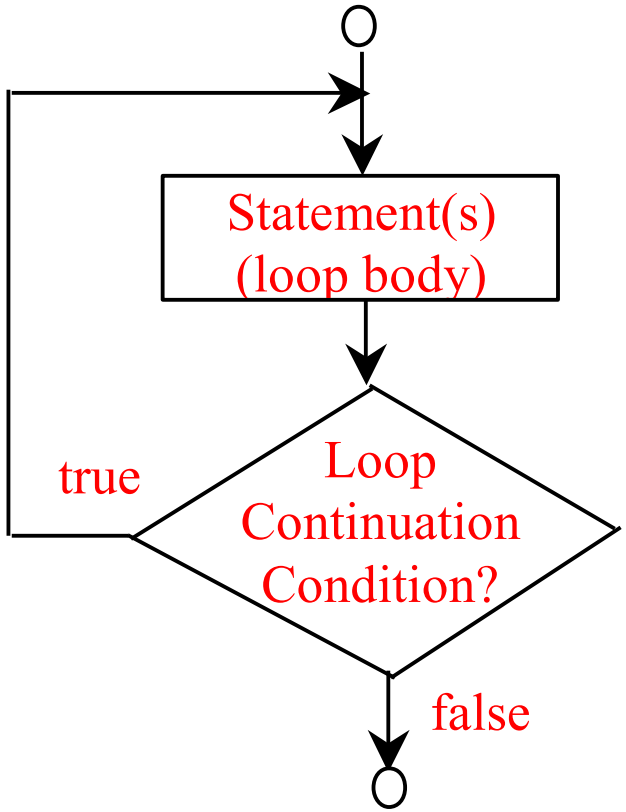
# Caution

Don't use floating-point values for equality checking in a loop control. Since floating-point values are approximations for some values, using them could result in imprecise counter values and inaccurate results. Consider the following code for computing 1 + 0.9 + 0.8 + … + 0.1:

```
double item = 1; double sum = 0;
while (item != 0) { // No guarantee item will be 0
  sum += item;
  item -= 0.1;
}
System.out.println(sum);
```

Variable item starts with 1 and is reduced by 0.1 every time the loop body is executed. The loop should terminate when item becomes 0. However, there is no guarantee that item will be exactly 0, because the floating-point arithmetic is approximated. This loop seems OK on the surface, but it is actually an infinite loop.

# do-while Loop

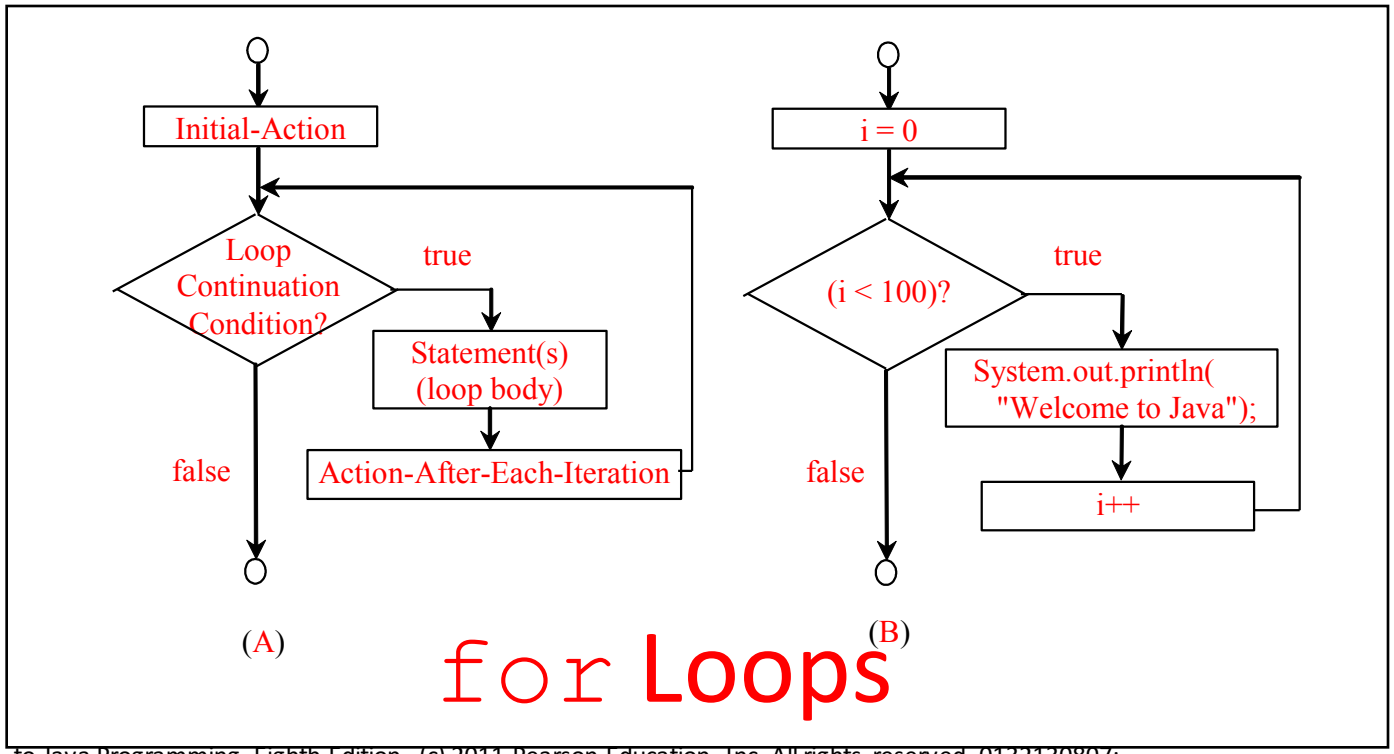



```
do {
  // Loop body;
  Statement(s);
} while (loop-continuation-condition);
```

# 1.3 Java Loops – for, while, … statements

```
for (initial-action; loop-
    continuation-condition;
    action-after-each-iteration) {
  // loop body;
  Statement(s);
}
```

```
int i;
for (i = 0; i < 100; i++) {
  System.out.println(
    "Welcome to Java!");
}
```



# Trace for Loop

```
int i;
for (i = 0; i < 2; i++) {
  System.out.println("Welcome to Java!");
}
```

Declare i

# Trace for Loop, cont.

```
int i;
for (i = 0; i < 2; i++) {
  System.out.println("Welcome to Java!");
}
```

Execute initializer
i is now 0

# Trace for Loop, cont.

```
int i;
for (i = 0; i < 2; i++) {
  System.out.println("Welcome to Java!");
}
```

(i < 2) is true
since i is 0

# Trace for Loop, cont.

```
int i;
for (i = 0; i < 2; i++) {
  System.out.println("Welcome to Java!");
}
```

# Trace for Loop, cont.

```
int i;
for (i = 0; i < 2; i++) {
  System.out.println("Welcome to Java!");
}
```

Execute adjustment statement
i now is 1

# Trace for Loop, cont.

```
int i;
for (i = 0; i < 2; i++) {
  System.out.println("Welcome to Java!");
}
```

(i < 2) is still true
since i is 1

# Trace for Loop, cont.

```
int i;
for (i = 0; i < 2; i++) {
  System.out.println("Welcome to Java!");
}
```

Print Welcome to Java 2nd time

# Trace for Loop, cont.

```
int i;
for (i = 0; i < 2; i++) {
  System.out.println("Welcome to Java!");
}
```

Execute adjustment statement
i now is 2

# Trace for Loop, cont.

```
int i;
for (i = 0; i < 2; i++) {
  System.out.println("Welcome to Java!");
}
```

(i < 2) is false
since i is 2

# Trace for Loop, cont.

```
int i;
for (i = 0; i < 2; i++) {
  System.out.println("Welcome to Java!");
}
```

# Note

The initial-action in a for loop can be a list of zero or more comma-separated expressions. The action-after-each-iteration in a for loop can be a list of zero or more comma-separated statements. Therefore, the following two for loops are correct. They are rarely used in practice, however.

```
for (int i = 1; i < 100; System.out.println(i++));


for (int i = 0, j = 0; (i + j < 10); i++, j++) {
  // Do something
}
```

# Note

If the loop-continuation-condition in a for loop is omitted, it is implicitly true. Thus the statement given below in (a), which is an infinite loop, is correct. Nevertheless, it is better to use the equivalent loop in (b) to avoid confusion:

```
for ( ; ; ) {
  // Do something
}
```

(a)

Equivalent

```
while (true) {
  // Do something
}
```

(b)

# Caution

Adding a semicolon at the end of the for clause before the loop body is a common mistake, as shown below:

Logic Error

```
for (int i=0; i<10; i++);
{
  System.out.println("i is " + i);
}
```

# 1.3 Java Loops – for, while, … statements

Similarly, the following loop is also wrong:

```
int i=0;
while (i < 10);
{
  System.out.println("i is " + i);
  i++;
}
```

Logic Error

Caution, cont.

In the case of the do loop, the following semicolon is needed to end the loop.

```
int i=0;
do {
  System.out.println("i is " + i);
  i++;
} while (i<10);
```

Correct

# Which Loop to Use?

The three forms of loop statements, **while**, **do-while**, and **for**, are expressively equivalent; that is, you can write a loop in any of these three forms. For example, a while loop in (a) in the following figure can always be converted into the following for loop in (b):

```
while (loop-continuation-condition) {
  // Loop body
}
```

(a)

Equivalent

```
for ( ; loop-continuation-condition; )
  // Loop body
}
```

(b)

A for loop in (a) in the following figure can generally be converted into the following while loop in (b) except in certain special:

```
for (initial-action;
     loop-continuation-condition;
     action-after-each-iteration) {
  // Loop body;
}
```

(a)

Equivalent

```
initial-action;
while (loop-continuation-condition) {
  // Loop body;
  action-after-each-iteration;
}
```

(b)

# Recommendations

Use the one that is most intuitive and comfortable for you. In general, a for loop may be used if the number of repetitions is known, as, for example, when you need to print a message 100 times. A while loop may be used if the number of repetitions is not known, as in the case of reading the numbers until the input is 0. A do-while loop can be used to replace a while loop if the loop body has to be executed before testing the continuation condition.

**ASSIGNMENT 02 - Problem: Write a program that uses nested for loops to print a multiplication table – nested loops.**

# Problem

```
int sum = 0;
for (int i = 1; i <= 10; i++)
  sum += i;
System.out.println("Sum from 1 to 10 is " + sum);

sum = 0;
for (int i = 20; i <= 30; i++)
  sum += i;
System.out.println("Sum from 20 to 30 is " + sum);

sum = 0;
for (int i = 35; i <= 45; i++)
  sum += i;
System.out.println("Sum from 35 to 45 is " + sum);
```

# Problem

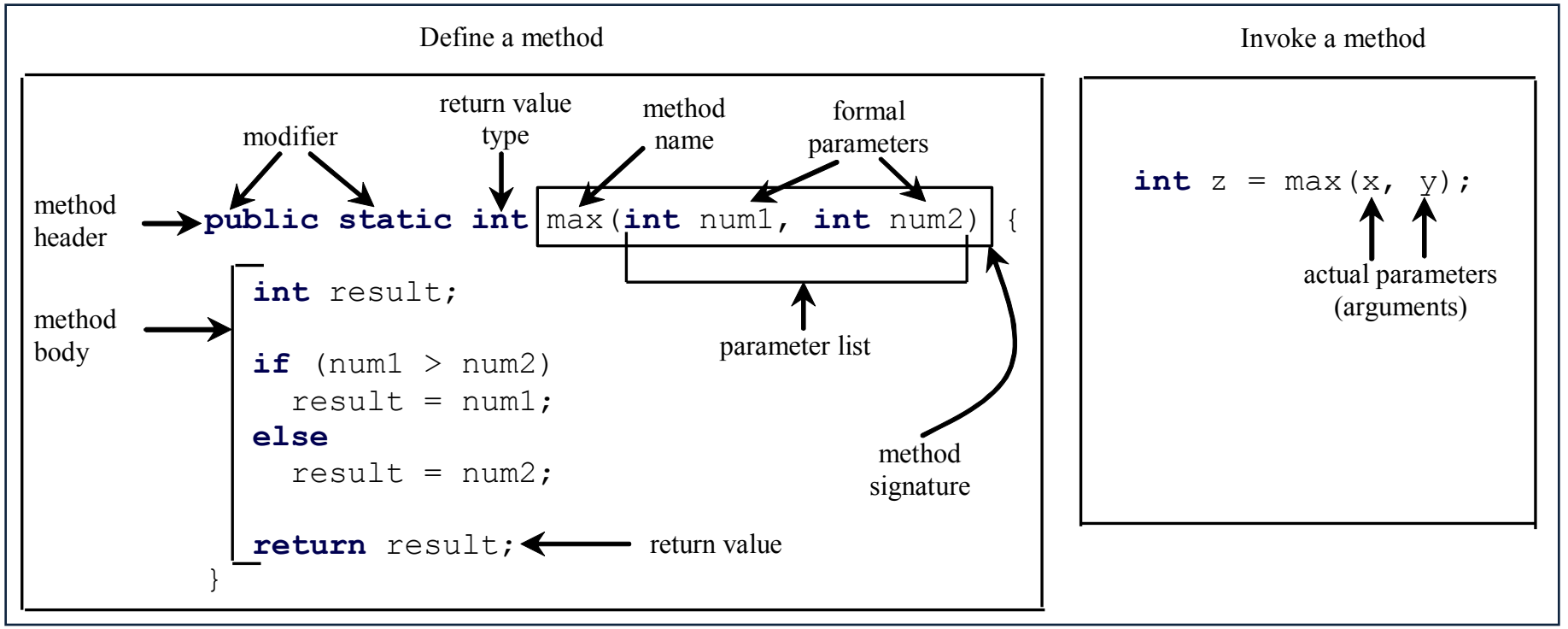
```
int sum = 0;
for (int i = 1; i <= 10; i++)
  sum += i;
System.out.println("Sum from 1 to 10 is " + sum);

sum = 0;
for (int i = 20; i <= 30; i++)
  sum += i;
System.out.println("Sum from 20 to 30 is " + sum);

sum = 0;
for (int i = 35; i <= 45; i++)
  sum += i;
System.out.println("Sum from 35 to 45 is " + sum);
```

## Solution

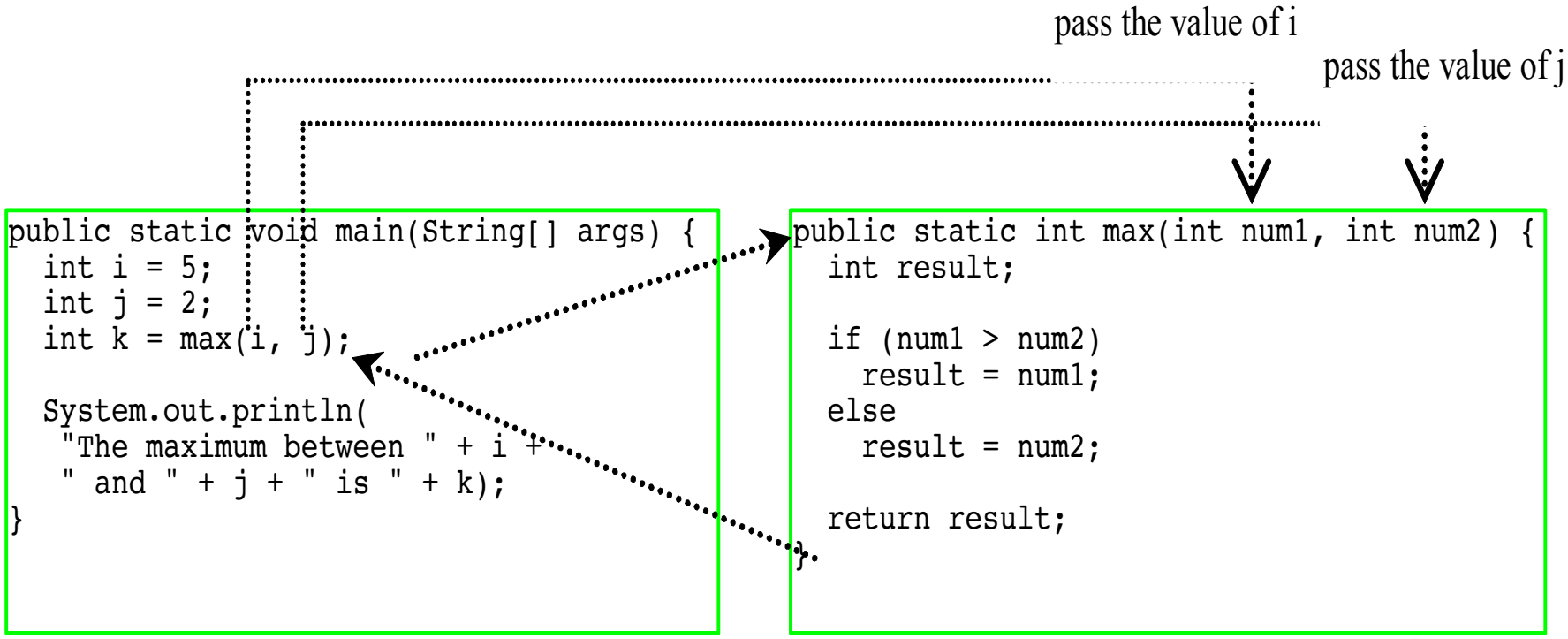
```
public static int sum(int i1, int i2) {
  int sum = 0;
  for (int i = i1; i <= i2; i++)
    sum += i;
  return sum;
}

public static void main(String[] args) {
  System.out.println("Sum from 1 to 10 is " + sum(1, 10));
  System.out.println("Sum from 20 to 30 is " + sum(20, 30));
  System.out.println("Sum from 35 to 45 is " + sum(35, 45));
}
```

# Defining Methods

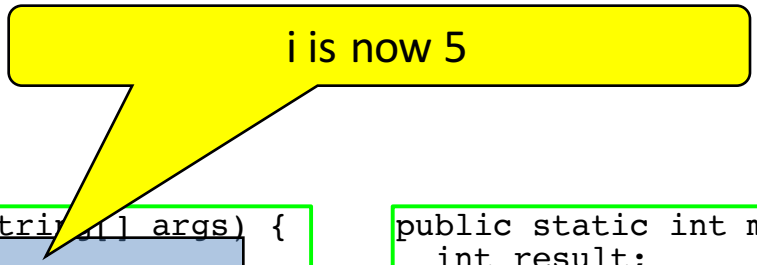
A method is a collection of statements that are grouped together to perform an operation.



# Calling Methods



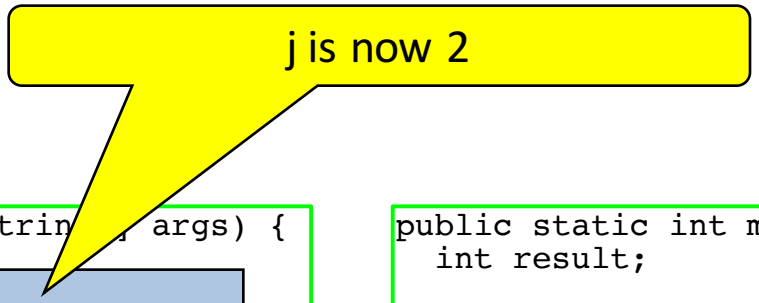
# Trace Method Invocation




```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

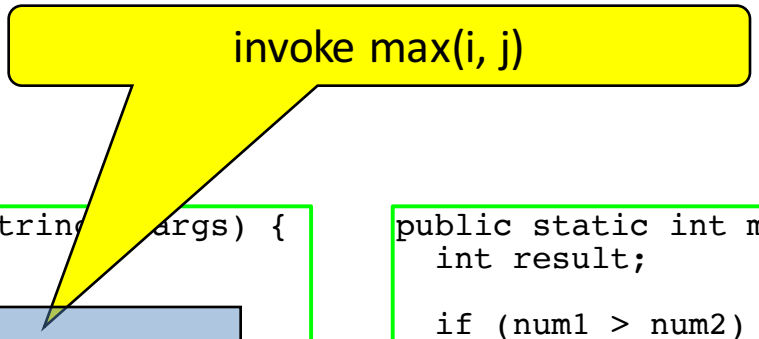
# Trace Method Invocation




```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

# Trace Method Invocation




```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

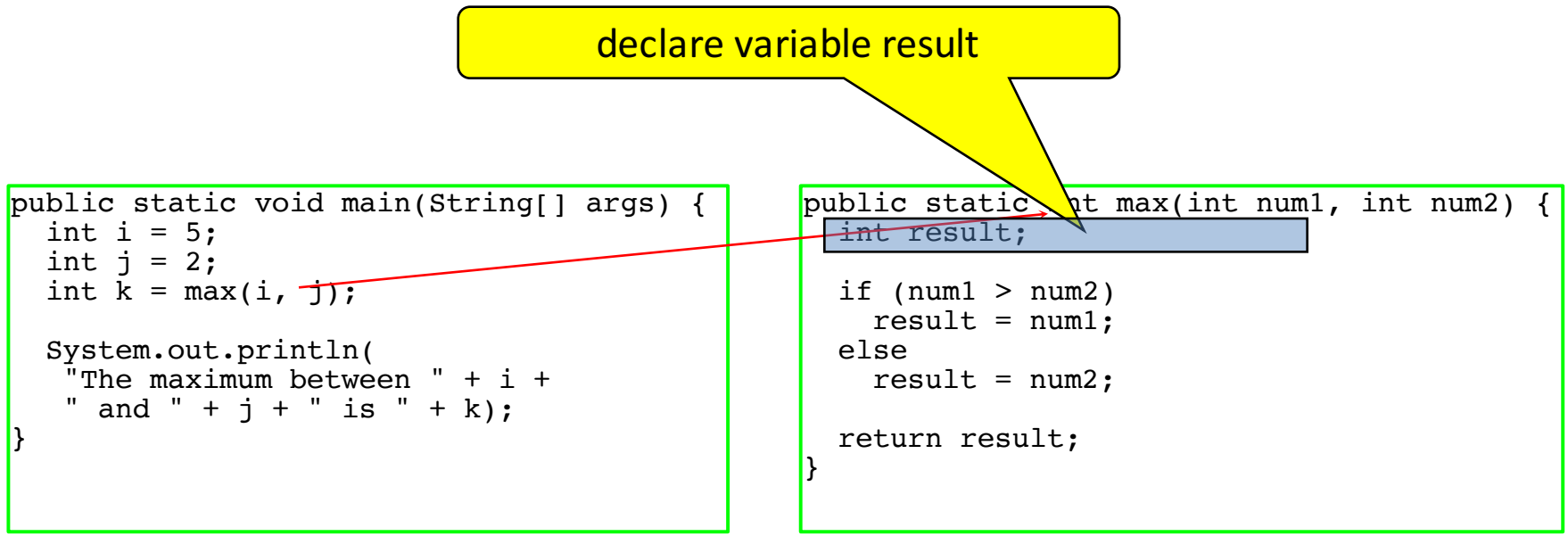
# Trace Method Invocation

**invoke max(i, j)**
**Pass the value of i to num1**
**Pass the value of j to num2**

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

# Trace Method Invocation



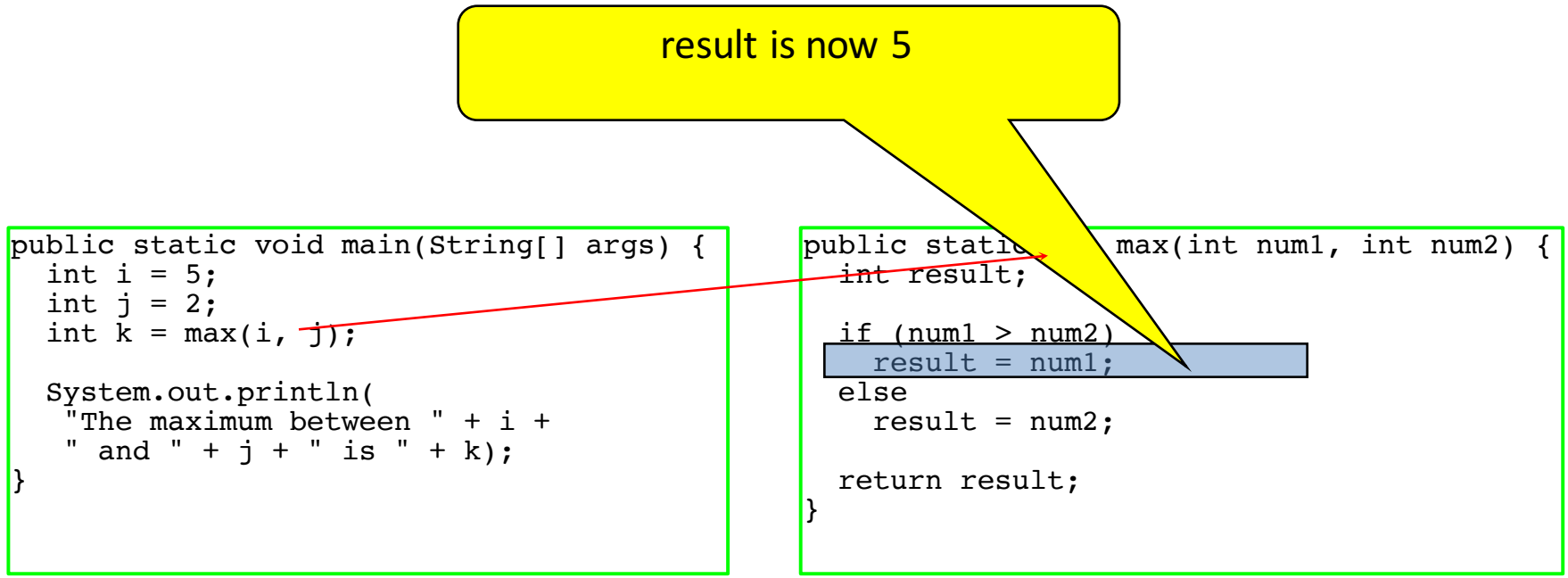
# Trace Method Invocation

(num1 > num2) is true since num1 is 5 and num2 is 2

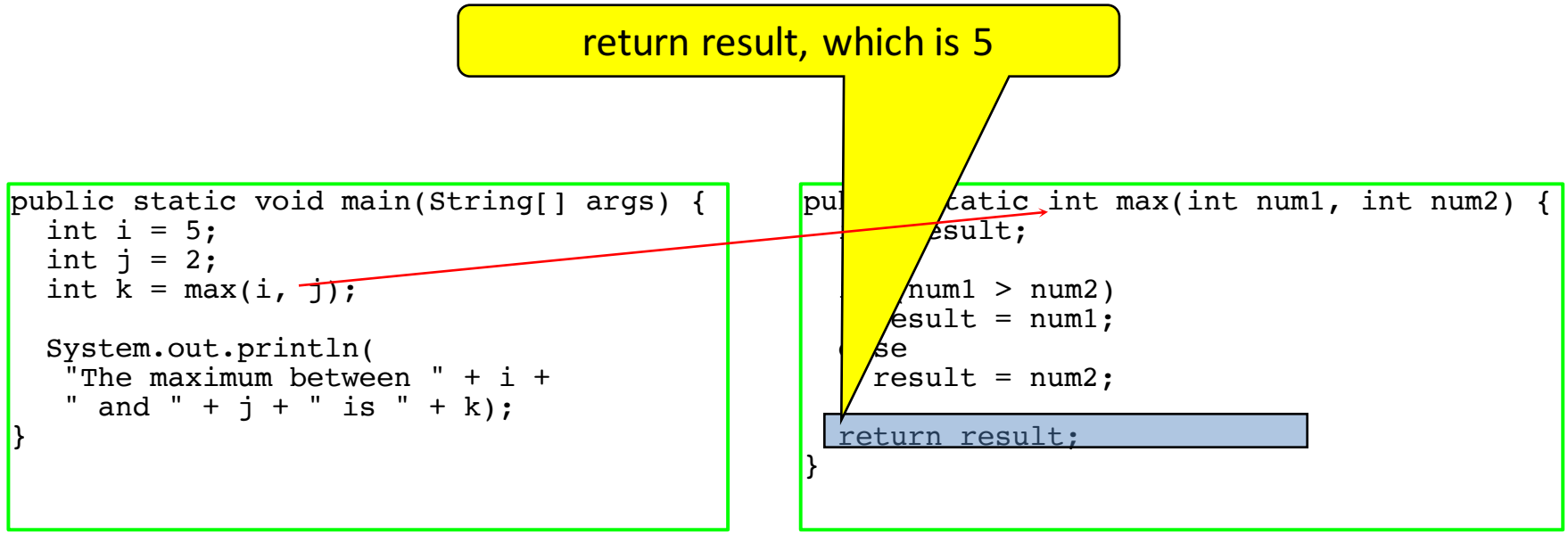
```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

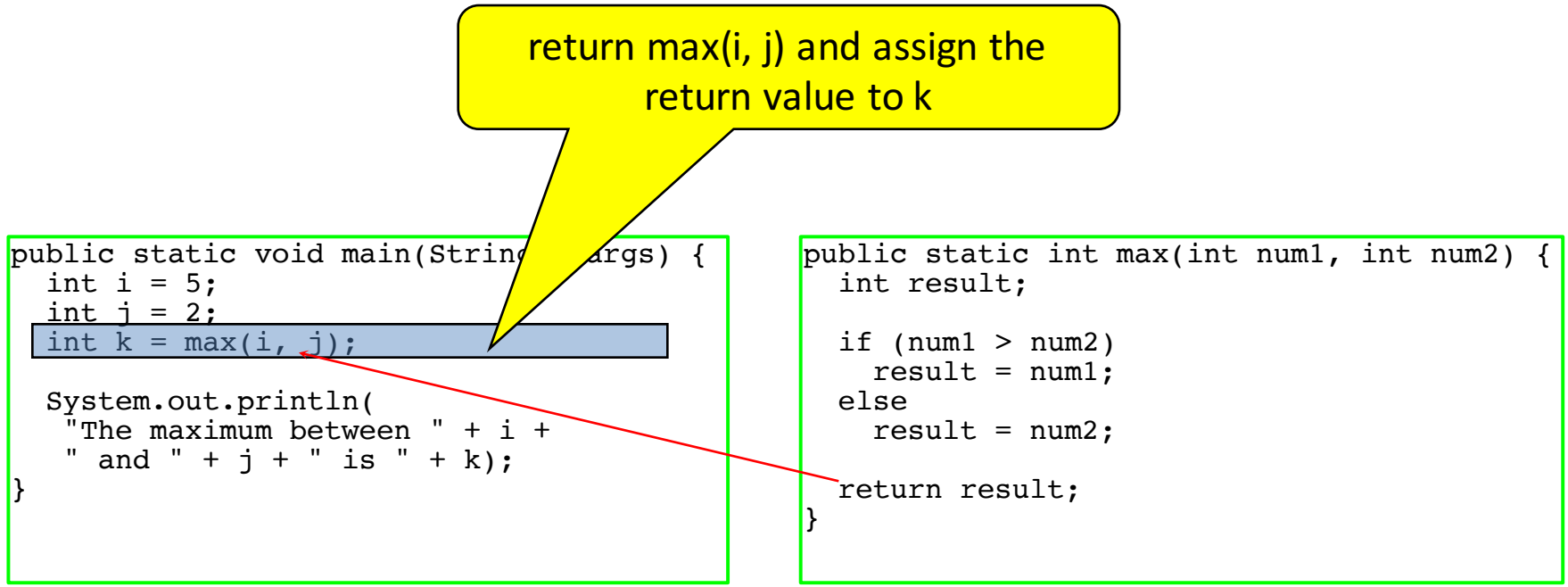
# Trace Method Invocation



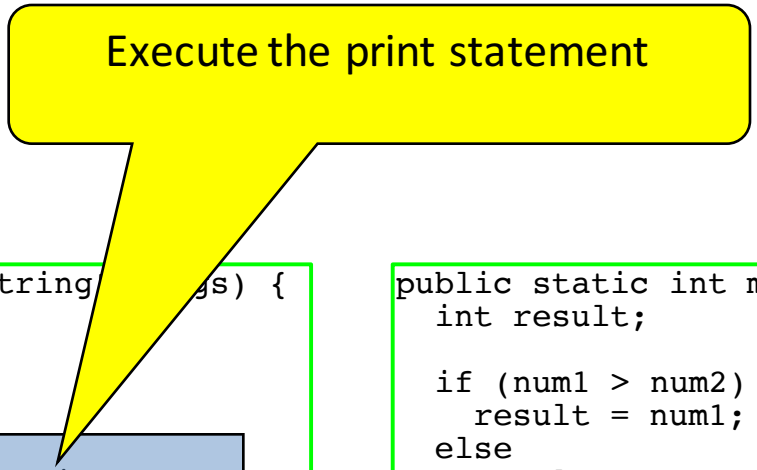
# Trace Method Invocation



# Trace Method Invocation



# Trace Method Invocation




```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

# CAUTION

A return statement is required for a value-returning method. The method shown below in (a) is logically correct, but it has a compilation error because the Java compiler thinks it possible that this method does not return any value.

```
public static int sign(int n) {
  if (n > 0)
    return 1;
  else if (n == 0)
    return 0;
  else if (n < 0)
    return -1;
}
```

(a)

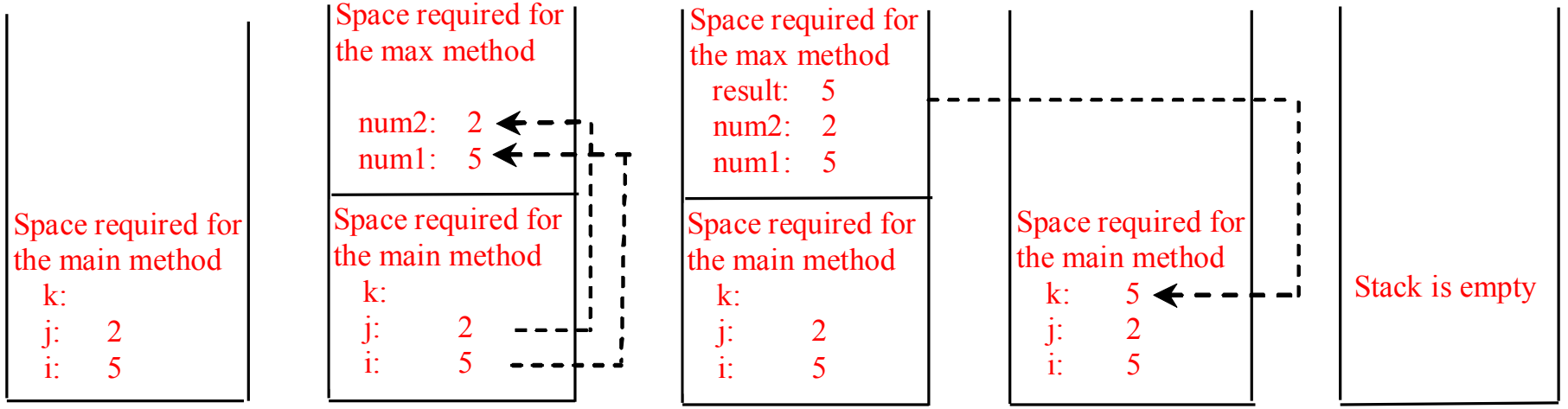
Should be

```
public static int sign(int n) {
  if (n > 0)
    return 1;
  else if (n == 0)
    return 0;
  else
    return -1;
}
```

(b)

To fix this problem, delete *if (n < 0)* in (a), so that the compiler will see a return statement to be reached regardless of how the if statement is evaluated.

# Call Stacks




(a) The main method is invoked.

(b) The max method is invoked.

(c) The max method is being executed.

(d) The max method is finished and the return value is sent to k.

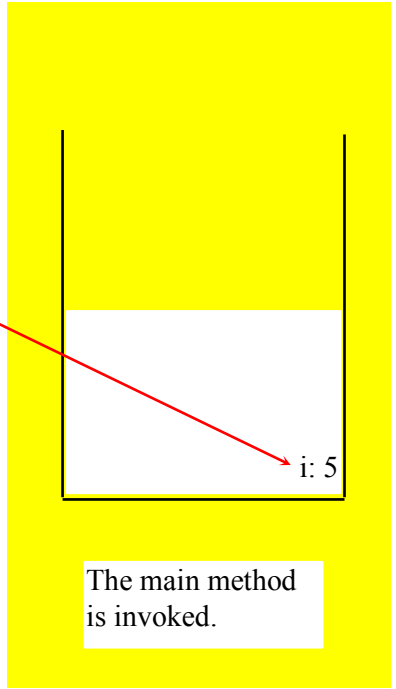
(e) The main method is finished.

# Trace Call Stack

**i is declared and initialized**

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
    "The maximum between " + i +
    " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

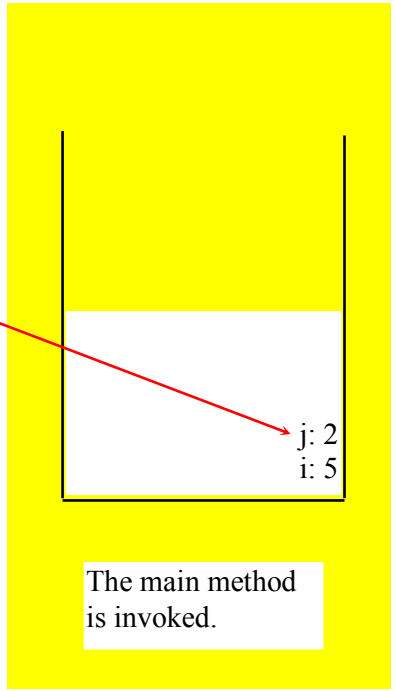


# Trace Call Stack

**j is declared and initialized**

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
    "The maximum between " + i +
    " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

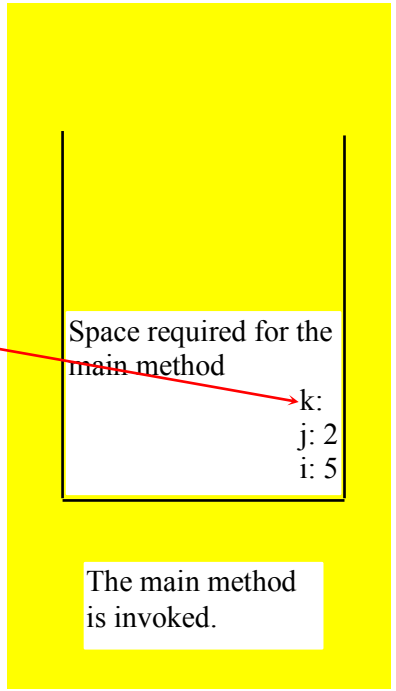


# Trace Call Stack

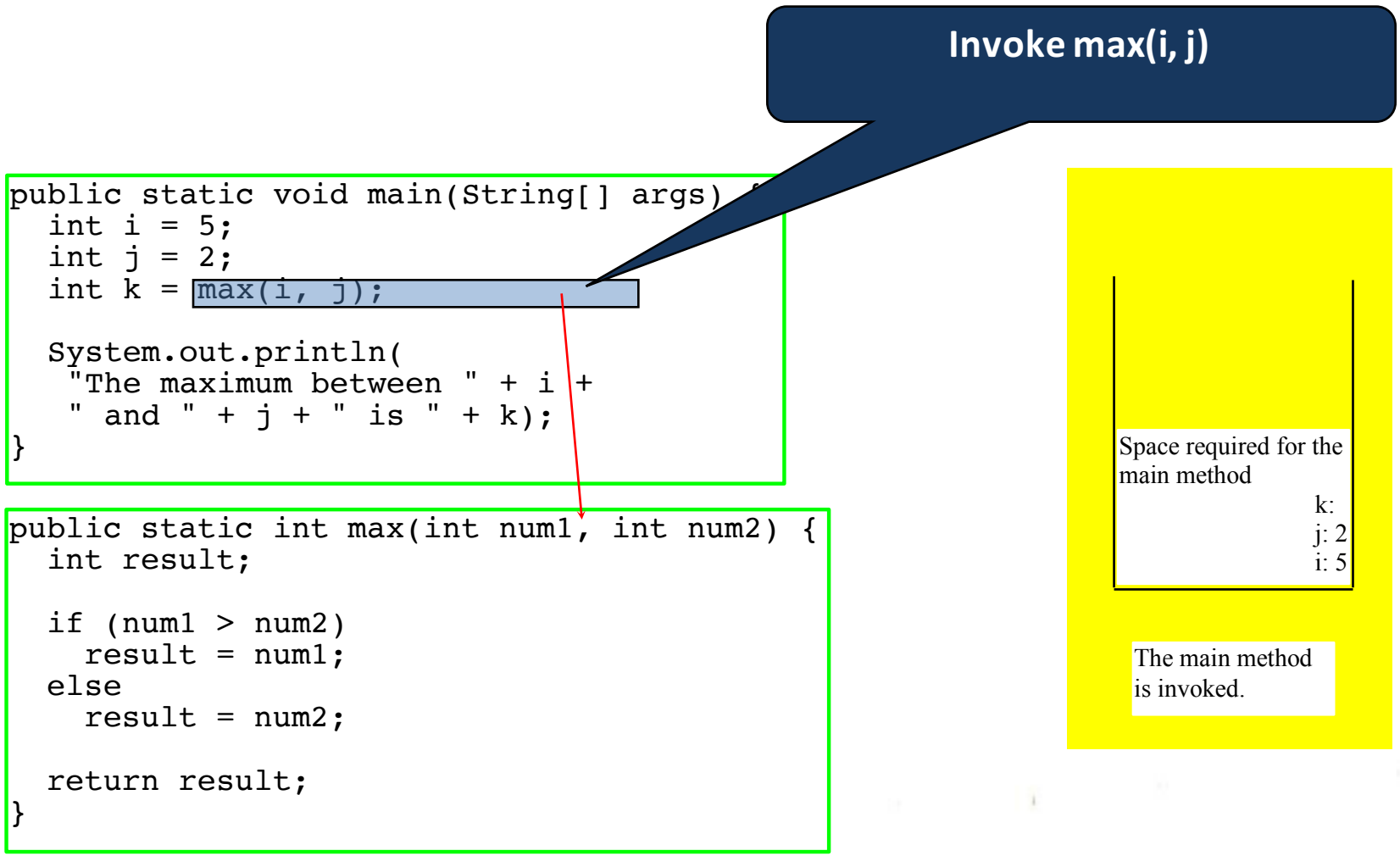
**Declare k**

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
    "The maximum between " + i +
    " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```



# Trace Call Stack

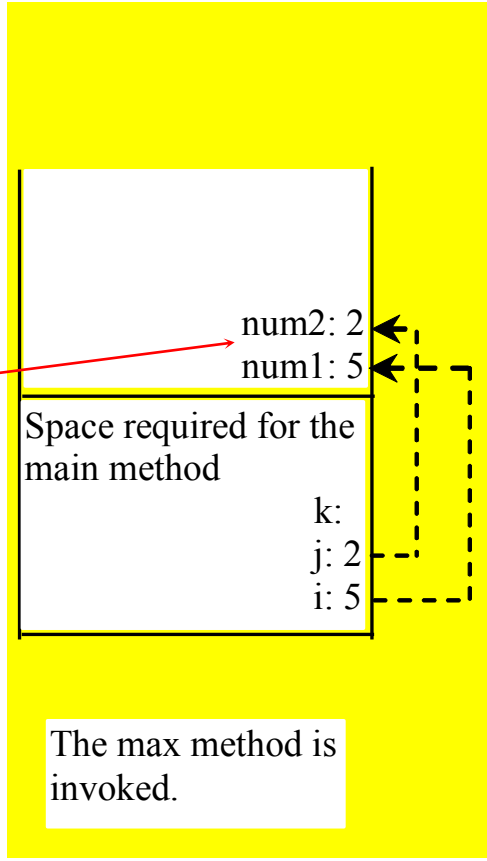


# Trace Call Stack

pass the values of i and j to num1 and num2

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
    "The maximum between " + i +
    " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

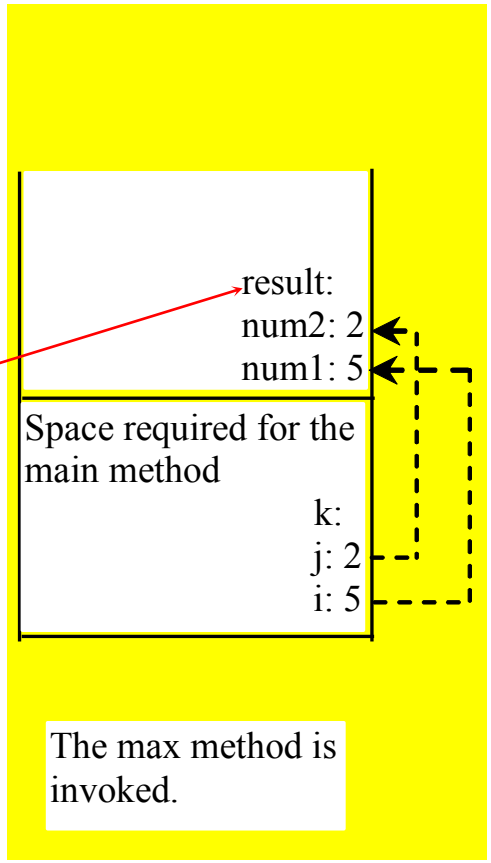


# Trace Call Stack

pass the values of i and j to num1 and num2

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
    "The maximum between " + i +
    " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

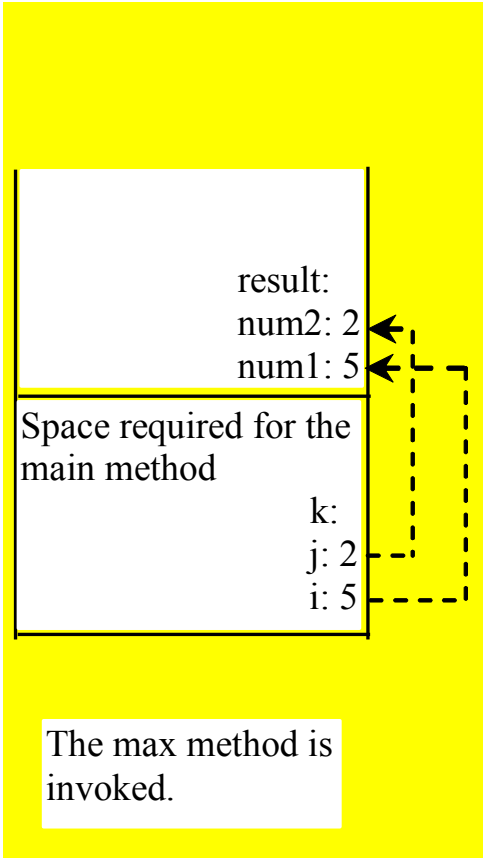


# Trace Call Stack

(num1 > num2) is true

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

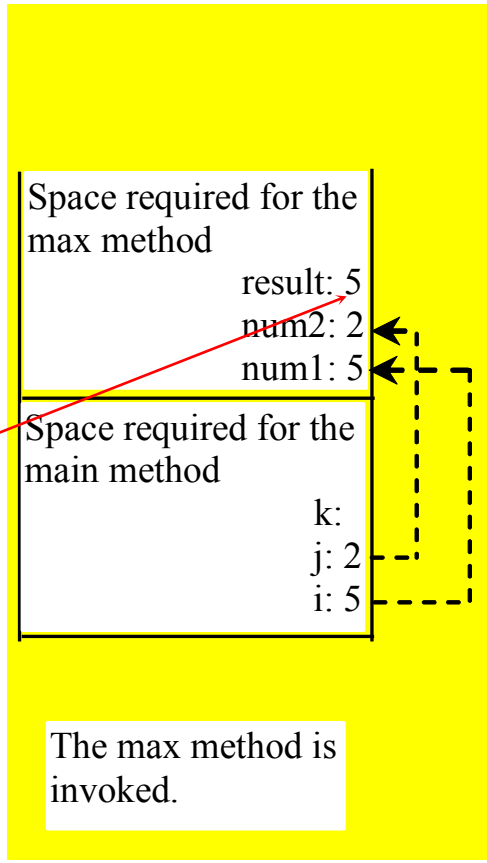


# Trace Call Stack

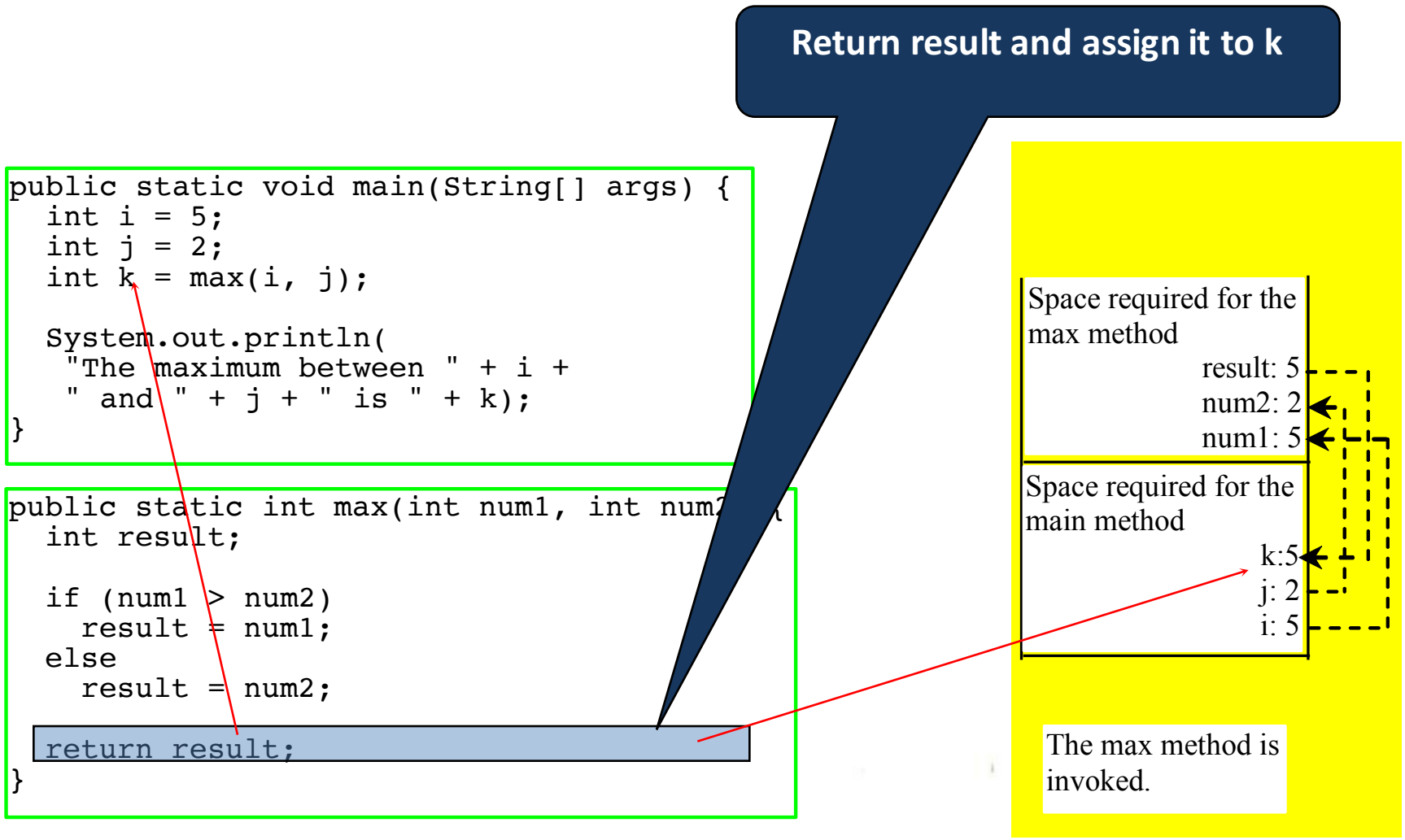
**Assign num1 to result**

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
    "The maximum between " + i +
    " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2)
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```



# Trace Call Stack

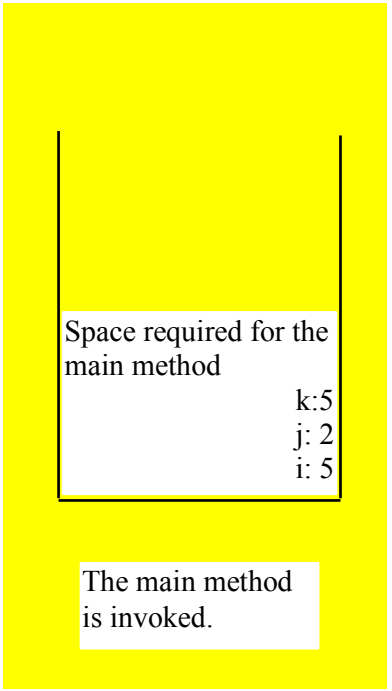


# Trace Call Stack

**Execute print statement**

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```



# Passing Parameters

```
public static void nPrintln(String message, int n) {
  for (int i = 0; i < n; i++)
    System.out.println(message);
}
```

Suppose you invoke the method using

nPrintln("Welcome to Java", 5);

What is the output?

Suppose you invoke the method using

nPrintln("Computer Science", 15);

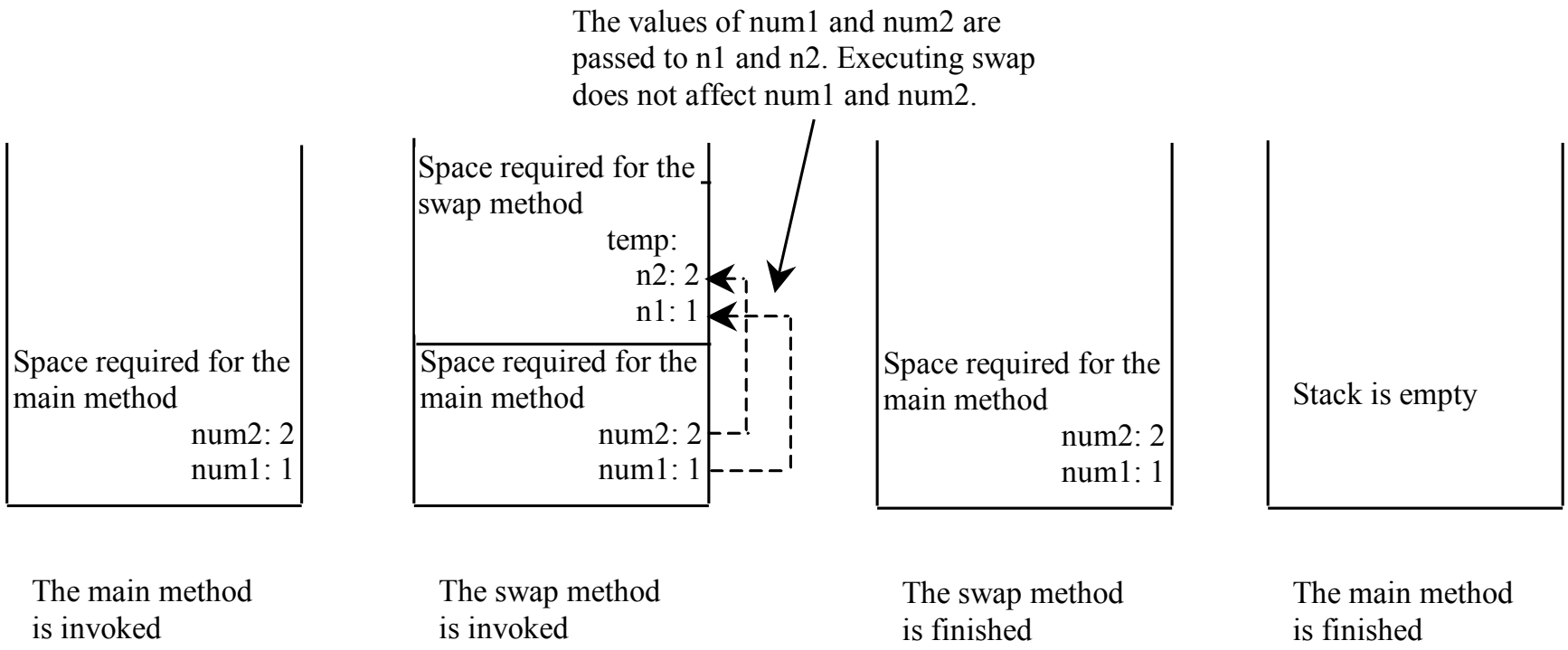
What is the output?

# 1.4 Java Methods – defining, signature, formal / actual parameters

Passing parameters by value

```
public class TestPassByValue {
  /** Main method */
  public static void main(String[] args) {
    // Declare and initialize variables
    int num1 = 1;
    int num2 = 2;

    System.out.println("Before invoking the swap method, num1 is " +
      num1 + " and num2 is " + num2);

    // Invoke the swap method to attempt to swap two variables
    swap(num1, num2);

    System.out.println("After invoking the swap method, num1 is " +
      num1 + " and num2 is " + num2);
  }

  /** Swap two variables */
  public static void swap(int n1, int n2) {
    System.out.println("\tInside the swap method");
    System.out.println("\t\tBefore swapping n1 is " + n1
      + " n2 is " + n2);

    // Swap n1 with n2
    int temp = n1;
    n1 = n2;
    n2 = temp;

    System.out.println("\t\tAfter swapping n1 is " + n1
      + " n2 is " + n2);
  }
}
```

# Call Stacks – Passing parameters by value



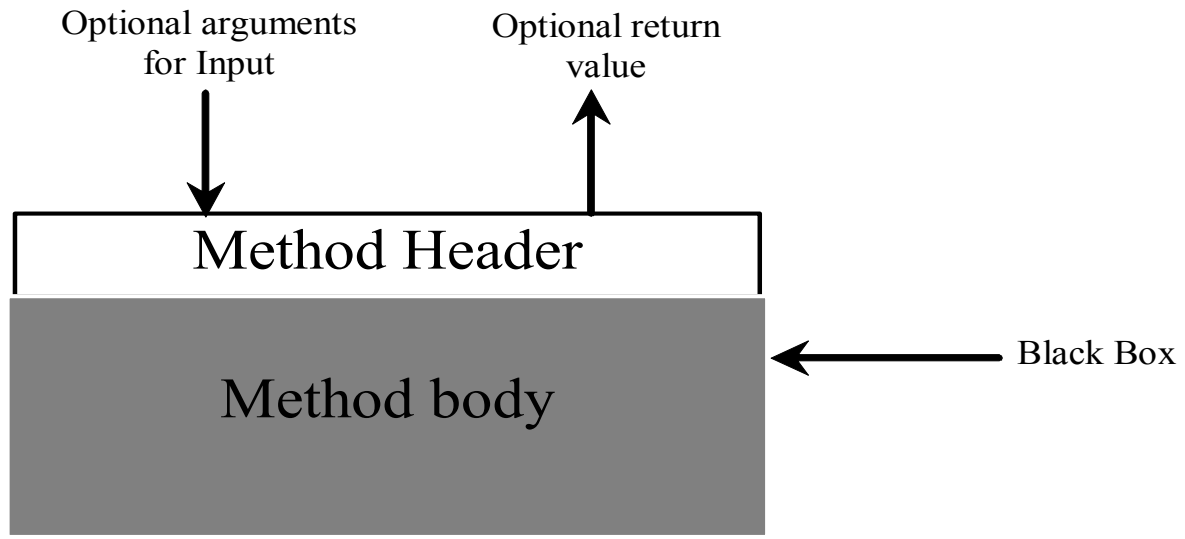
# 1.4 Java Methods – defining, signature, formal / actual parameters

## Overloading methods - Ambiguous Invocation

```
public class AmbiguousOverloading {
  public static void main(String[] args) {
    System.out.println(max(1, 2));
  }

  public static double max(int num1, double num2) {
    if (num1 > num2)
      return num1;
    else
      return num2;
  }

  public static double max(double num1, int num2) {
    if (num1 > num2)
      return num1;
    else
      return num2;
  }
}
```

# Method Abstraction

A programmer can think of the method body as a black box that contains the detailed implementation for the method.




Benefits of using methods:

- Write a method once and reuse it anywhere.
- Information hiding. Hide the implementation from the user.
- Reduce complexity.

# The `Math` Class

**Case study:**

- Class constants:
  - `PI`
  - `E`
- Class methods:
  - Trigonometric Methods
  - Exponent Methods
  - Rounding Methods
  - min, max, abs, and random Methods

# Trigonometric Methods

- `sin(double a)`
- `cos(double a)`
- `tan(double a)`
- `acos(double a)`
- `asin(double a)`
- `atan(double a)`

```
Examples:

Math.sin(0) returns 0.0
Math.sin(Math.PI / 6) returns 0.5
Math.sin(Math.PI / 2) returns 1.0
Math.cos(0) returns 1.0
Math.cos(Math.PI / 6) returns
  0.866
Math.cos(Math.PI / 2) returns 0
```

# Exponent Methods

- `exp(double a)`
  Returns `e` raised to the power of `a`.
- `log(double a)`
  Returns the natural logarithm of `a`.
- `log10(double a)`
  Returns the 10-based logarithm of `a`.
- `pow(double a, double b)`
  Returns `a` raised to the power of `b`.
- `sqrt(double a)`
  Returns the square root of `a`.

```
Examples:

Math.exp(1) returns 2.71
Math.log(2.71) returns 1.0
Math.pow(2, 3) returns 8.0
Math.pow(3, 2) returns 9.0
Math.pow(3.5, 2.5) returns
   22.91765
Math.sqrt(4) returns 2.0
Math.sqrt(10.5) returns 3.24
```

# min, max, and abs

- `max(a, b)` and `min(a, b)`

  Returns the maximum or minimum of two parameters.

- `abs(a)`

  Returns the absolute value of the parameter.

- `random()`

  Returns a random `double` value in the range [0.0, 1.0).

```
Examples:

Math.max(2, 3) returns 3
Math.max(2.5, 3) returns
   3.0
Math.min(2.5, 3.6) returns
   2.5
Math.abs(-2) returns 2
Math.abs(-2.1) returns 2.1
```

# Section Conclusion

Fact: **Java is natural**

In few **samples** it is simple to remember: selections, loops, methods – it is like in C/C++ syntax.


Java OOP

# Java OOP & JSE

# 2. Summary of Java SE - OOP

**Analogy with C++ and it will be reloaded in next lecture (C02)**

What is a class?

What is a package of classes in Java?

What is an object / instance?

How many bytes has an object inside JVM?

Why do we need clone, equals and hash methods?

Demo & memory model for the ***Certificate*** Java class

# 2. Summary of Java SE - OOP

***Polymorphism*** – "the ability to have multiple forms" is obtaining through:

- The overriding (not overloading; overloading is a form of polymorphism) of the methods from a Java class
- "Pure" form of polymorphism has to have the following conditions:
  - Inheritance mechanism – "***extends*** is the key word"
  - Virtual Methods – "in Java by default"
  - Overriding the virtual methods
  - Using the pointer / references to the objects – "in Java by default"

***Interface*** – "contract between objects of the class that implements the interface and another objects of another classes that interact with objects from the implementation class" – has the following features:

- static fields
- Static and non-static methods prototypes – declaration but NOT implementation
- The implementation class use the "***implements***" keyword
- It is possible to have interface as type – declare objects with type interface but you must call the constructor methods from a real Java class

# 2. Summary of Java SE - OOP

***Abstract Class*** – "a class that have at least one method abstract" – has the following features:

- At least one abstract method – keyword "***abstract***"
- May contain standard static and non-static fully implemented methods
- You may declare objects from an abstract class but you can NOT instantiate them, you should use constructor method from a real Java class

Pay attention @: Objects vs. vectors / arrays of objects + null pointer exception

**DEMO**

**Compile in command prompt & Eclipse – Shallow Copy vs. Deep Copy Example.**

Object Oriented Programming
for easy Java sharing

# Section Conclusions

Object Oriented Programming in Java – class, object, object's deep vs. shallow copy, interface, interface as type, abstract class, inheritance, polymorphism.

**Class is used to encapsulate attributes (fields | features | characteristics) and methods (behavior | 'interface') in order to produce instances / objects.**

**Object is a instance of a class with certain values for attributes and with same methods class for all the generated instances.**

**In OOP there are two major relationships: "has a" (composition) and "is a" (inheritance)**

Share knowledge, Empowering Minds

# Multi-paradigm Programming Intro

# 3.1 Multi-paradigm programming

Some paradigms are concerned mainly with implications for the execution model of the language, such as allowing side effects, or whether the sequence of operations is defined by the execution model. Other paradigms are concerned mainly with the way that code is organized, such as grouping a code into units along with the state that is modified by the code. Yet others are concerned mainly with the style of syntax and grammar.

Common programming paradigms include:

imperative which allows side effects,

functional which disallows side effects,

declarative which does not state the order in which operations execute,

object-oriented which groups code together with the state the code modifies,

procedural which groups code into functions,

logic which has a particular style of execution model coupled to a particular style of syntax and grammar, and

symbolic programming which has a particular style of syntax and grammar.[1][2][3]

For example, languages that fall into the **imperative paradigm** have two main features: they state the order in which operations occur, with constructs that explicitly control that order, and they allow side effects, in which state can be modified at one point in time, within one unit of code, and then later read at a different point in time inside a different unit of code. The communication between the units of code is not explicit. Meanwhile, in **object-oriented** programming, code is organized into objects that contain state that is only modified by the code that is part of the object. Most object-oriented languages are also imperative languages. In contrast, languages that fit the **declarative paradigm** do not state the order in which to execute operations. Instead, they supply a number of operations that are available in the system, along with the conditions under which each is allowed to execute.

# 3.1 Multi-paradigm programming

Just as software engineering (as a process) is defined by differing *methodologies*, so the programming languages (as models of computation) are defined by differing *paradigms*. Some languages are designed to support one paradigm (Smalltalk supports object-oriented programming, Haskell supports functional programming), while other programming languages support multiple paradigms (such as Object Pascal, C++, Java, C#, Scala, Visual Basic, Common Lisp, Scheme, Perl, PHP, Python, Ruby, Oz, and F#). For example, programs written in C++, Object Pascal or PHP can be purely procedural, purely object-oriented, or can contain elements of both or other paradigms. Software designers and programmers decide how to use those paradigm elements.

In object-oriented programming, programs are treated as a set of interacting objects. In functional programming, programs are treated as a sequence of stateless function evaluations. When programming computers or systems with many processors, in process-oriented programming, programs are treated as sets of concurrent processes acting on logically shared data structures.

Many programming paradigms are as well known for the techniques they *forbid* as for those they *enable*. For instance, pure functional programming disallows use of side-effects, while structured programming disallows use of the goto statement. Partly for this reason, new paradigms are often regarded as doctrinaire or overly rigid by those accustomed to earlier styles.[5] Yet, avoiding certain techniques can make it easier to understand program behavior, and to prove theorems about program correctness.

Programming paradigms can also be compared with *programming models* which allow invoking an execution model by using only an API. Programming models can also be classified into paradigms, based on features of the execution model.

For parallel computing, using a programming model instead of a language is common. The reason is that details of the parallel hardware leak into the abstractions used to program the hardware. This causes the programmer to have to map patterns in the algorithm onto patterns in the execution model (which have been inserted due to leakage of hardware into the abstraction). As a consequence, no one parallel programming language maps well to all computation problems. It is thus more convenient to use a base sequential language and insert API calls to parallel execution models, via a programming model. Such parallel programming models can be classified according to abstractions that reflect the hardware, such as shared memory, distributed memory with message passing, notions of *place* visible in the code, and so forth. These can be considered flavors of programming paradigm that apply to only parallel languages and programming models.

# 3.1 Multi-paradigm programming

A concise reference for the programming paradigms listed here – copyright Wiki:

- Concurrent programming – have language constructs for concurrency, these may involve multi-threading, support for distributed computing, message passing, shared resources (including shared memory), or futures
  - Actor programming – concurrent computation with *actors* that make local decisions in response to the environment (capable of selfish or competitive behavior)
- Constraint programming – relations between variables are expressed as constraints (or constraint networks), directing allowable solutions (uses constraint satisfaction or simplex algorithm)
- Dataflow programming – forced recalculation of formulas when data values change (e.g. spreadsheets)
- Declarative programming – describes actions (e.g. HTML describes a page but not how to actually display it)
- Distributed programming – have support for multiple autonomous computers that communicate via computer networks
- Functional programming – uses evaluation of mathematical functions and avoids state and mutable data
- Generic programming – uses algorithms written in terms of to-be-specified-later types that are then instantiated as needed for specific types provided as parameters
- Imperative programming – explicit statements that change a program state
- Logic programming – uses explicit mathematical logic for programming
- Metaprogramming – writing programs that write or manipulate other programs (or themselves) as their data, or that do part of the work at compile time that would otherwise be done at runtime
  - Template metaprogramming – metaprogramming methods in which templates are used by a compiler to generate temporary source code, which is merged by the compiler with the rest of the source code and then compiled
  - Reflective programming – metaprogramming methods in which a program modifies or extends itself

# 3.1 Multi-paradigm programming

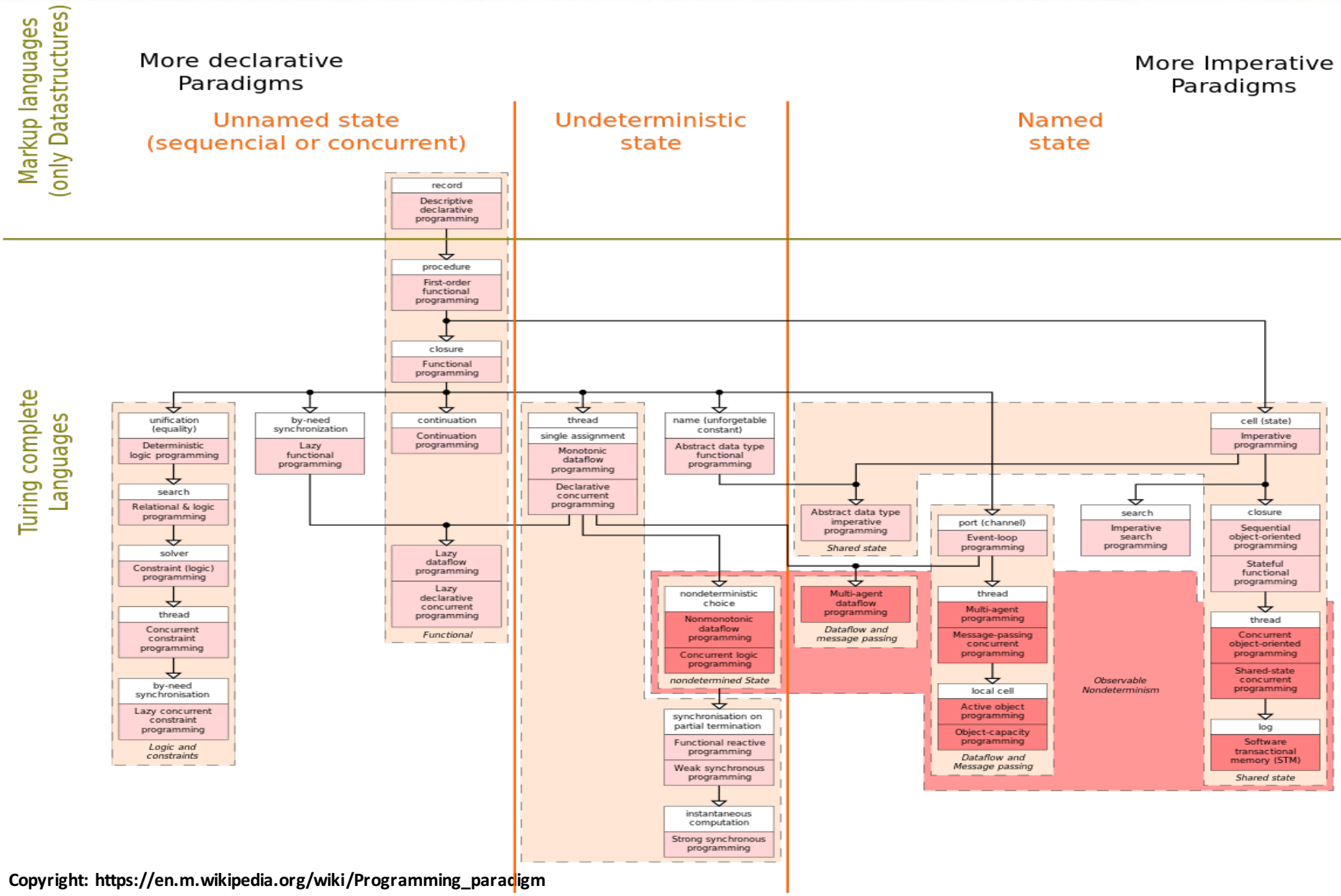
- Object-oriented programming – uses data structures consisting of data fields and methods together with their interactions (objects) to design programs
  - Class-based – object-oriented programming in which inheritance is achieved by defining classes of objects, versus the objects themselves
  - Prototype-based – object-oriented programming that avoids classes and implements inheritance via cloning of instances
- Pipeline programming – a simple syntax change to add syntax to nest function calls to language originally designed with none
- Rule-based programming – a network of rules of thumb that comprise a knowledge base and can be used for expert systems and problem deduction & resolution
- Visual programming – manipulating program elements graphically rather than by specifying them textually (e.g. Simulink or Scratch from MIT); also termed *diagrammatic programming*

# 3.1 Multi-paradigm programming

List of multi-paradigm programming languages

| Language | Number of Paradigms | Concurrent | Constraints | Dataflow | Declarative | Distributed | Functional | Meta-programming | Generic | Imperative | Logic | Reflection | Object-oriented | Pipelines | Visual | Rule-based | Other paradigms |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| C++ | 7 (15) | Yes[8][9][10] | Library[11] | Library[12][13] | Library[14][15] | Library[16][17] | Yes | Yes[18] | Yes[a 3] | Yes | Library[19][20] | Library[21] | Yes[a 2] | Yes[22] | No | Library[23] | Array (multi-dimensional; using STL) |
| C# | 6 (7) | Yes | No | Library[a 4] | No | No | Yes[a 5] | No | Yes | Yes | No | Yes | Yes[a 2] | No | No | No | reactive[a 6] |
| ECMAScript[27][28] (ActionScript, E4X, JavaScript, JScript) | 4 (5) | partial (promises, native extensions)[a 8] | No | No | No | No | Yes | No | No | Yes | No | Yes | Yes[a 9] | No | No | No | reactive[a 10] |
| Fortran | 4 (5) | Yes | No | No | No | No | Yes[a 11] | No | Yes[a 12] | No | No | No | Yes[a 2] | No | No | No | Array (multi-dimensional) |
| Go | 4 | Yes | No | No | No | No | No | No | No | Yes | No | Yes | No | Yes | No | No | No |
| Haskell | 2? | Yes | Library[29] | No | Yes | Library[30] | Yes (lazy) | No | Yes | Yes | No | No | No | No | Yes | No | Template:Reactive, dependent types (partial) |
| Java | 6 | Yes | Library[31] | Library[32] | No | No | Yes | No | Yes | Yes | No | Yes | Yes[a 2] | No | No | No | No |
| Kotlin | 8 | Yes | No | No | No | No | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | No | No | No |
| Python | 5 (10) | Library[65][66] | Library[67] | No | No | Library[68] | Partial | Yes[69][70] | Yes[71][72] | Yes | Library[73] | Yes | Yes[a 2] | No | No | No | structured |
| R | 4 | No | No | No | No | No | Yes | No | No | Yes | No | Yes | Yes | Yes[74] | No | No | Array (multi-dimensional) |
| Lua[citation needed] | 3 | No | No | No | No | No | Yes | No | No | Yes | No | No | Yes[a 9] | No | No | No | No |
| MATLAB | 6 (10) | Toolbox[41] | Toolbox[42] | Yes[43] | No | Toolbox[44] | No | Yes[45] | Yes[46] | No | No | Yes[47] | Yes[48] | No | Yes[49] | No | Array (multi-dimensional) |
| Swift | 3 | No | No | No | No | No | Yes | No | No | Yes | No | No | Yes[a 2] | No | No | No | block-structured |
| Visual Basic .NET | 6 (7) | Yes | No | Library[a 4] | No | No | Yes | No | Yes | Yes | No | Yes | Yes[a 2] | No | No | No | reactive[a 6] |
| Windows PowerShell | 6 | No | No | No | No | No | Yes | No | Yes | Yes | No | Yes | Yes[a 2] | Yes | No | No | No |
| Wolfram Language & Mathematica | 13[82] (14) | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | No | No | Yes | Knowledge Based |

# 3.1 Multi-paradigm programming



# 3.2 Repositories: Git from Linux VM

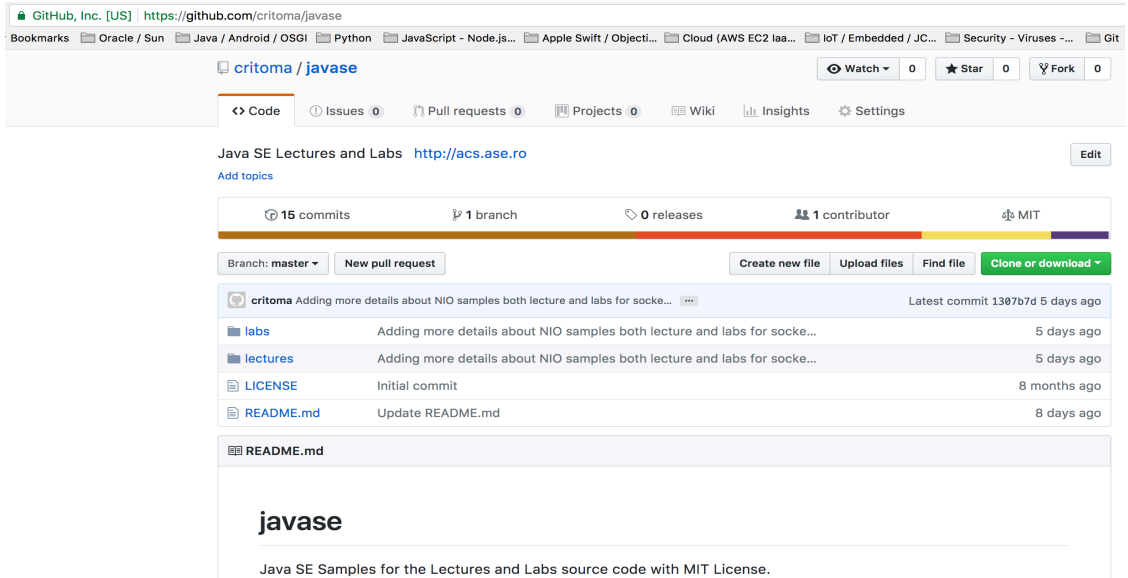
Mandatory steps to get the lectures and the labs: (Optional full Git presentation @ISM – Cyber-Security Master – www.ism.ase.ro by Cristian Toma & Casian Andrei @ Oracle):

1. Download & install the Linux Ubuntu 16.04 LTS VM OVA appliance from Google drive:

2. In Ubuntu Linux VM download the Git repo with the sources for the lecture & labs:
https://github.com/critoma/javase | http://acs.ase.ro/java

Optional – set your e-mail and Git global username if you want to contribute:
git config --global user.email cristian.toma@ie.ase.ro
git config --global user.name "critoma"

Mandatory – clone or pull the latest Git repo updates OR Download-ZIP (web):
**git clone https://github.com/critoma/javase.git**
**git pull**



Questions & Answers!

# Thanks!

Java Application Development
End of Lecture 1 – summary of Java SE