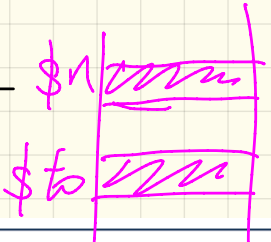
EECS1021 Object-Oriented Programming: From Sensors to Actuators

Winter 2019

Monday January 7

Lecture 1

Low-Level PL vs. High-Level PL



Assembly

```
lw      $t0, $n          # fetch N, store in $t0
mult    $t0, $t0, $t0    # store N*N in $t0
lw      $t1, $n          # fetch N, store in $t1
mult    $t1, $t1, 3      # store 3*N in $t1
add     $t2, $t0, $t1    # store N*N + 3*N in $t2
sw      $t2, $i          # store N*N + 3*N in $i
```

$n^2 + 3n$

OOP e.g. Java

```
1  Scanner keyboard = new Scanner(System.in);
2  int weight = keyboard.nextInt();
3  int height = keyboard.nextInt();
4  int bmi = weight / (height * height);
5  System.out.println("BMI (Body Mass Index) is: " + bmi);
```

w/h^2

Lab

→ programming assignment given

Completed

→ lab session

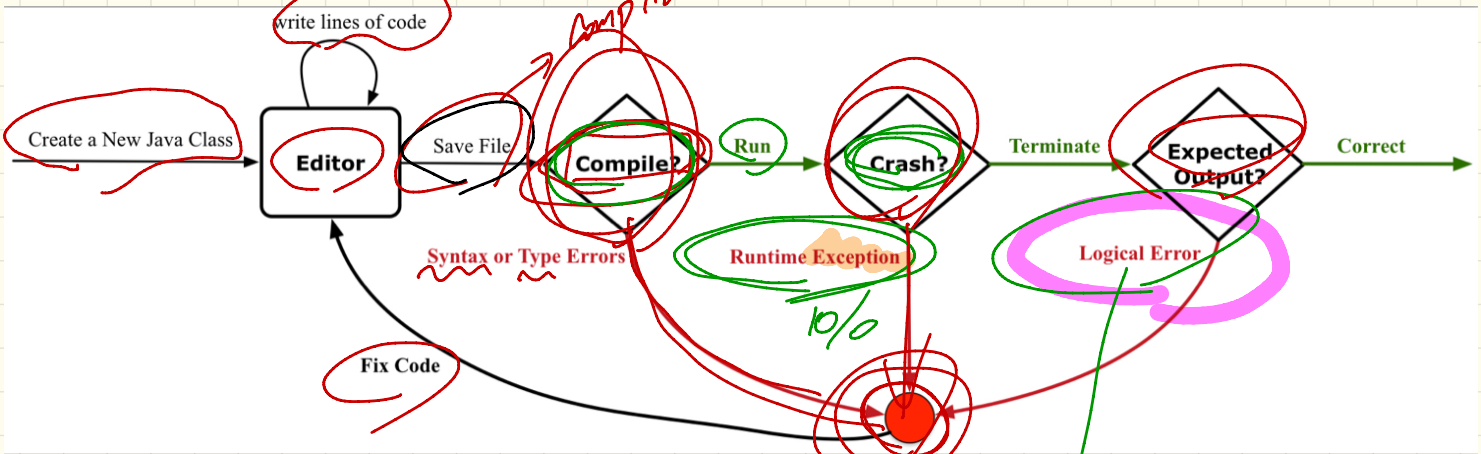
1. quit
2. graded
3. Wait until you can start checking out HW.

Wednesday January 9

Lecture 2

- Office Hours 3pm ~ 5pm
  W / F
- Lab 0 Part I
  W 6pm
  F 5:30pm
- More exercises : codingbat.com

Development Process

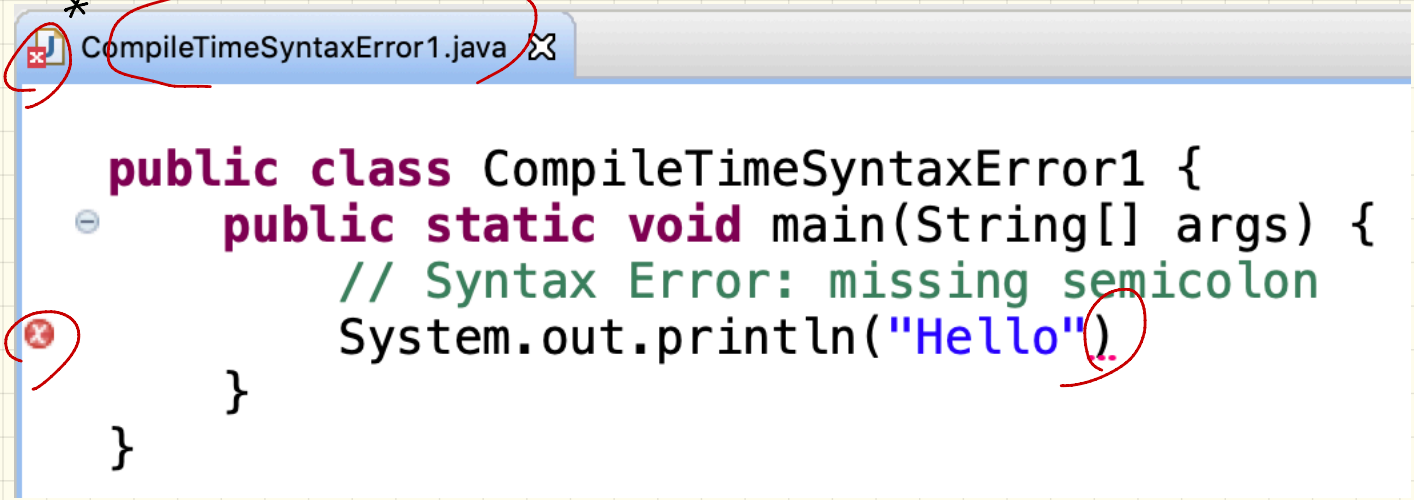


↓ ① Test

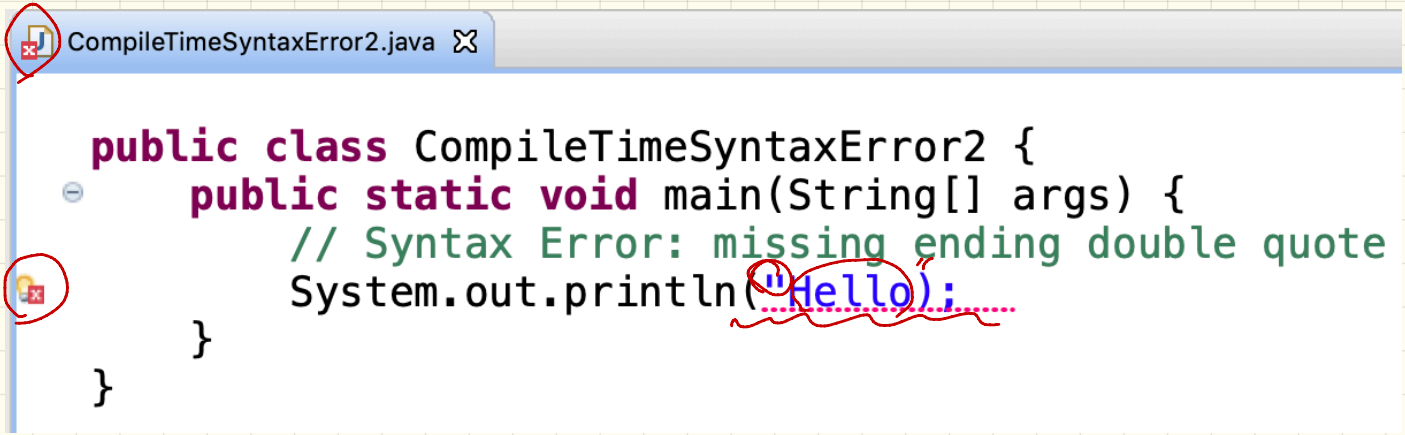
② debugger.

Error at the Compile Time : Syntax Error (I)

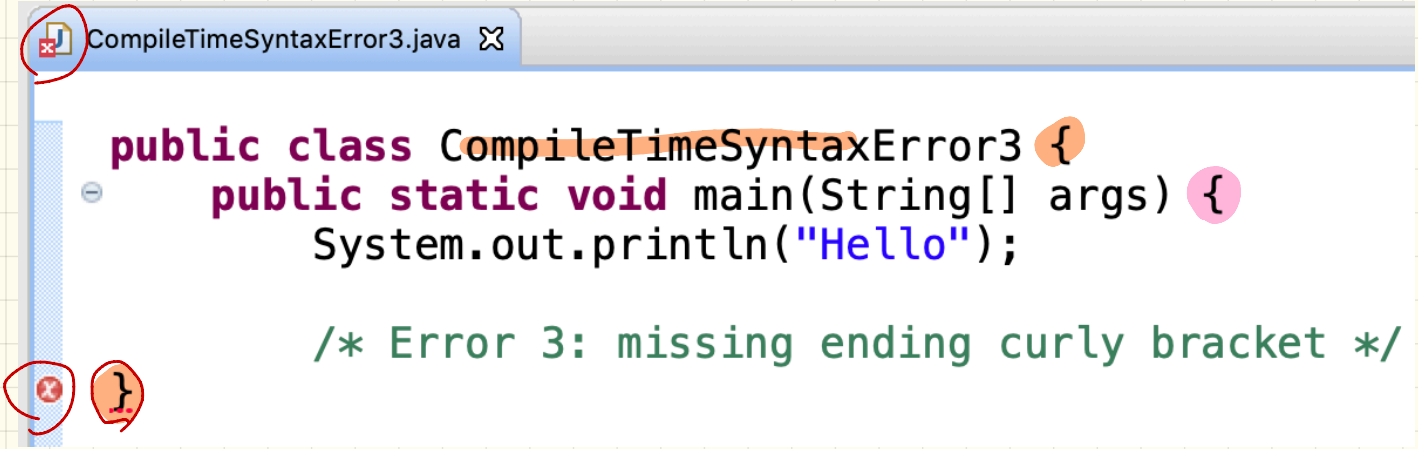
* ← unsaved



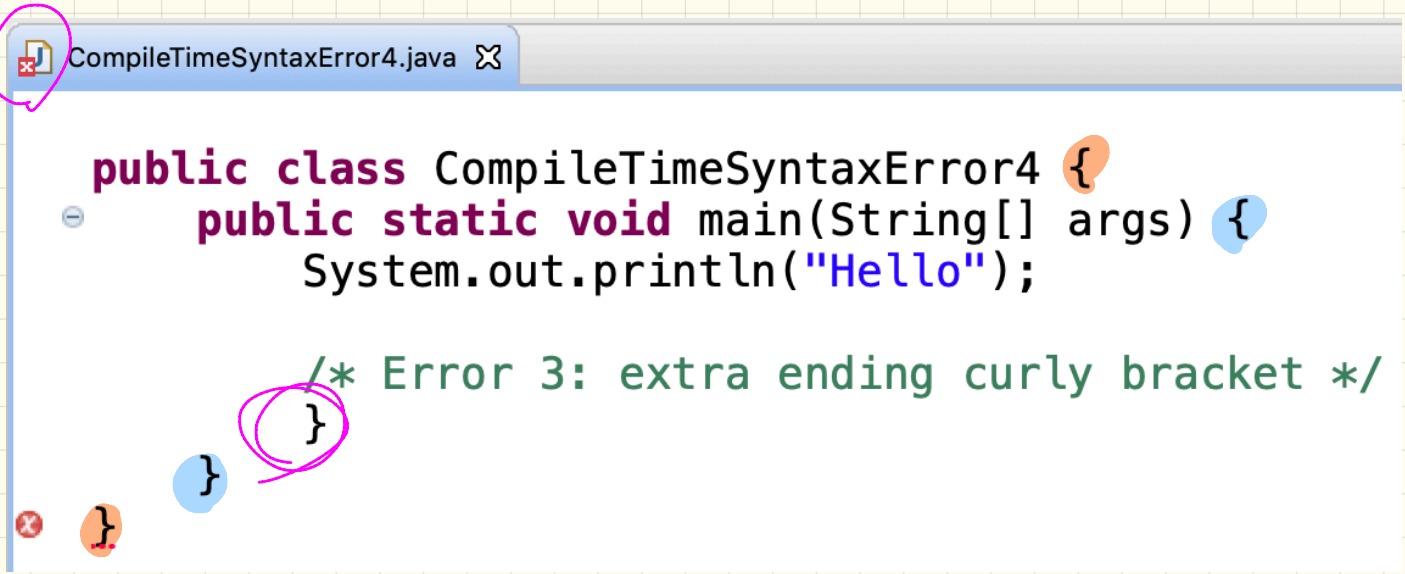
Error at the Compile Time : Syntax Error (2)



Error at the Compile Time : Syntax Error (3)



Error at the Compile Time : Syntax Error (4)



Error at the Compile Time : Type Error (I)

CompileTimeTypeError1.java

```
public class CompileTimeTypeError1 {
    public static void main(String[] args) {
        /* Type error: Apply operator to the wrong values */
        System.out.println("York" * 23);
    }
}
```

type error

Error at the Compile Time : Type Error (2)

CompileTimeTypeError2.java

```
public class CompileTimeTypeError2 {
    public static void main(String[] args) {
        /* Type error: Refer to undeclared variable */
        int i = 23;
        System.out.println(j / 3);
    }
}
```

undeclared variable

Error at the Run Time : Exception

RunTimeException.java

```java
public class RunTimeException {
    public static void main(String[] args) {
        /* Runtime exception: code compiles but crashes at runtime */
        System.out.println(10 / 0);
    }
}
```

Error at the Run Time : Logical Error

RunTimeLogicalError.java

```
import java.util.Scanner;

public class RunTimeLogicalError {
    public static void main(String[] args) {
        /* Runtime logical error: code compiles, does not crash at runtime,
         * but does not behave as expected.
         */
        Scanner input = new Scanner(System.in);

        System.out.println("Enter the integer radius of a circle:");
        int radius = input.nextInt();

        System.out.println("Area of circle is: " + (2 * 3.14 * radius));
        input.close();
    }
}
```

Document Your Code

- **Single-Lined** Comments:

```
// This is Comment 1.
... // Some code
// This is Comment 2.
```

- **Multiple-Lined** Comments:

```
/* This is Line 1 of Comment 1.
 */
... // Some code
/* This is Line 1 of Comment 2.
 * This is Line 2 of Comment 2.
 * This is Line 3 of Comment 2.
 */
```

character

X

'' ✓

'a'

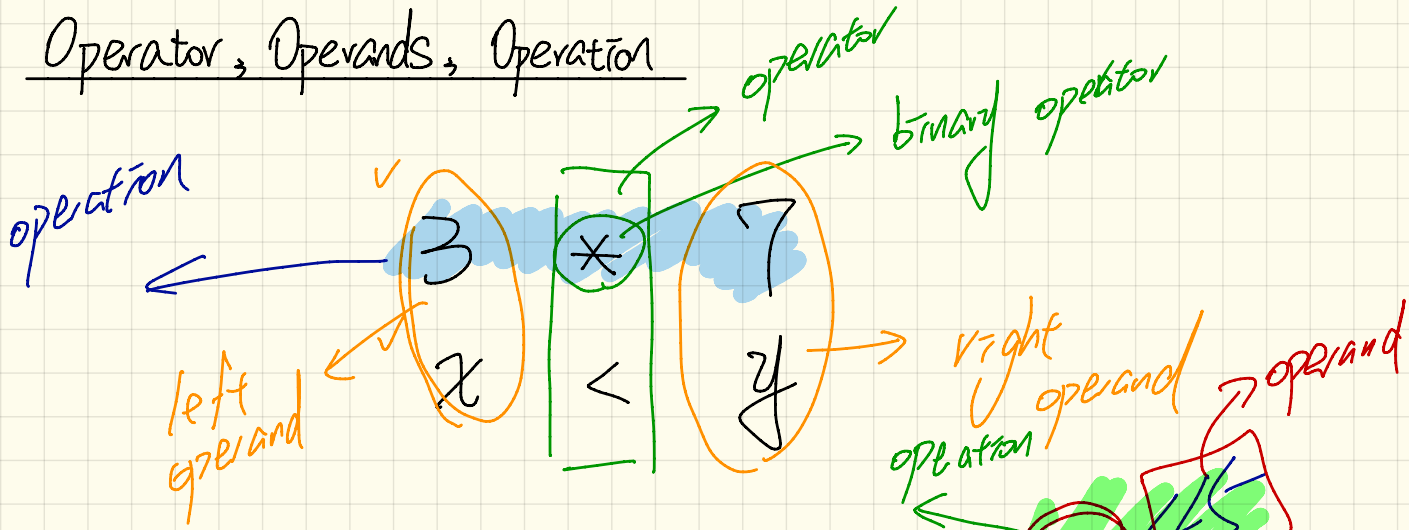
String : seq. of char.

"" ✓ empty string

"a"

"abcd"

Operator, Operands, Operation



✓ – An operation consists of an operator and one or more operands.

✓ – An operator has one or more applicable operands.

Monday January 14

Lecture 3

- Lab0 Part 2

```
if ( registered ) {
        go to your registered lab session
}
else {
        go to Tuesday 5pm session
}
```

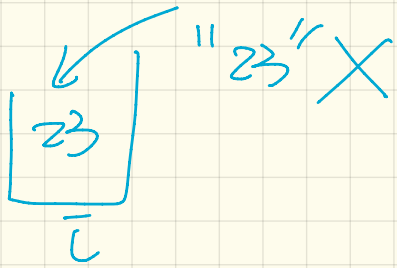
preparation guide soon

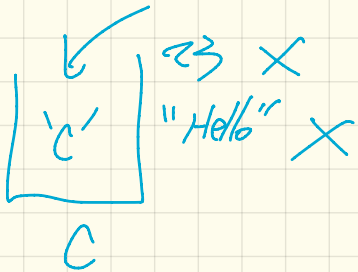
- Quiz 1 Week of Jan 21
- Lab Test 1 Week of Jan 28

Data Types

```
int i = ? ;
double d = ? ;
char c = ? ;
String s = ? ;
boolean b = ? ;
```

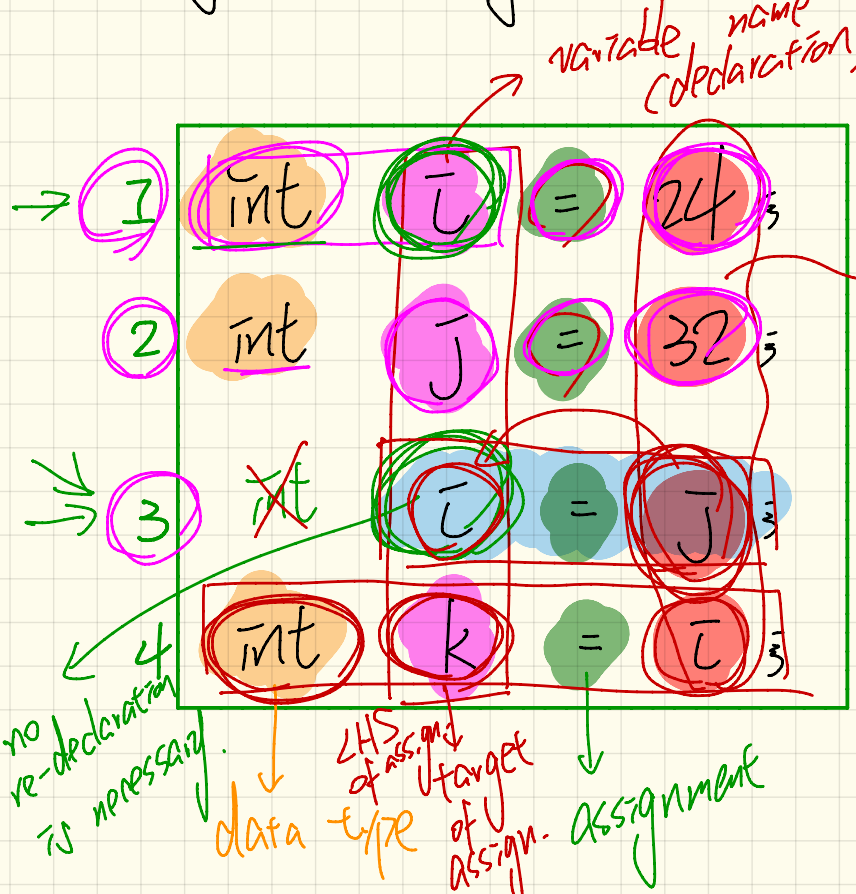
true, false





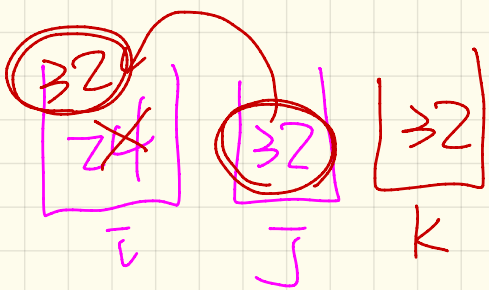
Assignment : Change of Value

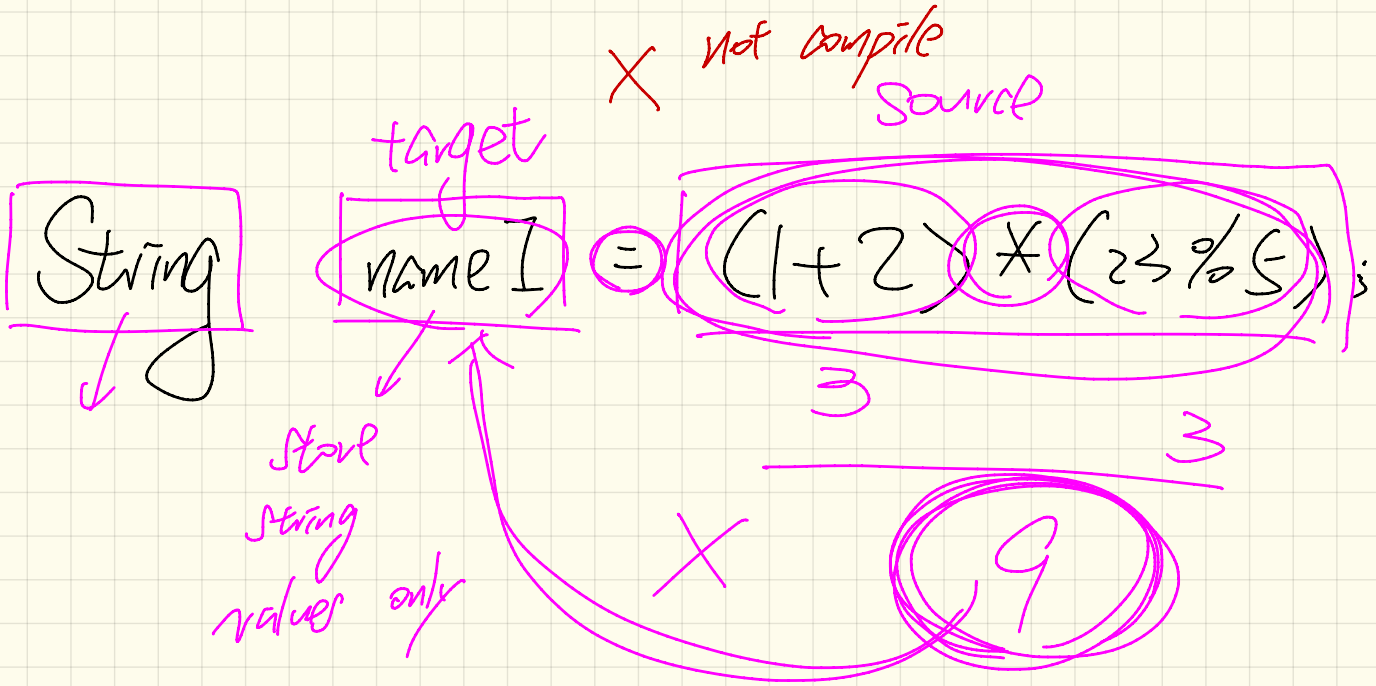
- type
- target
- source



RHS of assignment

source of assign.




X not compile
target
source
String
name1
=
(1+2)*(23%5);
3
3
store string values only
X
9

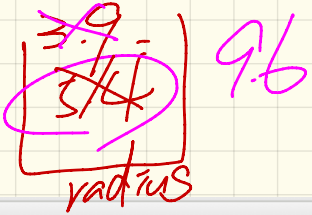
Constant : Initialization vs. Re-assignment

ConstantCannotBeReassigned.java

```
public class ConstantCannotBeReassigned {
    public static void main(String[] args) {
        /* A constant can only be initialized once. */
        final double pi = 3.14;
        /* Reassignment of a constant is illegal. */
        pi = 6.28;
    }
}
```

pi can be initialized, but cannot re-assigned.

Variable : Initialization vs. Re-assignment



VariableCanBeReassigned.java

```
public class VariableCanBeReassigned {
    public static void main(String[] args) {
        /* A variable can be initialized. */
        double radius = 5.4;
        System.out.println("Radius is: " + radius);

        /* A variable may be re-assigned for as many times as necessary */
        radius = 3.9;
        System.out.println("Radius is: " + radius);
        System.out.println("Radius is: " + radius);

        radius = 9.6;
        System.out.println("Radius is: " + radius);
    }
}
```

we can re-assign this var.

re-assign the variable to a new value.

Expressions (1)

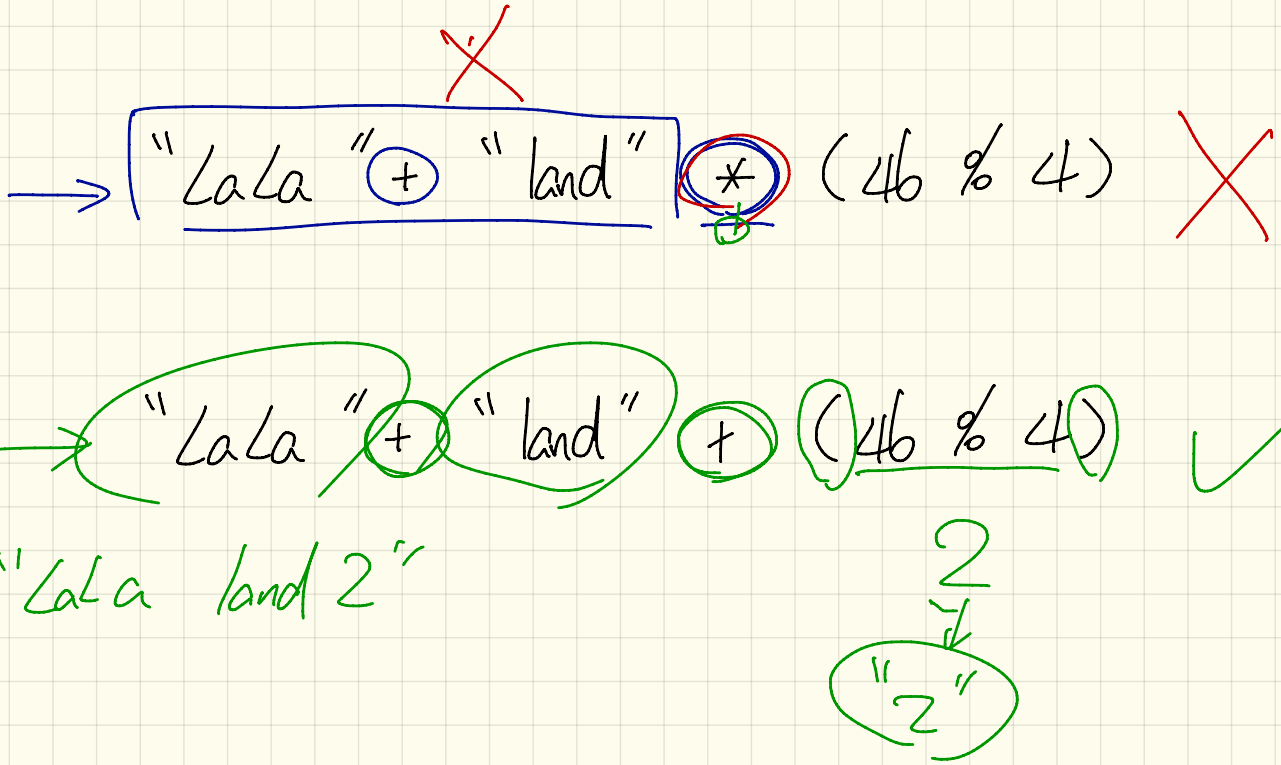
"Hello 5" ← "Hello " + (3+2)

concat. ↙ ↘ addition

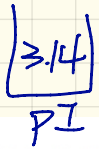
force the evaluation first

	Type Correct?
(1 + 2) * (23 % 5)	✓ 9
"Hello " + "world"	✓ "Hello world"
"Hello " * "world"	✗
"Hello " + 3 + 2	✓ "Hello 32"
"Hello " * 3	✗
"46" % "4"	✗

Expressions (2)



Combining Constants and Variables



e.g., Print statements involving literals or named constants only:

```
final double PI = 3.14; /* a named double constant */
System.out.println("Pi is " + PI); /* str. lit. and num. const. */
System.out.println("Pi is " + PI);
```

3.14

3.14

e.g., Print statements involving variables:

```
String msg = "Counter value is "; /* a string variable */
int counter = 1; /* an integer variable */
System.out.println(msg + counter);
System.out.println(msg + counter);
counter = 2; /* re-assignment changes variable's stored value */
System.out.println(msg + counter);
```

re-assignment to msg and counter

Console Application: With User Inputs vs. Without

```java
public class ComputeArea {
  public static void main(String[] args) {
    double radius; /* Declare radius */
    double area; /* Declare area */
    /* Assign a radius */
    radius = 20; /* assign value to radius */
    /* Compute area */
    area = radius * radius * 3.14159;
    /* Display results */
    System.out.print("The area of circle with radius ");
    System.out.println(radius + " is " + area);
  }
}
```

Without User Input

With User Input

```java
import java.util.Scanner;
public class ComputeAreaWithConsoleInput {
  public static void main(String[] args) {
    /* Create a Scanner object */
    Scanner input = new Scanner(System.in);
    /* Prompt the user to enter a radius */
    System.out.print("Enter a number for radius: ");
    double radius = input.nextDouble();
    /* Compute area */
    final double PI = 3.14159; /* Declared constant for π */
    double area = PI * radius * radius; /* area = πr² */
    /* Display result */
    System.out.println(
      "Area for circle of radius " + radius + " is " + area);
  }
}
```

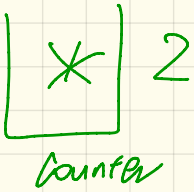
Common Mistake: Declaring the Same Variable More Than Once

```
int counter = 1;
int counter = 2;
```

re-declaration is not allowed.

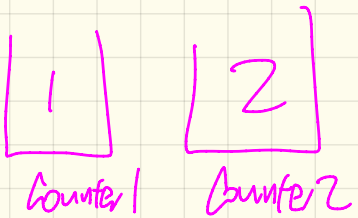
Fix 1: Only Keep the 1st Declaration

```
int counter = 1;
counter = 2;
```



Fix 2: Declare New Variables

```
int counter1 = 1;
int counter2 = 2;
```



Common Mistake: Using a Variable Before Declaring It

int counter = 1;

variable

```
① System.out.println("Counter value is " + counter);
② int counter = 1;
③ counter = 2;
④ System.out.println("Counter value is " + counter);
```

Example : Convert Seconds to Minutes

```java
import java.util.Scanner;
public class DisplayTime {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    /* Prompt the user for input */
    System.out.print("Enter an integer for seconds: ");
    int seconds = input.nextInt();
    int minutes = seconds / 60; /* minutes */
    int remainingSeconds = seconds % 60; /* seconds */
    System.out.print(seconds + " seconds is ");
    System.out.print(" minutes and ");
    System.out.println(remainingSeconds + " seconds");
  }
}
```

Test: 500 seconds

Exercise: Modify the program so that it will display hours if necessary. e.g. 7945 seconds → 2 hours 12 minutes 9 seconds

Where Can Assignment Source (RHS) Come From?

In `tar = src`, the *assignment source* `src` may come from:

- A literal

```
int i = 23;
```

- A variable

```
int i = 23;
int j = i;
```

- An expression involving literals and variables

```
int i = 23;
int j = i * 2;
```

- An input from the user

```
Scanner input = new Scanner(System.in);
int i = input.nextInt();
int j = i * 2;
```

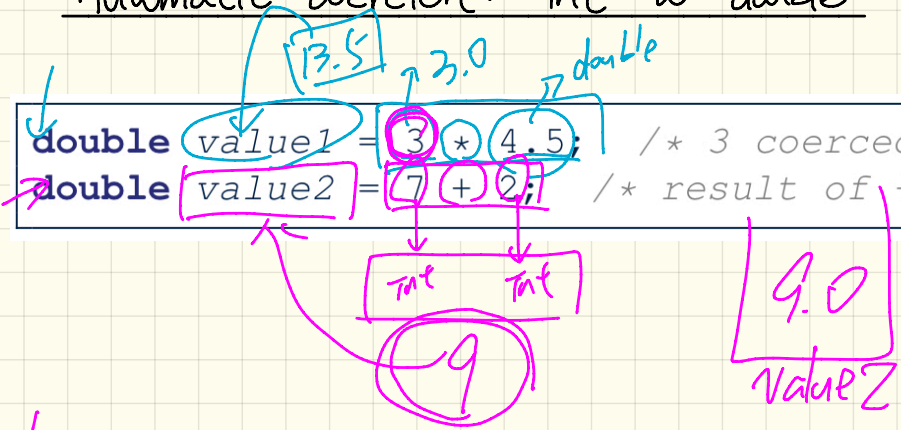
Wednesday January 16

Lecture 4

- Quiz 1 guide posted
- Lab 1 programming tasks posted
- Lab Test 1 guide (tomorrow)

Automatic Coercion: int to double

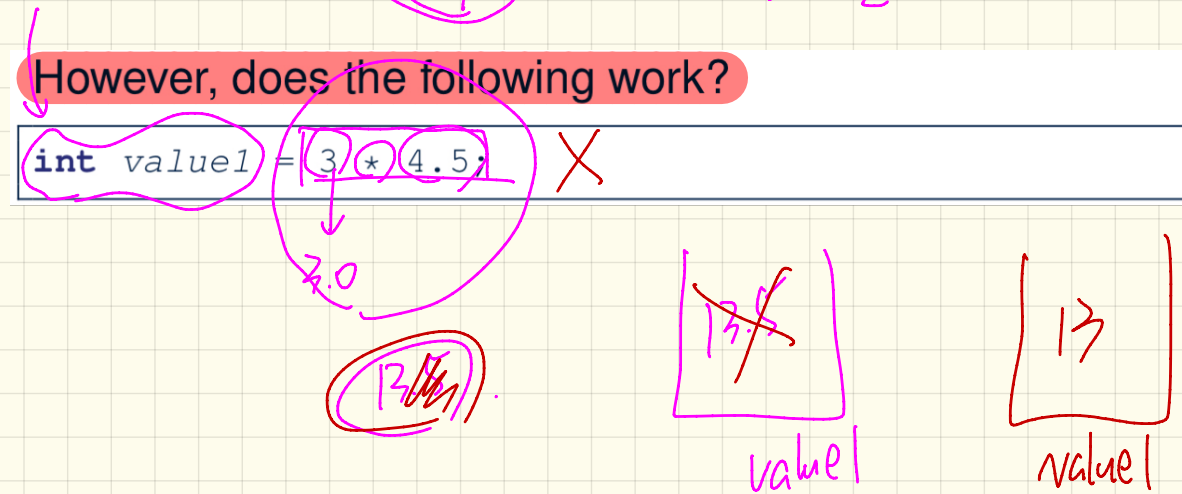
```
double value1 = 3 * 4.5;  /* 3 coerced to 3.0 */
double value2 = 7 + 2;    /* result of + coerced to 9.0 */
```



However, does the following work?

```
int value1 = 3 * 4.5;
```

X



Manual Casting

Case 1 : double to int

int value1 = 3.1415926; ✕

int value2 = (int) 3.1415926;

double

int

3

(x.0 + y) / 2

int int

9 / 2 4

9 / 2.0

9.0

4.5

int x = input.nI

int y = input.nI

7

double avg = (x+y)/2;

Manual Casting

precedence

(double) (1/2) → 0 → 0.0

Case 2: int to double

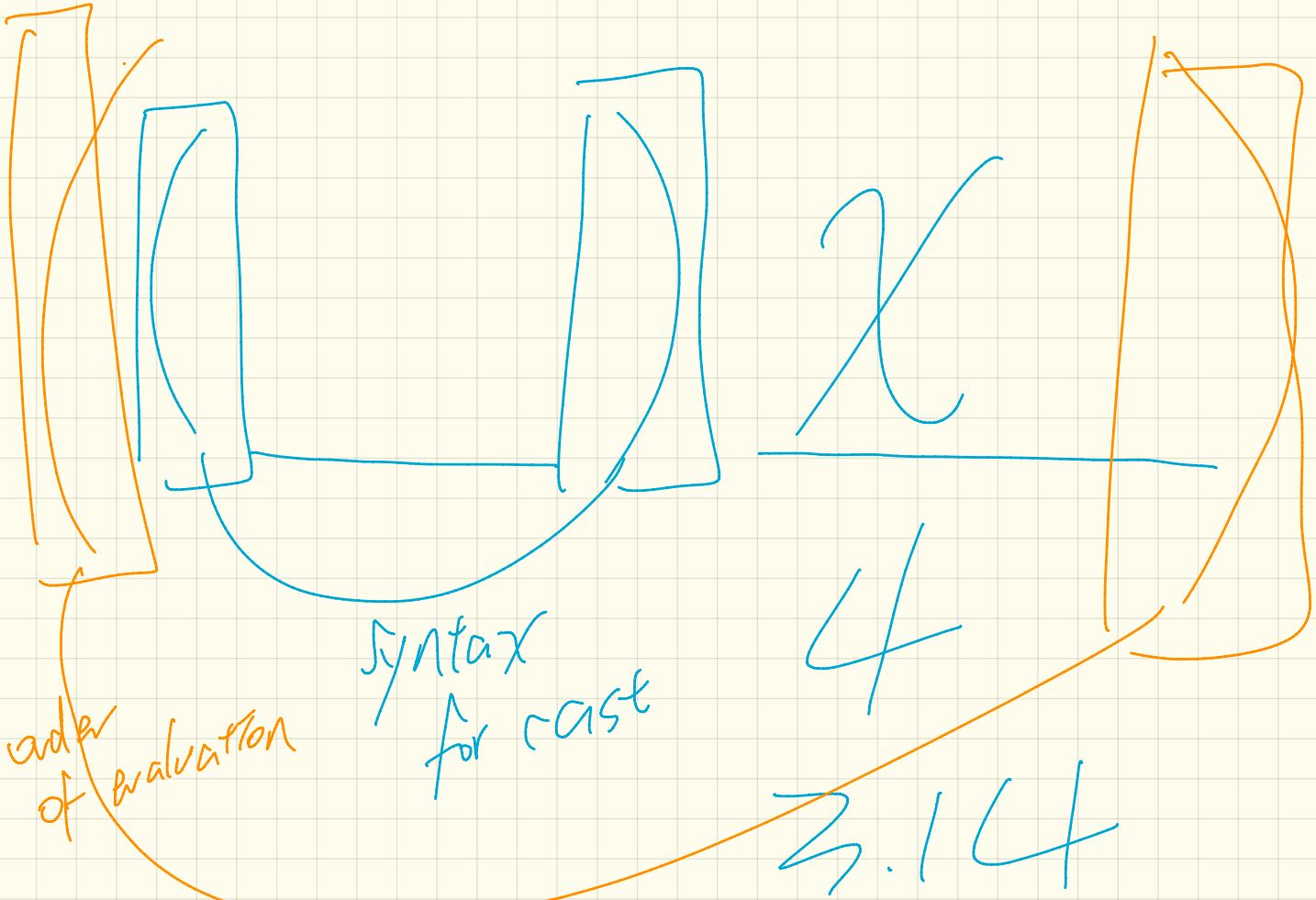
```
1  System.out.println( 1 / 2 );                          /* 0 */
2  System.out.println( ((double) 1) / 2 );               /* 0.5 */
3  System.out.println( 1 / ((double) 2) );               /* 0.5 */
4  System.out.println( ((double) 1) / ((double) 2) );    /* 0.5 */
5  System.out.println( (double) 1 / 2 );                 /* 0.5 */
6  System.out.println( (double) (1 / 2) );               /* 0.0 */
```

✓ 0.5

(double) 1 → 1.0 / 2

(double) 1 / 2

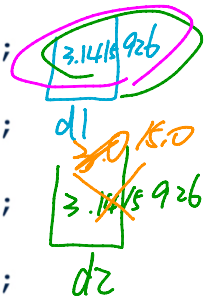
cast: * div: /


(int) x
4
3.14
syntax for cast
order of evaluation

Exercise

Consider the following Java code:

```
1  double d1 = 3.1415926;
2  System.out.println("d1 is " + d1);
3  double d2 = d1;
4  System.out.println("d2 is " + d2);
5  int i1 = (int) d1;
6  System.out.println("i1 is " + i1);
7  d2 = i1 * 5;
8  System.out.println("d2 is " + d2);
```



Write the **exact** output to the console.

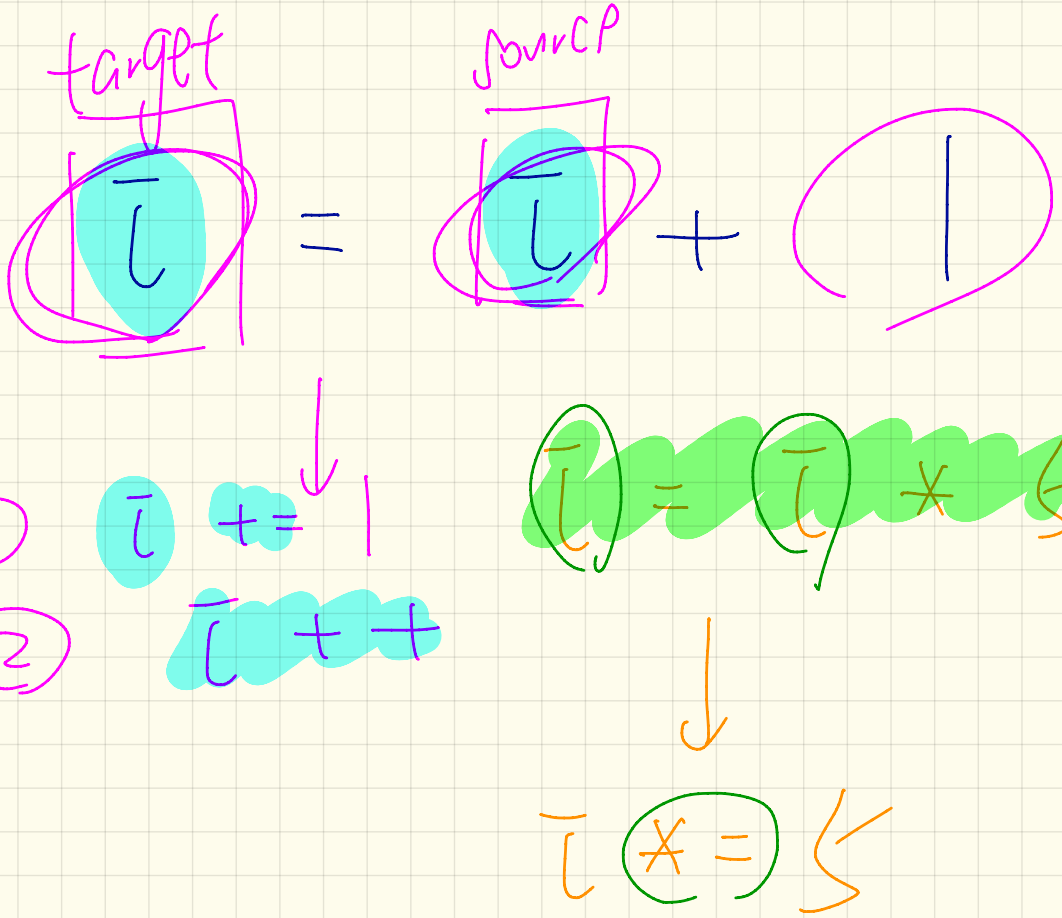
d1 is 3.1415926
d2 is 3.1415926
i1 is 3
d2 is 3*5=15.0

Exercise

Consider the following Java code, is each line type-correct? Why and Why Not?

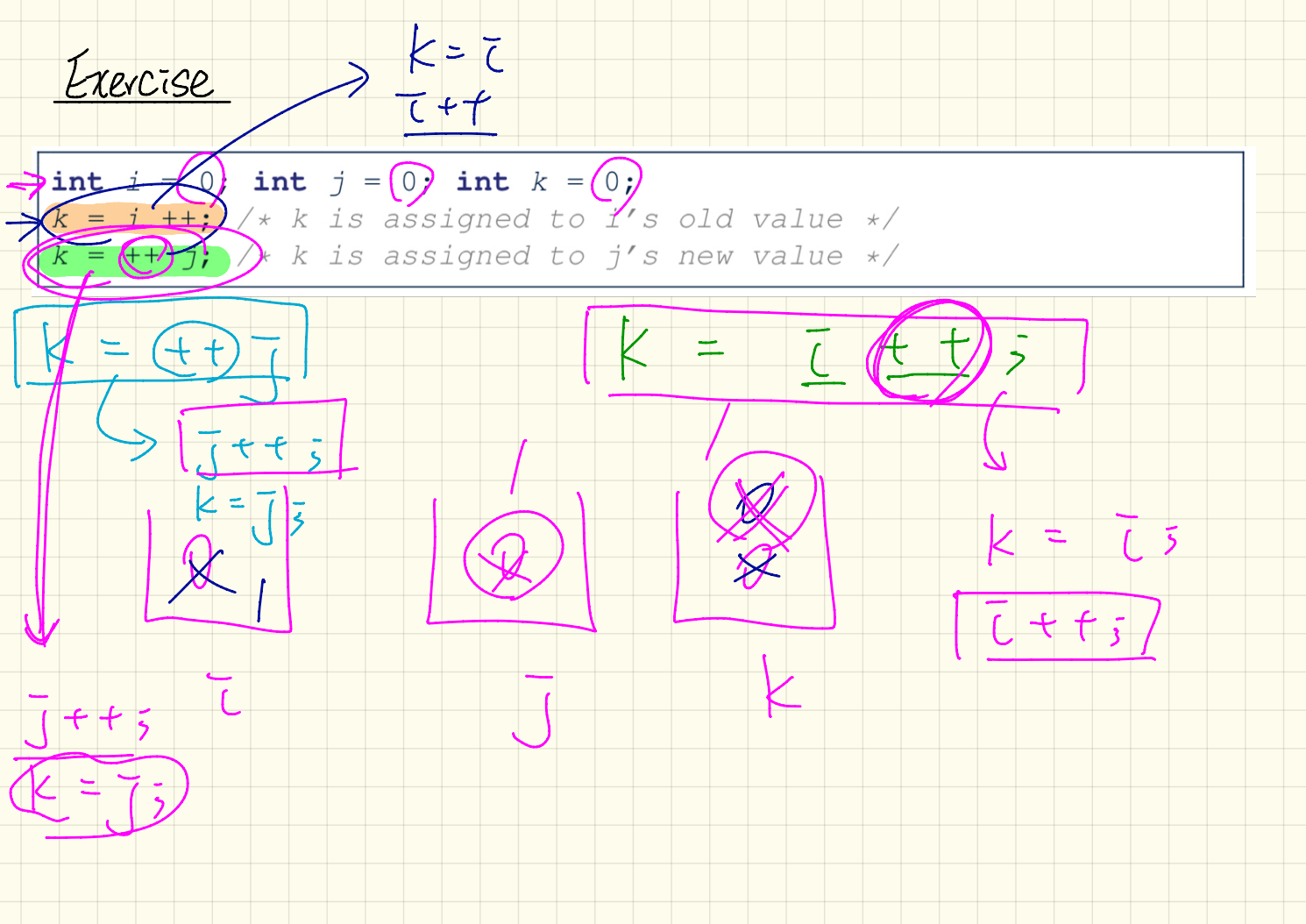
```
1  double d1 = 23;
2  int i1 = 23.6;
3  String s1 = " ";
4  char c1 = " ";
```

```
1  int i1 = (int) 23.6;
2  double d1 = i1 * 3;
3  String s1 = "La ";
4  String s2 = s1 + "La Land";
5  i1 = (s2 * d1) + (i1 + d1);
```


target
sourCP
i = i + 1
① i += 1
② i++
i = i * 5
i *= 5

Exercise

```
int i = 0; int j = 0; int k = 0;
k = i++; /* k is assigned to i's old value */
k = ++j; /* k is assigned to j's new value */
```



Comparison of Values

int
double
char
boolean

use ==

e.g.
```
char c1 = 'a';
println ( c1 == 'b' );
```

c1 → 'a'

'b' → False

String

use equals

e.g.
```
String s1 = input.nextLine();
println ( s1.equals ("quit") );
```

```
s1 == "quit"
```

compile but not working

Escape Sequence

```
String s1 = "A";
String s2 = "B";
```

\ / /

start

start or end ?

```
print ( s1 );
print ( s2 );
```

AB

"\n"

```
println ( s1 );
println ( s2 );
```

A
B

new line

```
print ( s1 + "\n" );
println ( s2 );
```

A
B

Monday January 21

Lecture 5

- Lab Test 1 (week of Jan. 28)
  - ~ guide
  - ~ tutorial video
  - ~ two example tests

Why Selective Actions

```
1  import java.util.Scanner;
2  public class ComputeArea {
3    public static void main(String[] args) {
4      Scanner input = new Scanner(System.in);
5      final double PI = 3.14;
6      System.out.println("Enter the radius of a circle:");
7      double radiusFromUser = input.nextDouble();
8      double area = radiusFromUser * radiusFromUser * PI;
9      System.out.print("Circle with radius " + radiusFromUser);
10     System.out.println(" has an area of " + area);
11   }
12 }
```

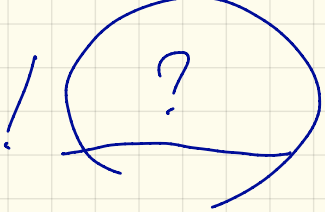
If the user enters a positive radius value as expected:

```
Enter the radius of a circle:
3
Circle with radius 3.0 has an area of 28.26
```

However, if the user enters a negative radius value:

```
Enter the radius of a circle:
-3
Circle with radius -3.0 has an area of 28.26
```

Not Equal to




```
int x = 3;
int y = 4;
int z = 4;
```

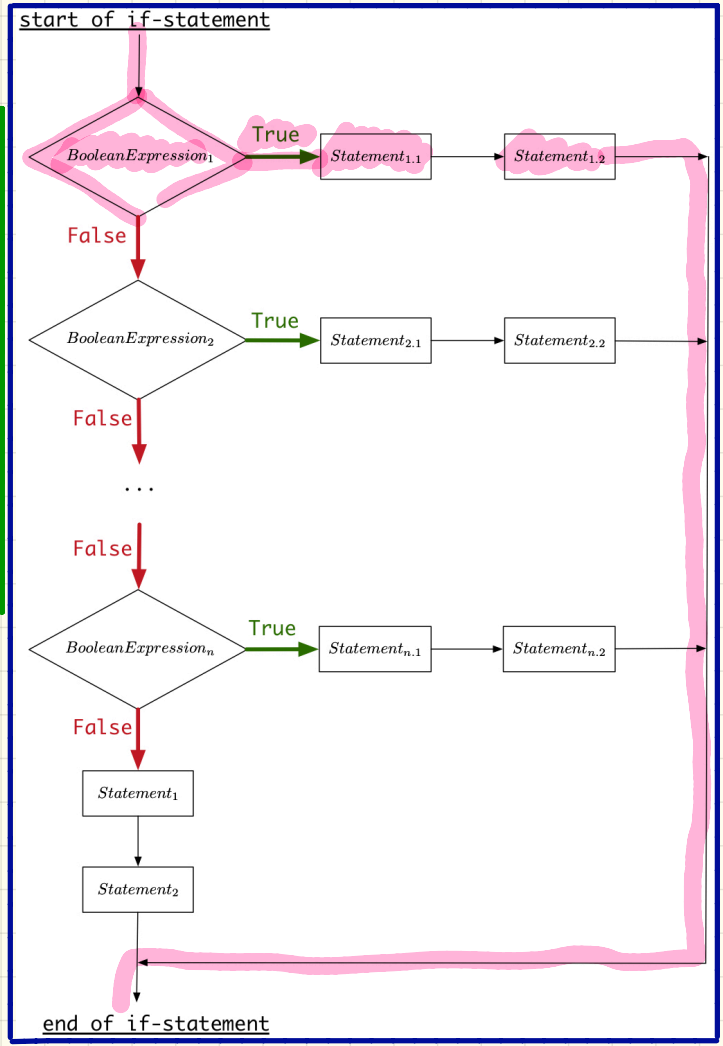
A Single If-Statement

```
if ( BooleanExpression1 ) {  /* Mandatory */
  Statement1.1; Statement2.1;
}
else if ( BooleanExpression2 ) {  /* Optional */
  Statement2.1; Statement2.2;
}
... /* as many else-if branches as you like */
else if ( BooleanExpressionn ) {  /* Optional */
  Statementn.1; Statementn.2;
}
else {  /* Optional */
  /* when all previous branching conditions ar
  Statement1; Statement2;
}
```

Syntax

Case 1: BooleanExpression1 evaluates to true

Semantics/ Meaning



Only **first** satisfying branch *executed*; later branches *ignored*.

```
int i = -4;
if (i < 0) {
  System.out.println("i is negative");
}
else if(i < 10) {
  System.out.println("i is less than than 10");
}
else if(i == 10) {
  System.out.println("i is equal to 10");
}
else {
  System.out.println("i is greater than 10");
}
```

```
i is negative
```

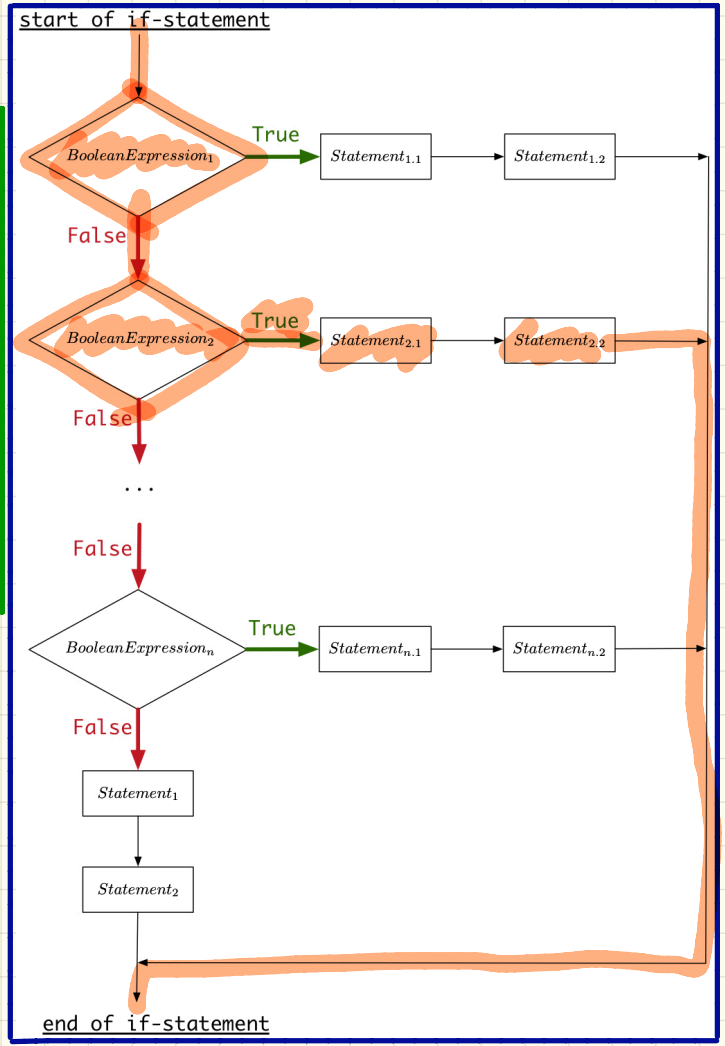
A Single If-Statement

```
if ( BooleanExpression1 ) {  /* Mandatory */
  Statement1.1; Statement2.1;
}
else if ( BooleanExpression2 ) {  /* Optional */
  Statement2.1; Statement2.2;
}
... /* as many else-if branches as you like */
else if ( BooleanExpressionn ) {  /* Optional */
  Statementn.1; Statementn.2;
}
else {  /* Optional */
  /* when all previous branching conditions ar
  Statement1; Statement2;
}
```

Syntax

Case 2: $BooleanExpression_1$ evaluates to false but $BooleanExpression_2$ evaluates to true

Semantics/ Meaning



Only **first** satisfying branch *executed*; later branches *ignored*.

```
int i = 5;
if (i < 0) {
  System.out.println("i is negative");
}
else if (i < 10) {
  System.out.println("i is less than than 10");
}
else if(i == 10) {
  System.out.println("i is equal to 10");
}
else {
  System.out.println("i is greater than 10");
}
```

```
i is less than 10
```

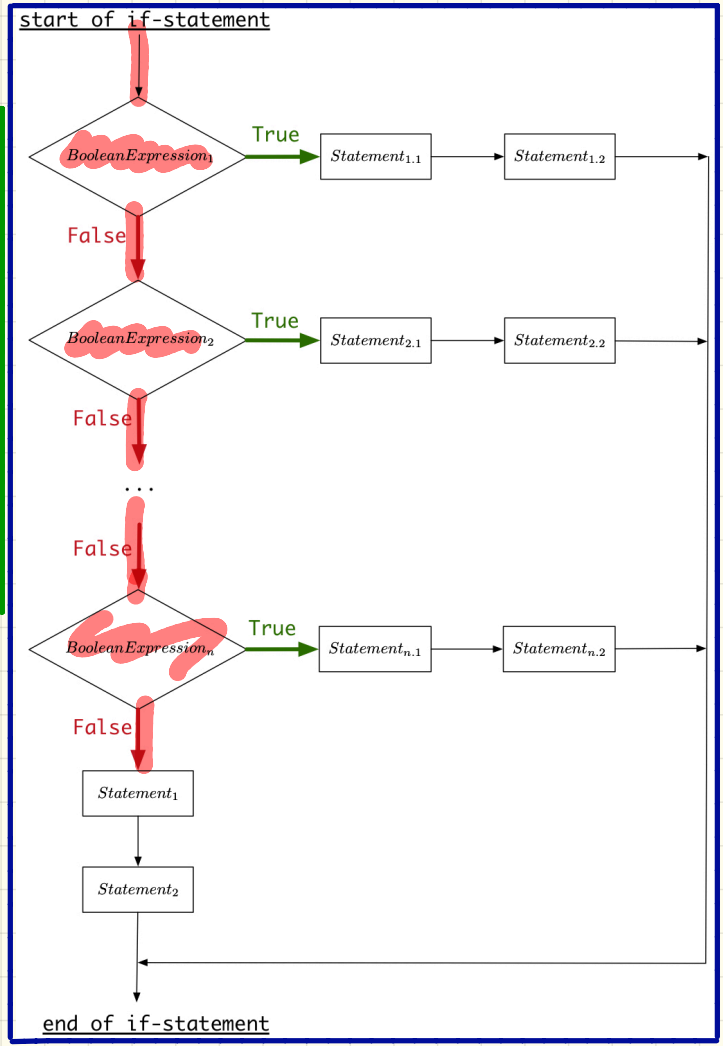
A Single If-Statement

```
if ( BooleanExpression1 ) {  /* Mandatory */
  Statement1.1; Statement2.1;
}
else if ( BooleanExpression2 ) {  /* Optional */
  Statement2.1; Statement2.2;
}
... /* as many else-if branches as you like */
else if ( BooleanExpressionn ) {  /* Optional */
  Statementn.1; Statementn.2;
}
else {  /* Optional */
  /* when all previous branching conditions ar
  Statement1; Statement2;
}
```

Syntax

Case 3: BooleanExpression1 evaluates to false but BooleanExpression2 evaluates to false

Semantics/ Meaning



A Single If-Statement

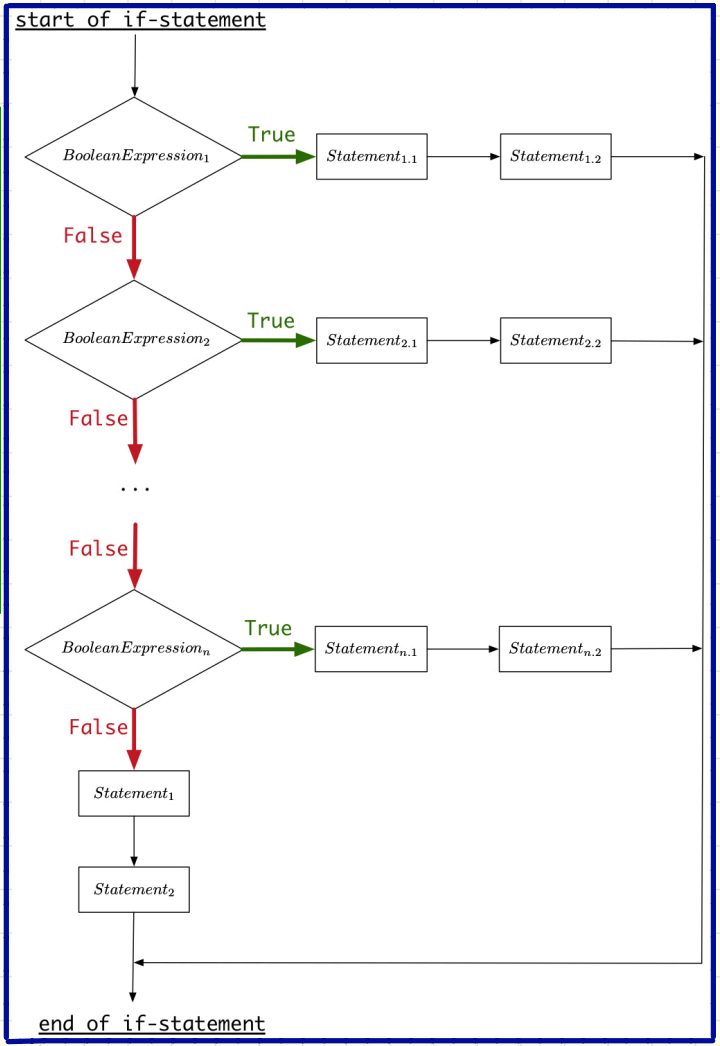
```
if ( BooleanExpression1 ) {  /* Mandatory */
  Statement1.1; Statement2.1;
}
else if ( BooleanExpression2 ) {  /* Optional */
  Statement2.1; Statement2.2;
}
... /* as many else-if branches as you like */
else if ( BooleanExpressionn ) {  /* Optional */
  Statementn.1; Statementn.2;
}
else {  /* Optional */
  /* when all previous branching conditions ar
  Statement1; Statement2;
}
```

default.

Syntax

Case 4: BooleanExpression1 & BooleanExpressionn all evaluate to false

Semantics/ Meaning



No satisfying branches, and no `else` part, then *nothing* is executed.

```
int i = 12;
if (i < 0) {
 System.out.println("i is negative");
}
else if (i < 10) {
 System.out.println("i is less than than 10");
}
else if (i == 10) {
 System.out.println("i is equal to 10");
}
```

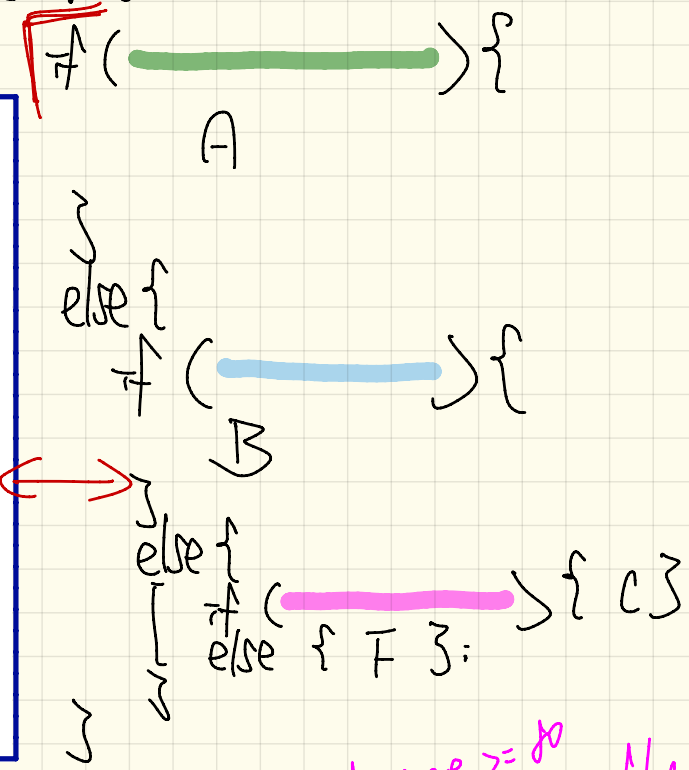
No satisfying branches, then `else` part, if there, is *executed*.

```
int i = 12;
if (i < 0) {
  System.out.println("i is negative");
}
else if (i < 10) {
  System.out.println("i is less than than 10");
}
else if (i == 10) {
  System.out.println("i is equal to 10");
}
else {
  System.out.println("i is greater than 10");
}
```

```
i is greater than 10
```

Multi-Way If-Statement with else Part

```
if (score >= 80.0) {
  System.out.println("A");
}
else if (score >= 70.0) {
  System.out.println("B");
}
else if (score >= 60.0) {
  System.out.println("C");
}
else {
  System.out.println("F");
}
```



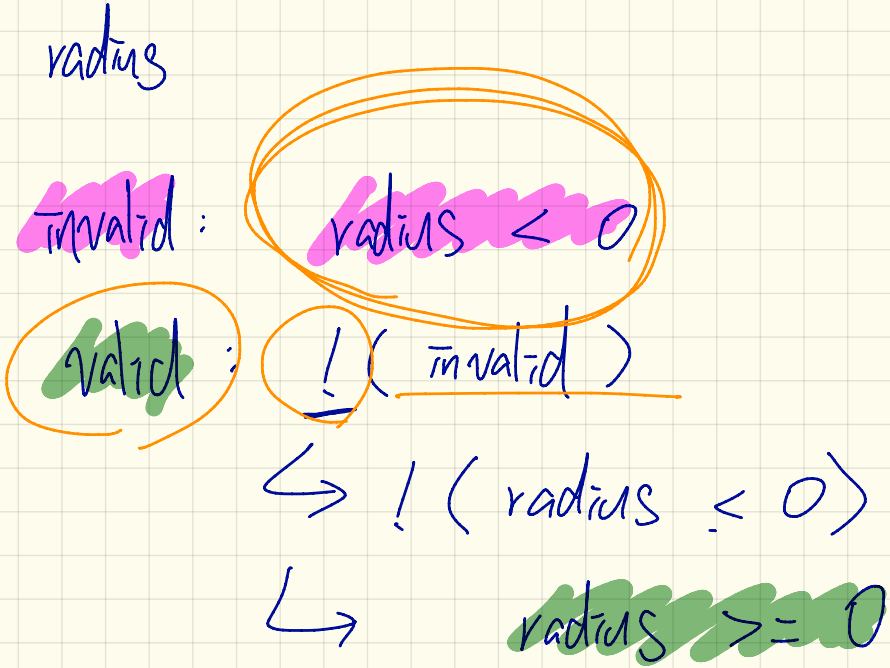
always evaluated at runtime? No, only if score >= 80 evaluates to false.

Multi-Way If-Statement without else part

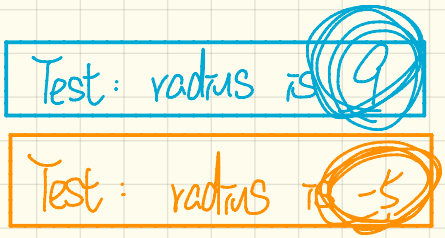
```
String lettGrade = "F";
if (score >= 80.0) {
  letterGrade = "A";
}
else if (score >= 70.0) {
  letterGrade = "B";
}
else if (score >= 60.0) {
  letterGrade = "C";
}
```

score 50

```
String lg = "";
if (s >= 80){
  lg = "A";
}else if (s >= 70){
  lg = "B";
}else if (s >= 60){
  lg = "C"
}else { lg = "F"; }
```


radius
invalid : radius < 0
valid : ! (invalid)
↳ ! (radius < 0)
↳ radius >= 0

Two Ways to Handling Errors




```java
public class ComputeArea {
  public static void main(String[] args) {
    System.out.println("Enter a radius value:");
    Scanner input = new Scanner(System.in);
    double radius = input.nextDouble();
    final double PI = 3.14159;
    if (radius < 0) {  /* condition of invalid inputs */
      System.out.println("Error: Negative radius value!");
    }
    else {  /* implicit:  !(radius < 0), or radius >= 0 */
      double area = radius * radius * PI;
      System.out.println("Area is " + area);
    }
  }
}
```

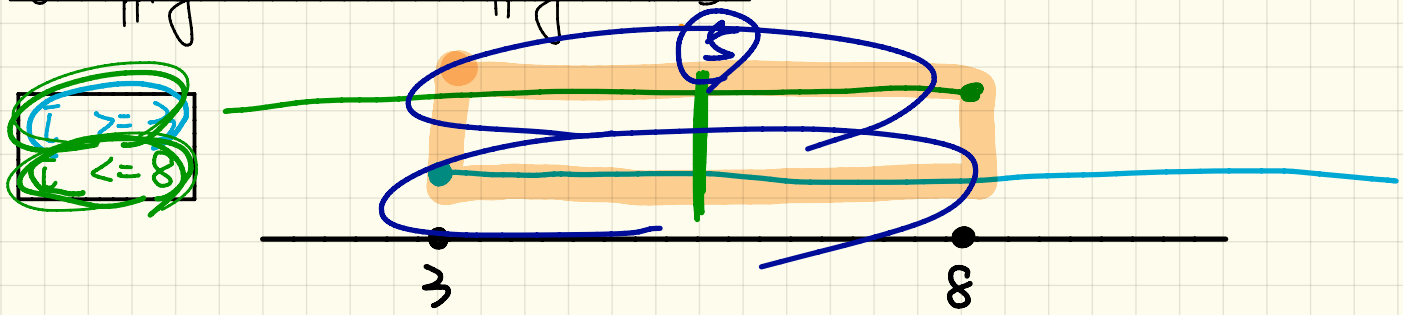
```java
public class ComputeArea2 {
  public static void main(String[] args) {
    System.out.println("Enter a radius value:");
    Scanner input = new Scanner(System.in);
    double radius = input.nextDouble();
    final double PI = 3.14159;
    if (radius >= 0) {  /* condition of valid inputs */
      double area = radius * radius * PI;
      System.out.println("Area is " + area);
    }
    else {  /* implicit:  !(radius >= 0), or radius < 0 */
      System.out.println("Error: Negative radius value!");
    }
  }
}
```

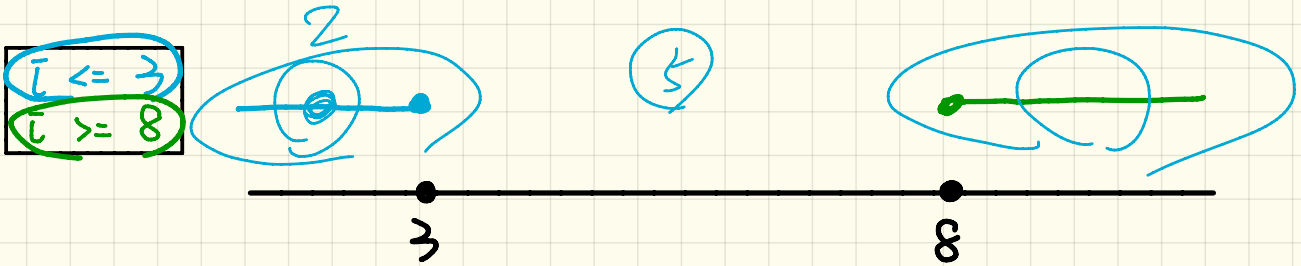
$-5 >= 0$ F

Wednesday January 23

Lecture 6

Overlapping vs. Non-Overlapping Intervals





Single If-Statement vs. Multiple If-Statements: Overlapping Conditions

a single i.s.

```
int i = 5;
if(i >= 3) {System.out.println("i is >= 3");}
else if(i <= 8) {System.out.println("i is <= 8");}
```

```
i is >= 3
```

```
int i = 5;
if(i >= 3) {System.out.println("i is >= 3");}
if(i <= 8) {System.out.println("i is <= 8");}
```

```
i is >= 3
i is <= 8
```

2 if statements.

Single If-Statement vs. Multiple If-Statements: Non-Overlapping Conditions

```
int i = 2;
if(i <= 3) {System.out.println("i is <= 3");}
else if(i >= 8) {System.out.println("i is >= 8");}
```

```
i is <= 3
```

```
int i = 2;
if(i <= 3) {System.out.println("i is <= 3");}
if(i >= 8) {System.out.println("i is >= 8");}
```

```
i is <= 3
```

Scope of variables : method

```
public static void main(String[] args) {
  int i = input.nextInt();
  System.out.println("i is " + i);
  if (i > 0) {
    i = i * 3; /* both use and re-assignment, why? */
  }
  else {
    i = i * -3; /* both use and re-assignment, why? */
  }
  System.out.println("3 * |i| is " + i);
}
```

sub-scope

subscope

Scope of Variables: Branches

```
public static void main(String[] args) {
    int i = input.nextInt();
    if (i > 0) {
        int j = i * 3; /* a new variable j */
        if (j > 10) { ... }
    }
    else {
        int j = i * -3; /* a new variable also called j */
        if (j < 10) { ... }
    }
}
```

Scope of Variables : Illegal Use of Variable from Other Branch

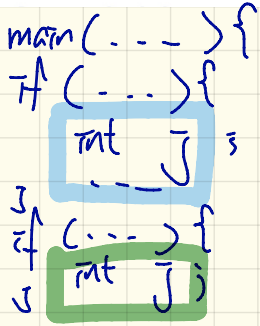
```
public static void main(String[] args) {
  int i = input.nextInt();
  if (i > 0) {
    int j = i * 3; /* a new variable j */
    if (j > 10) { ... }
  }
  else {
    int k = i * -3; /* a new variable also called j */
    if (j < k) { ... }
  }
}
```

Illegal: k is declared in a dif. sub-scope.

Illegal: j is declared in a different sub-scope.

Scope of Variables: Illegal Use of Variable Outside If-Statement

```
1  public static void main(String[] args) {
2    int i = input.nextInt();
3    if (i > 0) {
4      int j = i * 3; /* a new variable j */
5      if (j > 10) { ... }
6    }
7    else {
8      int j = i * -3; /* a new variable also called j */
9      if (j < 10) { ... }
10   }
11   System.out.println("i * j is " + i * j);
12 }
```



Compound If-Statement

Test 1: x = 5
Test 2: x = 10
Test 3: x = -2

```
1  int x = input.nextInt();
2  int y = 0;
3  if (x >= 0) {
4    System.out.println("x is positive");
5    if (x > 10) { y = x * 2; }
6    else if (x < 10) { y = x % 2; }
7    else { y = x * x; }
8  }
9  else { /* x < 0 */
10   System.out.println("x is negative");
11   if(x < -5) { y = -x; }
12 }
```

Truth Tables of Logical Operators

Negation (not)

P	!P
true	false
false	true

Conjunction (and)

P	q	P && q
false	false	false
false	true	
true	false	
true	true	true

Disjunction (or)

P	q	P || q
false	false	false
false	true	true
true	false	
true	true	

Example of Logical Operation: Negation

Test 1: 0
Test 2: -3
Test 3: 5

Operand op	!op
true	false
false	true

```
double radius = input.nextDouble();
boolean isPositive = radius > 0;
if (!isPositive) {/* not the case that isPositive is true */
System.out.println("Error: radius value must be positive.");
}
else {
System.out.println("Area is " + radius * radius * PI);
}
```

Example of Logical Operation: Conjunction

Test 1: age = 30
Test 2: age = 50
Test 3: age = 70

Left Operand `op1`	Right Operand `op2`	`op1 && op2`
true	*true*	*true*
true	*false*	*false*
false	*true*	*false*
false	*false*	*false*

```
int age = input.nextInt();
boolean isOldEnough = age >= 45;
boolean isNotTooOld = age < 65;
if (!isOldENough) { /* young */ }
else if (isOldEnough && isNotTooOld) { /* middle-aged */ }
else { /* senior */ }
```

Example of Logical Operation: Disjunction

Test 1: age = 70
Test 2: age = 15
Test 3: age = 40

Left Operand op1	Right Operand op2	op1 || op2
false	false	false
true	false	true
false	true	true
true	true	true

```
int age = input.nextInt();
boolean isSenior = age >= 65;
boolean isChild = age < 18
if (isSenior || isChild) { /* discount */ }
else { /* no discount */ }
```

70 40

T F

F || F

- Lab Test I (Week of Jan 28) 5%

~ different format

~ strict on syntax and type errors

~ guide, tutorial video, example tests

Monday January 28

Lecture 7

- Lab 2 Tutorial Videos 20 ~ 24
  - ~ Loops
  - ~ Debugger
- Quiz 2 Week of Feb 4
  - ~ guide

Logical Law: Negation of Relation Operation

Relation	Negation	Equivalence
i > j	!(i > j)	i <= j
i >= j	!(i >= j)	i < j
i < j	!(i < j)	i >= j
i <= j	!(i <= j)	i > j

Test 1: i = 17, j = 3

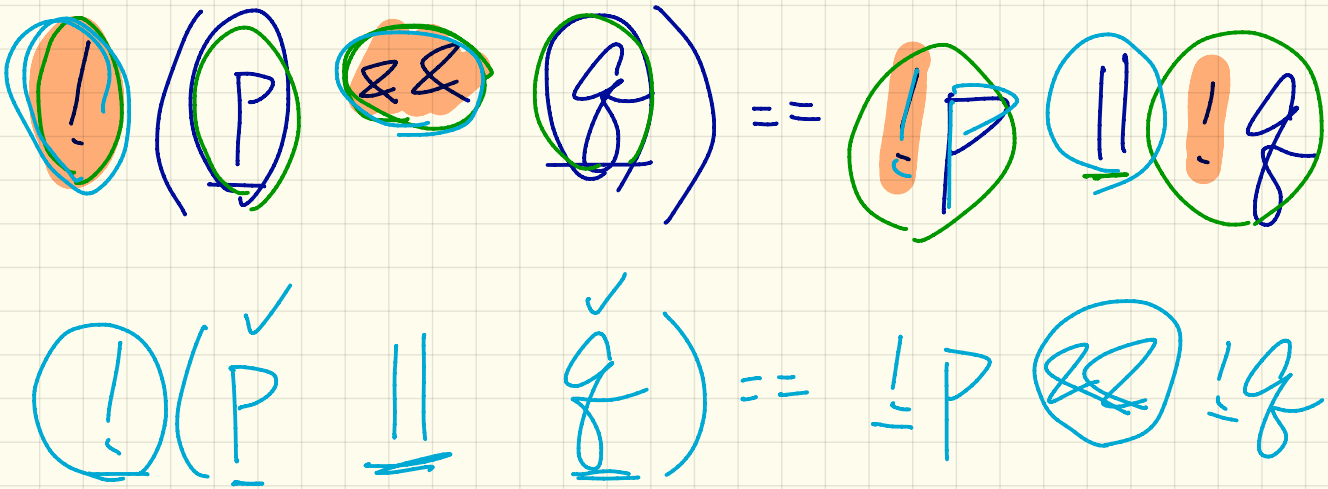
Test 2: i = -4, j = 13

```
if (i > j) {
   /* Action 1 */
}
else {   /* !(i > j)   */
   /* Action 2 */
}
```

equivalent to

```
if (i <= j) {
   /* Action 2 */
}
else {   /* !(i <= j)   */
   /* Action 1 */
}
```

boolean p;
boolean q;



Logical Laws: DeMorgan

B_1	B_2	! (B_1 && B_2)	! B_1 || ! B_2
true	true	false	false
true	false	[redacted]	[redacted]
false	true	[redacted]	[redacted]
false	false	[redacted]	[redacted]

B_1	B_2	! (B_1 || B_2)	! B_1 && ! B_2
true	true	[redacted]	[redacted]
true	false	[redacted]	[redacted]
false	true	[redacted]	[redacted]
false	false	[redacted]	[redacted]

De Morgan Law: Application I

```
if(0 <= i && i <= 10) { /* Action 1 */ }
else { /* Action 2 */ }
```

- **When** is *Action 2* executed? `i < 0 || i > 10`

```
if(i < 0 && false) { /* Action 1 */ }
else { /* Action 2 */ }
```

- **When** is *Action 1* executed? *false*
- **When** is *Action 2* executed? *true* (i.e., `i >= 0 || true`)

```
if(i < 0 && i > 10) { /* Action 1 */ }
else { /* Action 2 */ }
```

- **When** is *Action 1* executed? *false*
- **When** is *Action 2* executed? *true* (i.e., `i >= 0 || i <= 10`)

→ if (i<0 && false) {

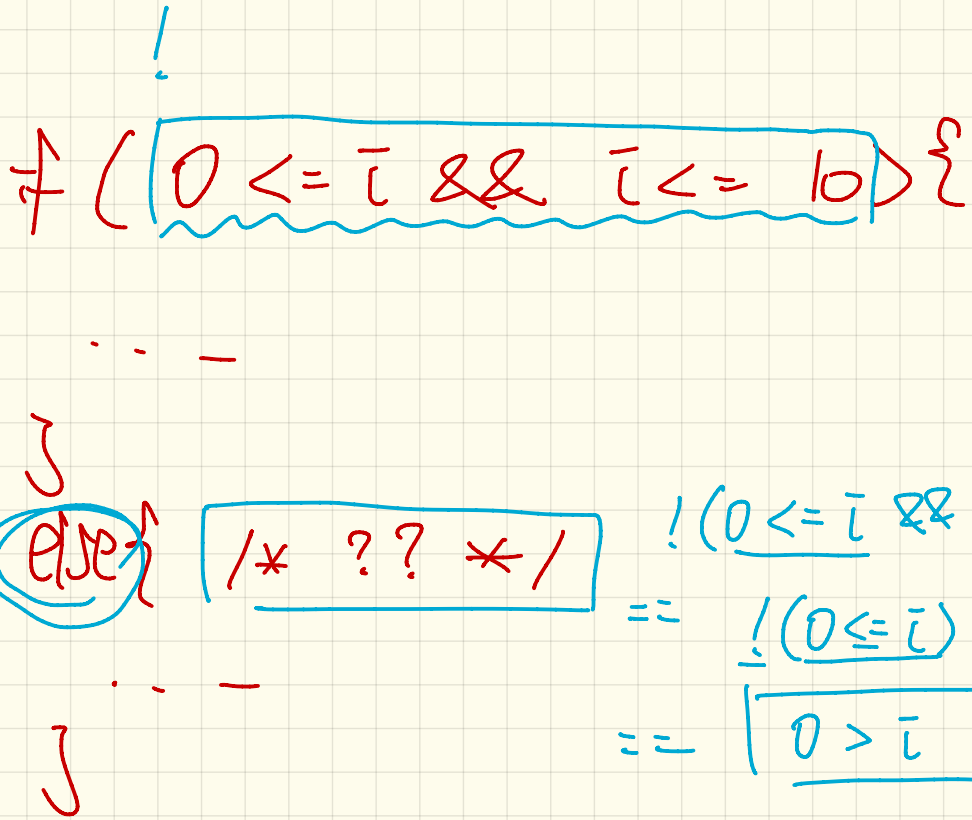
}
else{ /* ?? */ → !(i<0 && false)

}

== !(i<0) || !false

== i >= 0 || true

== true.


!
if (0 <= i && i <= 10) {
. . .
}
else {
/* ?? */
. . .
}
!(0 <= i && i<=10)
== !(0<=i) || !(i<=10)
== 0 > i || i > 10

De Morgan Law: Application 2

```
if(i < 0 || i > 10) { /* Action 1 */ }
else { /* Action 2 */ }
```

- **When** is *Action 2* executed?

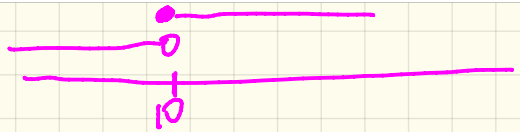
0 <= i && i <= 10

```
if(i < 0 || true) { /* Action 1 */ }
else { /* Action 2 */ }
```

- **When** is *Action 1* executed?
- **When** is *Action 2* executed?

```
if(i < 10 || i >= 10) { /* Action 1 */ }
else { /* Action 2 */ }
```

- **When** is *Action 1* executed?
- **When** is *Action 2* executed?



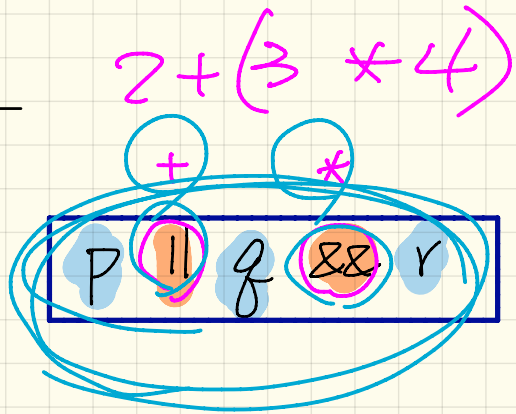
```
if (i < 0 || i > 10) {
    ...
}
else { /* ?? */
    ...
}
```

$!(i < 0 \;||\; i > 10)$

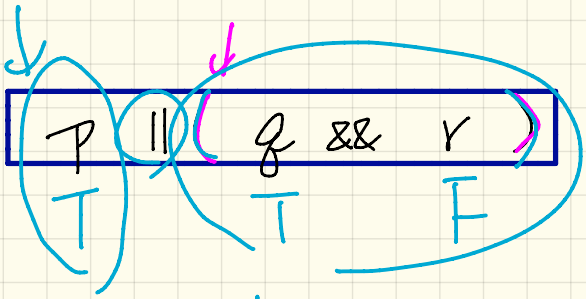
$== !(i < 0) \;\&\&\; !(i > 10)$

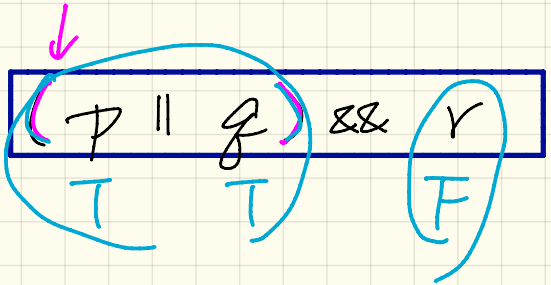
$== i >= 0 \;\&\&\; i <= 10$

Precedence of Logical Operators




```
boolean p = true;
boolean q = true;
boolean r = false;
```

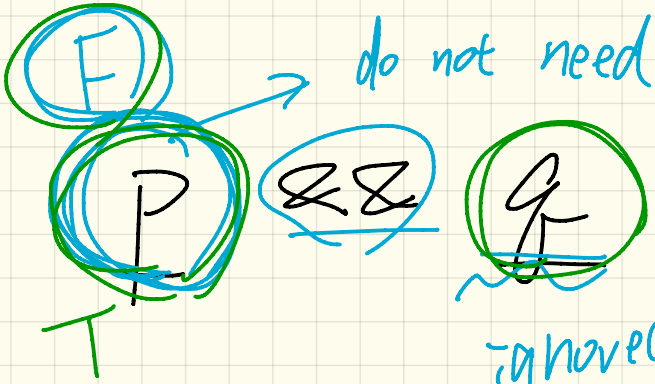




T - p && q

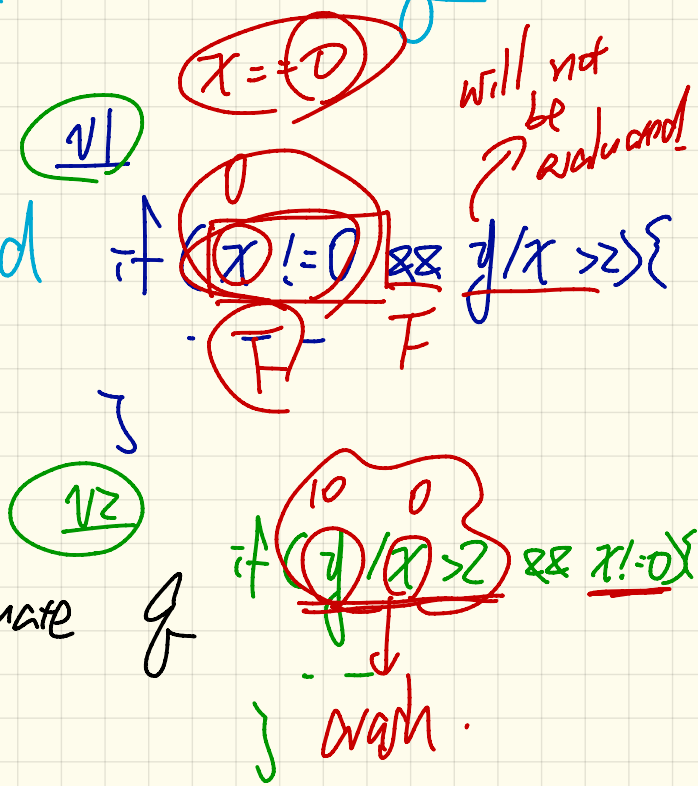
T - q && p

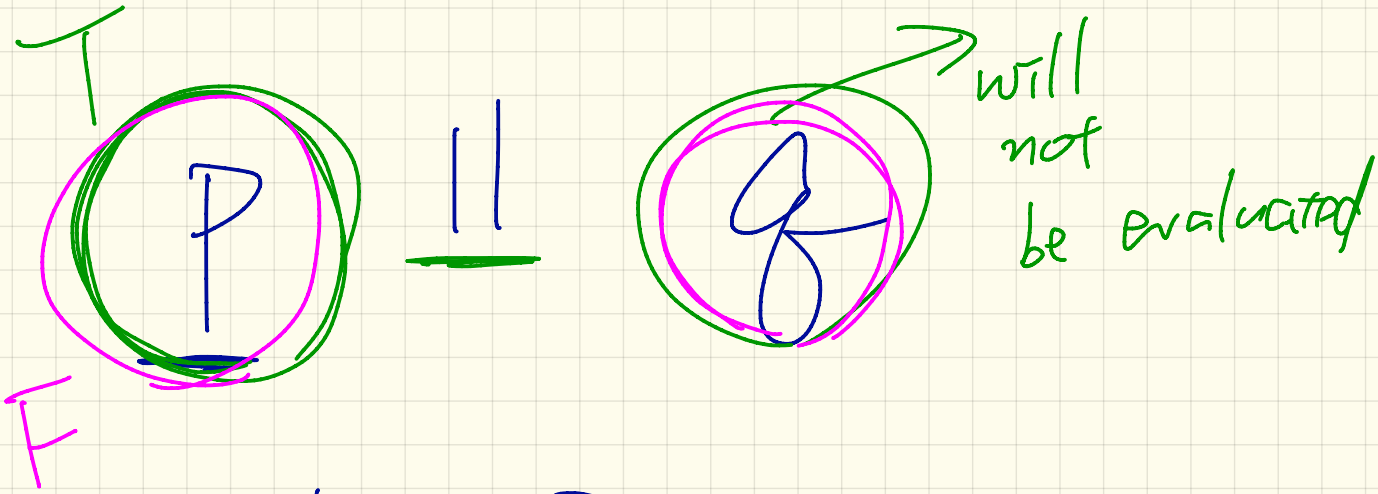
F



runtime

1. Evaluate P
2. "If necessary", evaluate Q





1. Evaluate P

2. If necessary, evaluate g.

Wednesday January 30

Lecture 8

- Lab 2 released
- Quiz 2 guide released
- Lab Sessions 09/10 special agenda
  to be sent tomorrow

Short-Circuit Evaluation: &&

Left Operand op1	Right Operand op2	op1 && op2
true	true	true
true	false	false
false	true	false
false	false	false

```
System.out.println("Enter x:");
int x = input.nextInt();
System.out.println("Enter y:");
int y = input.nextInt();
if(x != 0 && y / x > 2) {
    System.out.println("y / x is greater than 2");
}
else { /* !(x != 0 && y / x > 2) == (x == 0 || y / x <= 2) */
    if(x == 0) {
        System.out.println("Error: Division by Zero");
    }
    else {
        System.out.println("y / x is not greater than 2");
    }
}
```

y / x > 2 && x != 0

crash

Test Case:
x = 0
y = 10

Test Case:
x = 5
y = 10

Short-Circuit Evaluation: ||

Left Operand op1	Right Operand op2	op1 || op2
false	false	false
true	false	true
false	true	true
true	true	true

```java
System.out.println("Enter x:");
int x = input.nextInt();
System.out.println("Enter y:");
int y = input.nextInt();
if(x == 0 || y / x > 2) {
    if(x == 0) {
        System.out.println("Error: Division by Zero");
    }
    else {
        System.out.println("y / x is greater than 2");
    }
}
else { /* !(x == 0 || y / x > 2) == (x != 0 && y / x <= 2) */
    System.out.println("y / x is not greater than 2");
}
```

x==0 || y/x>2

y/x>2 || x==0

10/0>2 → crash.

Test Case:
x = 0
y = 10

Test Case:
x = 5
y = 10

Short-Circuit Evaluation: Common Error

Test Case:
$x = 0$
$y = 10$

crash when $x == 0$

Short-Circuit Evaluation is not exploited: crash when `x == 0`

```
if (y / x > 2 && x != 0) {
  /* do something */
}
else {
  /* print error */ }
```

crash when $x == 0$

Short-Circuit Evaluation is not exploited: crash when `x == 0`

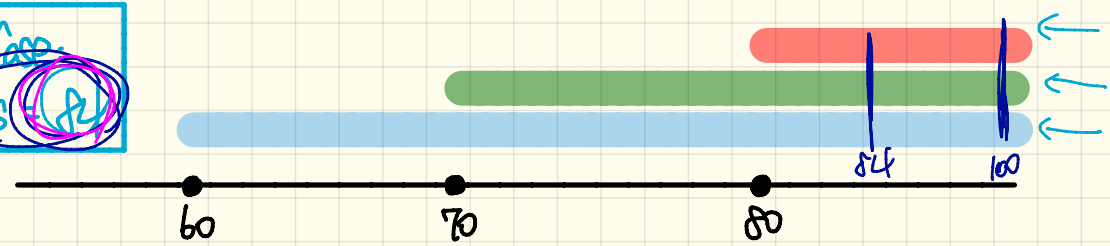
```
if (y / x <= 2 || x == 0) {
  /* print error */
}
else {
  /* do something */ }
```

Common Error: Overlapping Conditions in Multiple If-Statements

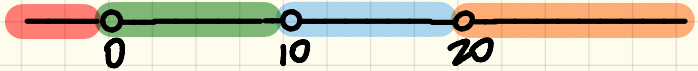
```java
if (marks >= 80) {
  System.out.println("A");
}
if (marks >= 70) {
  System.out.println("B");
}
if (marks >= 60) {
  System.out.println("C");
}
else {
  System.out.println("F");
}
```

```java
if (marks >= 80) {
  System.out.println("A");
}
else if (marks >= 70) {
  System.out.println("B");
}
else if (marks >= 60) {
  System.out.println("C");
}
else {
  System.out.println("F");
}
```

Test Case: marks = 84



Exercise: Overlapping Conditions



Does this program always print exactly one line?

```
if (x < 0)  { println("x < 0"); }
if (0 <= x && x < 10)  { println("0 <= x < 10"); }
if (10 <= x && x < 20)  { println("10 <= x < 20"); }
if (x >= 20)  { println("x >= 20"); }
```

multiple if → non-overlapping

Does this program always print exactly one line?

```
if (x < 0)  { println("x < 0"); }
else if (0 <= x && x < 10)  { println("0 <= x < 10"); }
else if (10 <= x && x < 20)  { println("10 <= x < 20"); }
else if (x >= 20)  { println("x >= 20"); }
```

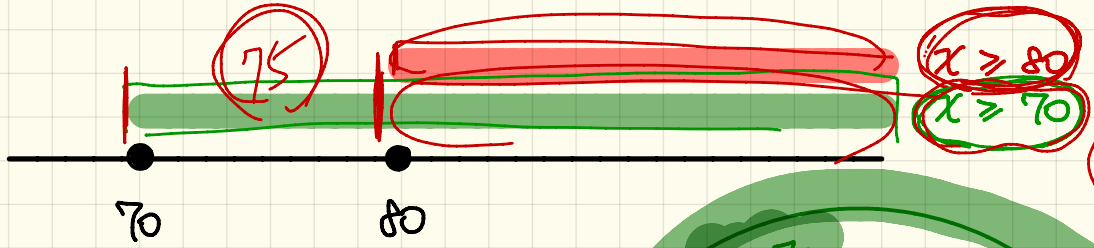
single if's non-overlapping

This simplified version is equivalent:

```
if (x < 0)  { println("x < 0"); }
else if (x < 10)  { println("0 <= x < 10"); }
else if (x < 20)  { println("10 <= x < 20"); }
else  { println("x >= 20"); }
```

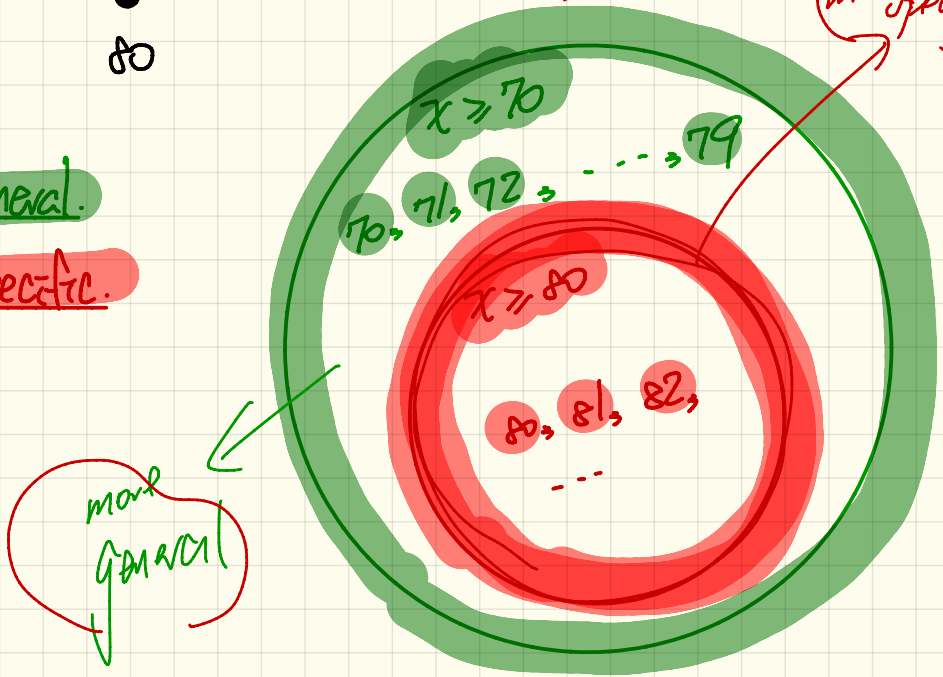
!(x<0) == x>=0

Overlapping Conditions: General vs. Specific

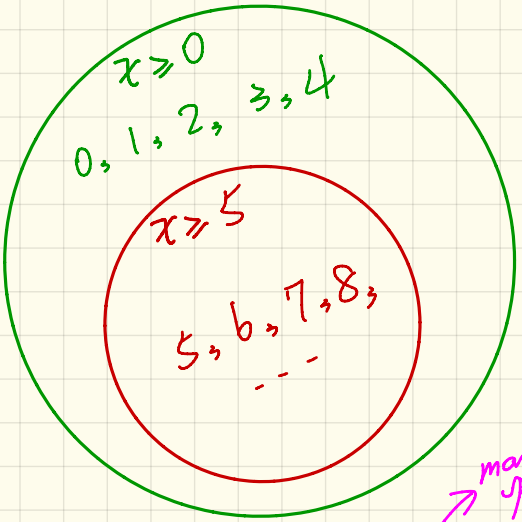


$x \geq 70$ is **more general.**

$x \geq 80$ is **more specific.**

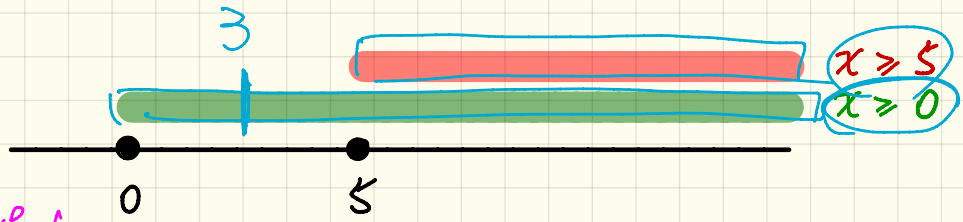


Overlapping Conditions in a Single If-Statement



$x \geq 0$ is more general.

$x \geq 5$ is more specific.



Test Case: $x = 5$

If we have a single if statement, then having this order

more specific

```
if (x >= 5) { System.out.println("x >= 5"); }
else if (x >= 0) { System.out.println("x >= 0"); }
```

is different from having this order

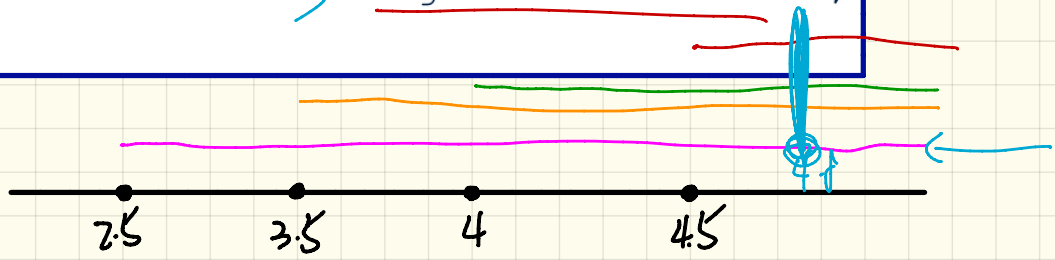
more general

```
if (x >= 0) { System.out.println("x >= 0"); }
else if (x >= 5) { System.out.println("x >= 5"); }
```

Common Error: General Condition comes before Specific Condition

```
if (gpa >= 2.5) {
  graduateWith = "Pass";
}
else if (gpa >= 3.5) {
  graduateWith = "Credit";
}
else if (gpa >= 4) {
  graduateWith = "Distinction";
}
else if (gpa >= 4.5) {
  graduateWith = "High Distinction" ;
}
```

Test Case: gpa = 4.8



Common Error: Missing Braces

Confusingly, braces can be omitted if the block contains a *single* statement.

```
final double PI = 3.1415926;
Scanner input = new Scanner(System.in);
double radius = input.nextDouble();
if (radius >= 0)
  System.out.println("Area is " + radius * radius * PI);
```

Your program will *misbehave* when a block is supposed to execute *multiple statements*, but you forget to enclose them within braces.

```
final double PI = 3.1415926;
Scanner input = new Scanner(System.in);
double radius = input.nextDouble();
double area = 0;
if (radius >= 0)
  area = radius * radius * PI;
System.out.println("Area is " + area);
```

Test Case:
radius = -3

Common Error: Misplaced Semicolon

Test Case:
radius = -4

Semicolon (;) in Java marks *the end of a statement* (e.g., assignment, `if` statement).

```
if (radius >= 0);
{
   area = radius * radius * PI;
   System.out.println("Area is " + area);
}
```

start of something unconditional

This program will calculate and output the area even when the input radius is *negative*, why? Fix?

Common Error: Variable Not Properly Reassigned

= "" → "Fail"

```
String graduateWith = "";
if (gpa >= 4.5) {
graduateWith = "High Distinction" ; }
else if (gpa >= 4) {
graduateWith = "Distinction"; }
else if (gpa >= 3.5) {
graduateWith = "Credit"; }
else if (gpa >= 2.5) {
graduateWith = "Pass"; }
```

Test Case: gpa = 1.5

else

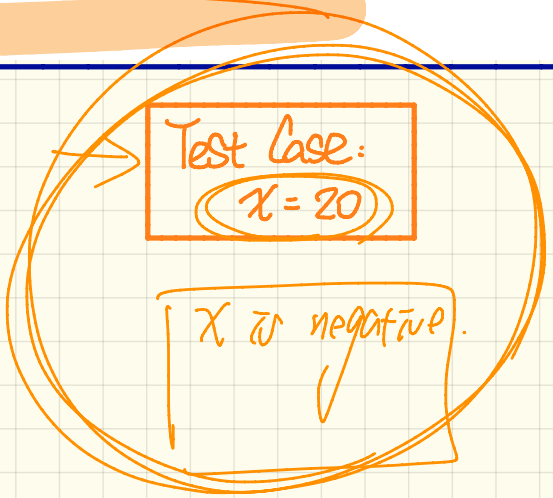
```
else {
    gw = "Fail"
}
```

Common Error: Ambiguous "else"

```
if (x >= 0) {
    if (x > 100) {
        System.out.println("x is larger than 100");
    }
else {
    System.out.println("x is negative");
}
```

Test Case:
x = 20

no output



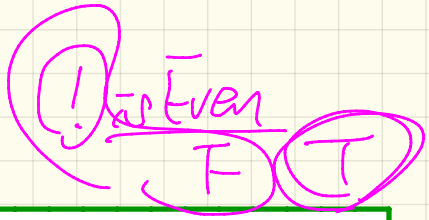
Common Pitfall

```
boolean isEven;
if (number % 2 == 0) {
  isEven = true;
}
else {
  isEven = false;
}
```

==

```
isEven = number % 2 == 0;
```

```
if (isEven == false) {
  System.out.println("Odd Number");
}
else {
  System.out.println("Even Number");
}
```



Monday February 4

Lecture 9

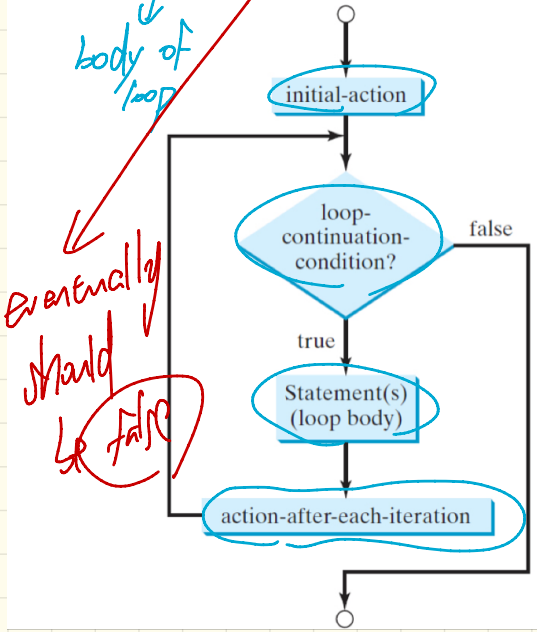
for-Loop: flow chart

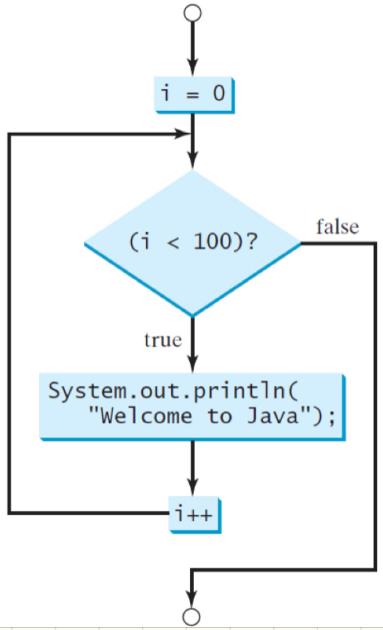
stay condition

update at the end

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

body of loop





eventually should be false

101 times

↳ 100 times (T)

↳ 101st (F)

⇒ How many times is "i<100" checked?

⇒ How many times is "println(...)" executed?

for-Loop : Tracing

True → stay in loop
False → exit from loop

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

i < 100
0
:
99
100

100 < 100 F

i	i < 100	Enter/Stay Loop?	Iteration	Actions
0	0 < 100	True	1	print, i ++
1	1 < 100	True	2	print, i ++
2	2 < 100	True	3	print, i ++
...				
99	99 < 100	True	100	print, i ++
100	100 < 100	False	–	–

How many times is "i<100" checked?
How many times is "println(...)" executed?

[0, 99] 100 times execute body of loop

for-Loop: Alternative Syntax

executed once

executed at the end of iteration.

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

- The *"initial-action"* is executed *only once*, so it may be moved right before the `for loop`.
- The *"action-after-each-iteration"* is executed repetitively to *make progress*, so it may be moved to the end of the `for loop` body.

So the above for-loop may be re-written as:

```
int i = 0;
for( ; i < 100; ) {
   println("...");
   i++;
}
```

for-Loop : Exercise (1)

Count = 2 * i - 1

(1) / 1

Compare the behaviour of this program

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

and this program

```
for (int count = 1; count < 201; count += 2) {
  System.out.println("Welcome to Java!");
}
```

Count = 2 * i - 1

1 → 2*1 - 1

3 → 2*2 - 1

[1, 3, 5, ..., 199]

- Are the outputs same or different?

[0, 99]

Count	Count < 100	Iteration (i)
0	T	1
1	T	2
⋮		
99	T	?
100	100 < 100 F	99 + 1 = 100

Count	Count < 201	Iteration (i)
1	1 < 201 T	1
3	3 < 201 T	2
5	5 < 201 T	3
⋮		
199	199 < 201 T	?
201	201 < 201 F	100

Count = 2*i - 1

199 = 2*100 - 1

for-Loop : Exercise (2)

Welcome ... 0
Welcome ... 1
Welcome ... 2
⋮
99

Compare the behaviour of this program

```
int count = 0;
for (; count < 100; ) {
  System.out.println("Welcome to Java " + count + "!");
  count ++; /* count = count + 1; */
}
```

[0, 99] 100

and this program

```
int count = 1;
for (; count <= 100; ) {
  System.out.println("Welcome to Java " + count + "!");
  count ++; /* count = count + 1; */
}
```

[1, 100] 100

Are the outputs same or different?

# of iterations : 100

Welcome ... 1
Welcome ... 2
⋮
Welcome ... 100

for-Loop : Exercise (3)

Compare the behaviour of the following three programs:

```
for (int i = 1; i <= 5 ; i ++) {
  System.out.print(i); }
```

Output: 12345

```
int i = 1;
for ( ; i <= 5 ; ) {
  System.out.print(i);
  i ++; }
```

Output: 12345

```
int i = 1;
for ( ; i <= 5 ; ) {
  i ++;
  System.out.print(i); }
```

i	i <= 5	i++	print
1	1<=5 T	2	2
2	2<=5 T	3	3
5	5<=5 T	6	6
6	6<=5 F		

Output: 23456

while-Loop: flow chart

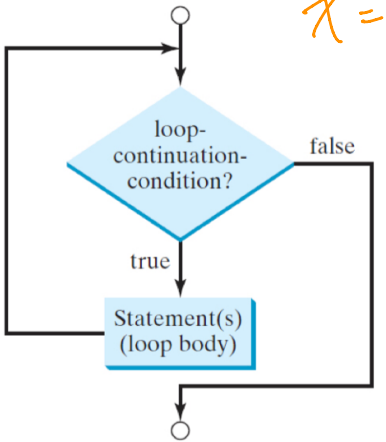
```
int count = 0;
while (count < 100) {
  System.out.println("Welcome to Java!");
  count ++; /* count = count + 1; */
}
```

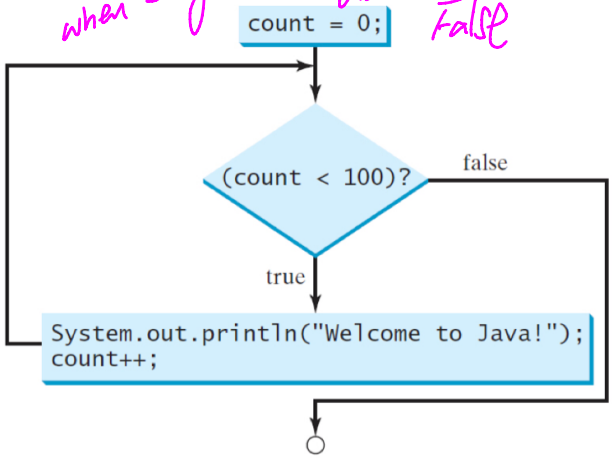
stay condition

for (___ ; ___ ; ___)

while (___)

x = y + 1 when stay condition evaluates to false





- How many times is "i<100" checked?
- How many times is "println(...)" executed?

while-Loop : Tracing

[3, 102] 102 − 3 + 1 = 100 # of iterations

```
int j = 3;
while (j < 103) {
  System.out.println("Welcome to Java!");
  j ++; /* j = j + 1; */ }
```

J − 2 = iteration

j	j < 103	Enter/Stay Loop?	Iteration	Actions
3	3 < 103	True	1	print, j ++
4	4 < 103	True	2	print, j ++
5	5 < 103	True	3	print, j ++
...				
102	102 < 103	True	100	print, j ++
103	103 < 103	False	–	–

How many times is "i<100" checked?
How many times is "println(...)" executed?

while-Loop : Exercise (1)

Compare the behaviour of this program

```
int count = 0;
while (count < 100) {
  System.out.println("Welcome to Java!");
  count ++; /* count = count + 1; */
}
```

[0, 99]
100 times

and this program

```
int count = 1;
while (count <= 100) {
  System.out.println("Welcome to Java!");
  count ++; /* count = count + 1; */
}
```

[1, 100]
100 times

count	count < 100	Iteration (i)

count	count <= 100	Iteration (i)

while-Loop : Exercise (2)

Welcome ... 0

Compare the behaviour of this program

```
int count = 0;
while (count < 100) {
 System.out.println("Welcome to Java " + count + "!");
 count ++; /* count = count + 1; */
}
```

and this program

```
int count = 1;
while (count <= 100) {
 System.out.println("Welcome to Java " + count + "!");
 count ++; /* count = count + 1; */
}
```

Are the outputs same or different?

Welcome ... 1

Compound Loop : Exercise (1)

count =+ 2 X
count = count + 2
count += 2

```
System.out.println("Enter a radius value:");
double radius = input.nextDouble();
while (radius >= 0) {
    double area = radius * radius * 3.14;
    System.out.println("Area is " + area);
    System.out.println("Enter a radius value:");
    radius = input.nextDouble();
}
System.out.println("Error: negative radius value.");
```

Test 1:
radius = -3

Test 2:
radius = 2
radius = -3

Area is - -
Error: neg. radius value

Test 3:
radius = 2
radius = 3

Wednesday February 6

Lecture 10

Compound Loop : Exercise (2.1)

```
1  System.out.println("Enter a radius value:");
2  double radius = input.nextDouble();
3  boolean isPositive = radius >= 0;
4  while (isPositive) {
5    double area = radius * radius * 3.14;
6    System.out.println("Area is " + area);
7    System.out.println("Enter a radius value:");
8    radius = input.nextDouble();
9    isPositive = radius >= 0; }
10 System.out.println("Error: negative radius value.");
```

```
1  System.out.println("Enter a radius value:");
2  double radius = input.nextDouble();
3  boolean isNegative = radius < 0;
4  while (!isNegative) {
5    double area = radius * radius * 3.14;
6    System.out.println("Area is " + area);
7    System.out.println("Enter a radius value:");
8    radius = input.nextDouble();
9    isNegative = radius < 0; }
10 System.out.println("Error: negative radius value.");
```

Test 1:
radius = -3

Test 2:
radius = 2
radius = -3

Test 3:
radius = 2
radius = 3

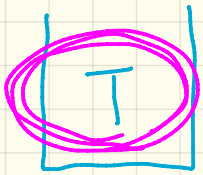
Compound Loop: Exercise (2.2)

Q: What if we delete the update at Line 9?

```
1  System.out.println("Enter a radius value:");
2  double radius = input.nextDouble();
3  boolean isPositive = radius >= 0;
4  while (isPositive) {
5      double area = radius * radius * 3.14;
6      System.out.println("Area is " + area);
7      System.out.println("Enter a radius value:");
8      radius = input.nextDouble();
9  }
10 System.out.println("Error: negative radius value.");
```

-3*-3*3.14

2*2*3.14



infinite loop.

isPositive

Test 2:

radius = 2

radius = -3

Console:

for-loop ⟷ while-loop

- To convert a `while` loop to a `for` loop, leave the initialization and update parts of the `for` loop empty.

```
while (B) {
  /* Actions */
}
```

is equivalent to:

```
for( ; B ; ) {
  /* Actions */
}
```

- To convert a `for` loop to a `while` loop, move the initialization part immediately before the `while` loop and place the update part at the end of the `while` loop body.

```
for(int i = 0 ; B ; i ++ ) {
  /* Actions */
}
```

is equivalent to:

```
int i = 0;
while (B) {
  /* Actions */
  i ++;
}
```

Stay Condition vs. Exit Condition

p q

!(p && q) == !p || !q

- When does the loop exit (i.e., stop repeating Action 1)?

```
while (p && q) { /* Action 1 */ }
```

!(p && q) : exit condition

exit condition = !(stay condition)

- When does the loop exit (i.e., stop repeating Action 2)?

```
while (p || q) { /* Action 2 */ }
```

!(p || q) : exit condition

!(p || q) == !p && !q

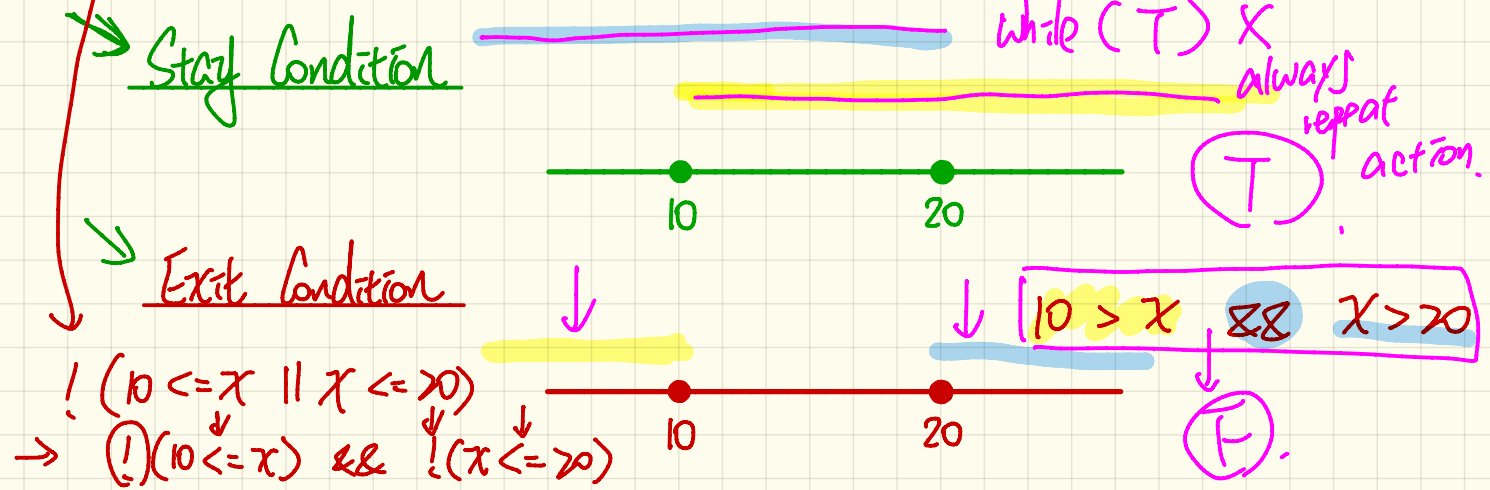
Stay Condition vs. Exit Condition: Exercise

Consider the following loop:

```
int x = input.nextInt();
while(10 <= x || x <= 20) {
  /* body of while loop */
}
```

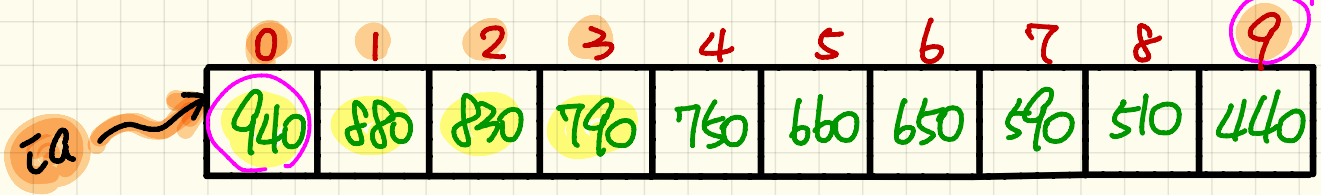
??

- It compiles, but has a logical error. Why?



Array of Integers (1)

No pattern on stored values



Declaration and Initialization: Approach 1 (Initializer)

```
int[] ia = {940, 880, 830, 790, 750, 660, 650,
            590, 510, 4440};
```

(int[] — type)

Declaration and Initialization: Approach 2 (Assignments)

```
int[] ia = new int[10];   // ← an array of int of size 10
ia[0] = 940; ia[1] = 880; ... ;
ia[9] = 4440;
```

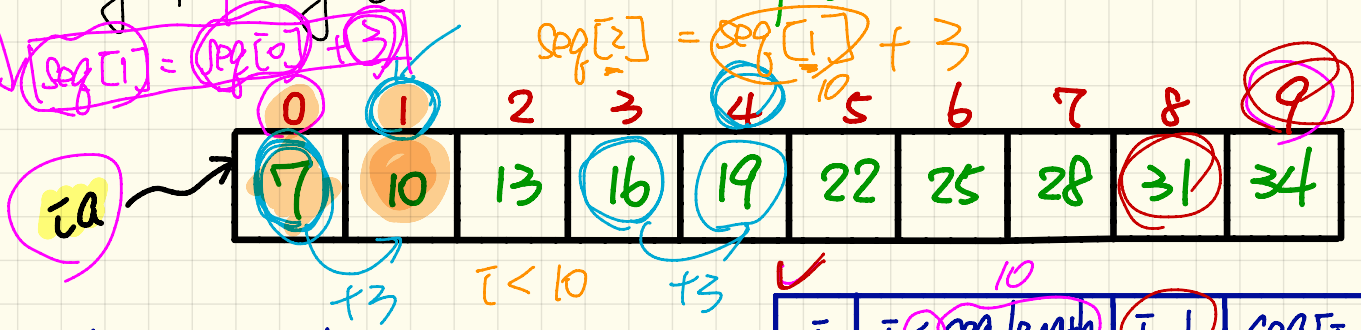
ia.length - 1 = 9

Monday February 11

Lecture 11

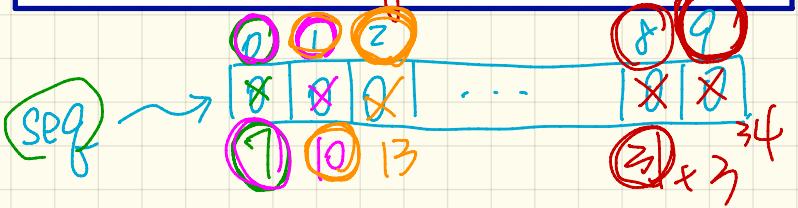
Array of Integers (2)

there is pattern on stored values



Declaration and Initialization:

```
int[] seq = new int[10];
seq[0] = 7;
for(int i = 1; i < seq.length; i++) {
  seq[i] = seq[i - 1] + 3;
}
```



i	i < seq.length	i-1	seq[i-1]
1	1<10 (T)	0	7
2	2<10 (T)	1	10
⋮	⋮	⋮	⋮
9	9<10 (T)	8	⋮
10	10<10 (F)	9	31

```
int[] a = new int[?];
String[] names = new String[?];
```

Re-assignment of a slot:

```
a[2] = 8;
```

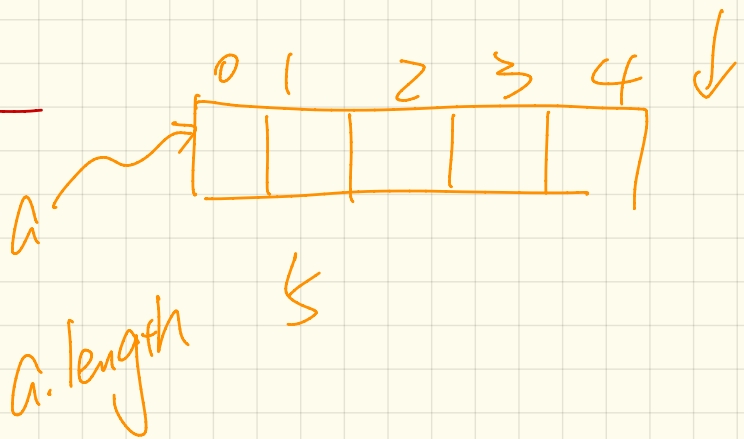
Accessing the value stored in a slot:

```
println( a[3] )
```

for (int i = 0; i <= nums.length; i++) {

[a[i]

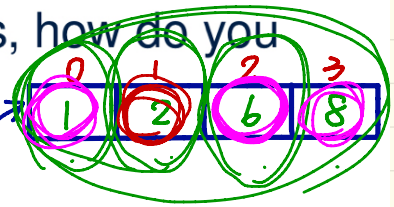
]



Computational Problem: Average

Problem: Given an array `numbers` of integers, how do you print its average?

e.g., Given array {1, 2, 6, 8}, print `4.25`.




```
int sum = 0;
for(int i = 0; i < numbers.length; i ++) {
    sum += numbers[i];
}
double average = (double) sum / numbers.length;
System.out.println("Average is " + average);
```

i	i<ns.length
0	0<4 T
1	1<4 T

Test Case 1

int[] numbers = {1, 2, 6, 8}

Test Case 2

int[] numbers = {}

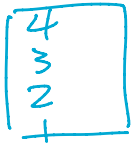
division by zero

numbers.length

Computational Problem: Printing Backwards

Problem: Given an array `numbers` of integers, how do you print its contents backwards?

e.g., Given array $\{1, 2, 3, 4\}$, print `4 3 2 1`.

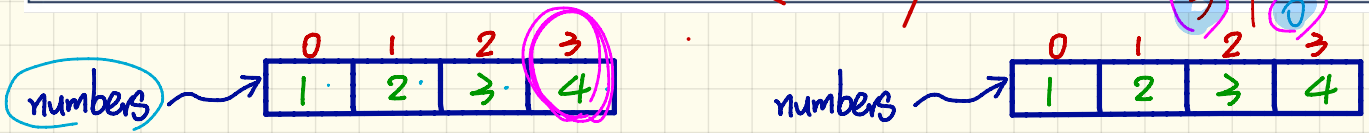


Solution 1: Change bounds and updates of loop counter.

```
for(int i = numbers.length - 1; i >= 0; i --) {
  System.out.println(numbers[i]);
}
```

Solution 2: Change indexing.

```
for(int i = 0; i < names.length; i ++) {
  System.out.println(numbers[ names.length - i - 1 ]);
}
```

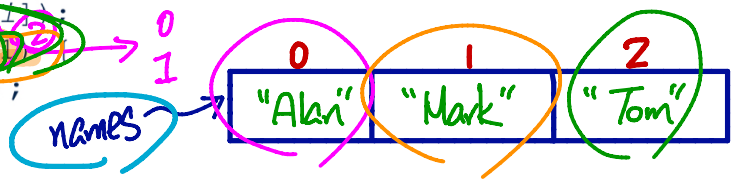


Wednesday February 13

Lecture 12

Computational Problem: Printing a Comma-Separated List

```
System.out.print("Names:")
for(int i = 0; i < names.length; i ++) {
    System.out.print(names[i]);
    if (i < names.length - 1)
        System.out.print(", ");
}
System.out.println(".");
```



Alan, Mark, Tom.

print vs. println

Console

Names: Alan, Mark, Tom.

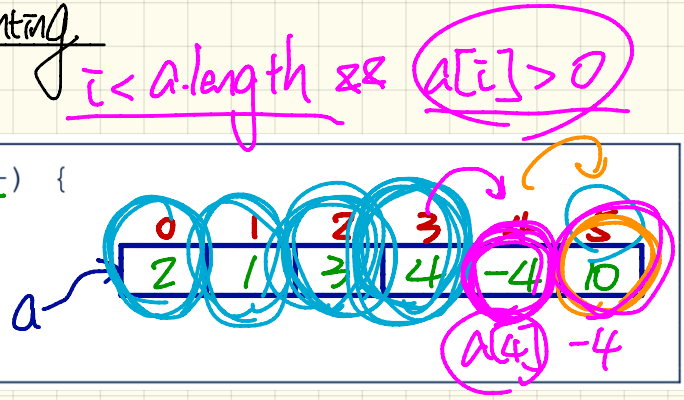
i	i < names.length		names[i]	i < names.length - 1	
0	0 < 3	T	"Alan"	0 < 2	T
1	1 < 3	T	"Mark"	1 < 2	T
2	2 < 3	T	"Tom"	2 < 2	F
3	3 < 3	F			

Computational Problem : Conditional Printing

```
for(int i = 0; i < a.length; i ++) {
    if (a[i] > 0) {
        System.out.println(a[i]);
    }
}
```

b

i < a.length && a[i] > 0




```
if( i < a.length-1){
    print (", ");
}
```

Console

```
2
1
3
4
10
```

i	i < a.length	a[i]	a[i] > 0
0	0<6 T	2	2>0 T
1		1	1>0 T
2		3	3>0 T
3		4	4>0 T
4	4<6 T	-4	-4>0 F
5	5<6	10	10>0 T

Computational Problem : Finding Maximum

max = a[i]

```
1 int max = a[0];
2 for(int i = 0; i < a.length; i ++) {
3   if (a[i] > max) { max = a[i]; }
4 }
5 System.out.println("Maximum is " + max);
```

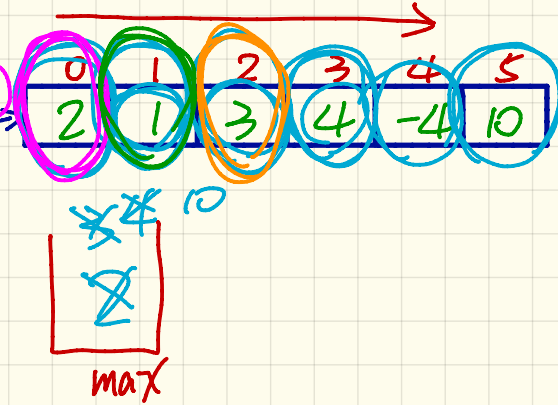
a[0] > a[0] → F - always

b

i	i < a.length	a[i]
0	0 < 6 T	a[0] > 2 F
1	1 < 6 T	a[1] > 2 F
2	2 < 6 T	a[2] > 2 T

the very 1st iteration always by passes the body of statement

max



Computational Problem: Finding Maximum

Q: What if we change the initialization in **L1** to `int max = 0`?

```
1  int max = a[0];
2  for(int i = 0; i < a.length; i ++) {
3    if (a[i] > max) { max = a[i]; }
4  }
5  System.out.println("Maximum is " + max);
```

0 3 ✗ ✗ ✗ 0 ✗ logical error

i	i < a.length	a[i]	
0	0<3 T	-3	-3 > 0 F
1	1<3 T	-4	-4 > 0 F
2	2<3 T	-1	-1 > 0 F

a → [0: -3] [1: -4] [2: -1]

0 → [0] max

Computational Problem : Finding Maximum

Q: What if we change the initialization in **L2** to `int i = 1`?

```
1  int max = a[0];
2  for(int i = 0; i < a.length; i++) {
3    if (a[i] > max) { max = a[i]; }
4  }
5  System.out.println("Maximum is " + max);
```

position 0 is already handled

max = a[2]

46

i	i < a.length	a[i]
1		-2 > 23 F
2		46 > 23 T

a → [23, -2, 46] (0, 1, 2)

max: 46

```
1  int max = a[0];
2  for(int i = 0; i < a.length; i ++) {
3     if (a[i] > max) {  max = a[i]; }
4  }
5  System.out.println("Maximum is " + max);
```

a.length - 1

i+1

logically incorrect though even it compiles

i

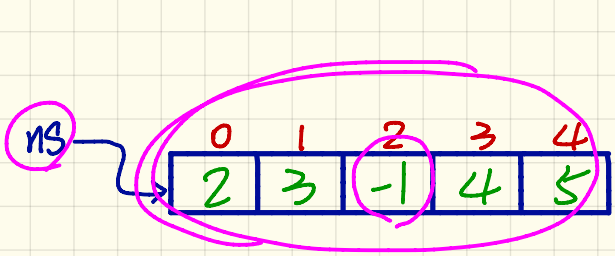
last iteration

i: a.length - 1

a[(a.length - 1) + 1]

a.length

i



answer

i

boolean allPositive =

ns[0] > 0
&&
ns[1] > 0
&&
ns[2] > 0
&&
ns[3] > 0
&&
ns[4] > 0

Computational Problem: Are all numbers positive?

```
1  int[] ns = {2, 3, -1, 4, 5};
2  boolean soFarOnlyPosNums = true;
3  int i = 0;
4  while (i < ns.length) {
5    soFarOnlyPosNums = soFarOnlyPosNums && (ns[i] > 0);
6    i = i + 1;
7  }
```

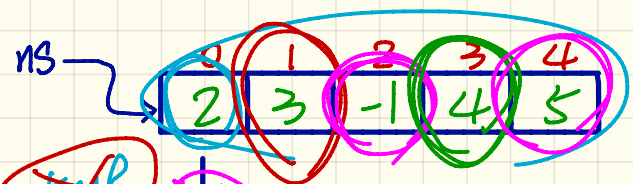
Version 1

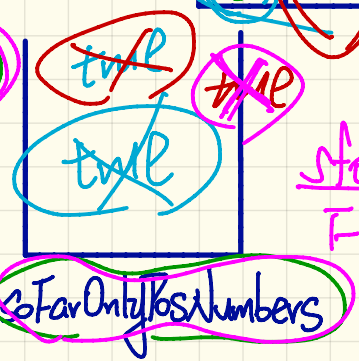
soFarOnlyPosNums = soFarOnlyPosNums && (ns[i] > 0);

soFarOnlyPosNums = soFarOnlyPosNums && (ns[0] > 0);

i	i < ns.length	ns[i] > 0
0	0 < 5 T	T
1	1 < 5 T	T
2	2 < 5 T	T
3	3 < 5 T	F

soFarOnlyPosNums = soFarOnlyPosNums && a[3] > 0;





soFarOnlyPosNums = soFarOnlyPosNums && a[2] > 0
-1
F

Monday February 25

Lecture 13

- Lab Test 2

Computational Problem: Are all numbers positive?

Is there at least one positive?

Four possible solutions (`posNumsSoFar` is initialized as *true*):

1. Scan the entire array and accumulate the result.

```
for (int i = 0; i < ns.length; i ++) {
  posNumsSoFar = posNumsSoFar && ns[i] > 0; }
```

2. Scan the entire array but the result is **not** accumulative.

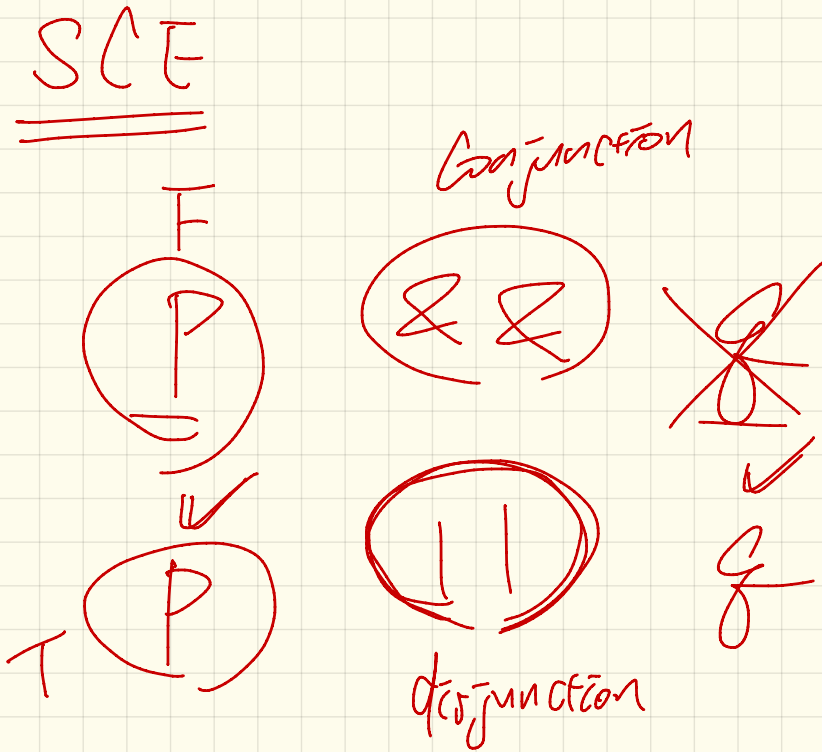
```
for (int i = 0; i < ns.length; i ++) {
  posNumsSoFar = ns[i] > 0; }  /* Not working.  Why?  */
```

3. The result is accumulative until the early exit point.

```
for (int i = 0; posNumsSoFar && i < ns.length; i ++) {
  posNumsSoFar = posNumsSoFar && ns[i] > 0; }
```

4. The result is **not** accumulative until the early exit point.

```
for (int i = 0; posNumsSoFar && i < ns.length; i ++) {
  posNumsSoFar = ns[i] > 0; }
```

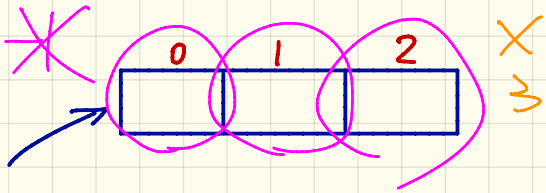

SCE
F
P
P
T
Conjunction
&&
||
disjunction

Short-Circuit and Array Indexing

```
1  Scanner input = new Scanner(System.in);
2  System.out.println("How many integers?");
3  int howMany = input.nextInt();
4  int[] ns = new int[howMany];
5  for(int i = 0; i < howMany; i ++) {
6    System.out.println("Enter an integer");
7    ns[i] = input.nextInt(); }
8  System.out.println("Enter an index:");
9  int i = input.nextInt();
10 if(ns[i] % 2 == 0) {
11   System.out.println("Element at index " + i + " is even."); }
12 else { /* Error ∵ ns[i] is odd */ }
```

Test Case 1: -1

Test Case 2: 3



Short-Circuit and Array Indexing

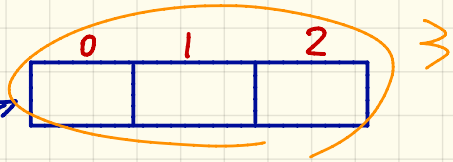
```
1  Scanner input = new Scanner(System.in);
2  System.out.println("How many integers?");
3  int howMany = input.nextInt();
4  int[] ns = new int[howMany];
5  for(int i = 0; i < howMany; i ++) {
6    System.out.println("Enter an integer");
7    ns[i] = input.nextInt(); }
8  System.out.println("Enter an index:");
9  int i = input.nextInt();
10 if(0 <= i && i < ns.length && ns[i] % 2 == 0) {
11   println(ns[i] + " at index " + i + " is even."); }
12 else { /* Error: invalid index or odd ns[i] */ }
```

Test Case 1: -1

Test Case 2: 3

Use of &&

valid conditions



0 <= -1 guard

F

0 <= 3 T

3 < 3 F

Short-Circuit and Array Indexing

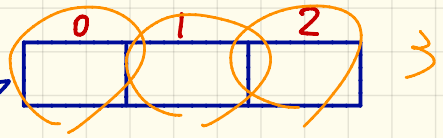
```
1  Scanner input = new Scanner(System.in);
2  System.out.println("How many integers?");
3  int howMany = input.nextInt();
4  int[] ns = new int[howMany];
5  for(int i = 0; i < howMany; i ++) {
6    System.out.println("Enter an integer");
7    ns[i] = input.nextInt(); }
8  System.out.println("Enter an index:");
9  int i = input.nextInt();
10 if (i < 0 || i >= ns.length || ns[i] % 2 == 1) {
11 /* Error: invalid index or odd ns[i] */ }
12 else { println(ns[i] + " at index " + i + " is even."); }
```

Test Case 1: -1

Test Case 2: 3

Use of ||

Invalid condition



-1 < 0 T

3 < 0 F

3 >= 3 T ✓

Short-Circuit and Array Indexing

```
1  Scanner input = new Scanner(System.in);
2  System.out.println("How many integers?");
3  int howMany = input.nextInt();
4  int[] ns = new int[howMany];
5  for(int i = 0; i < howMany; i ++) {
6    System.out.println("Enter an integer");
7    ns[i] = input.nextInt(); }
8  System.out.println("Enter an index:");
9  int i = input.nextInt();
10 if( 0 <= i && i < ns.length && ns[i] % 2 == 0) {
11   println(ns[i] + " at index " + i + " is even."); }
12 else { /* Error: invalid index or odd ns[i] */ }
```

Test Case 1: -1

Test Case 2: 3

Exercise

AIOBE

0 1 2

too late

0 <= i && ns[i] % 2 == 0 && i < ns.length

Nested Loops : Flow Chart

```
for(int i = 0; i < a.length; i ++) {
  for(int j = 0; j < a.length; j ++) {
    System.out.println("(" + i + ", " + j + ")");
  } }
```

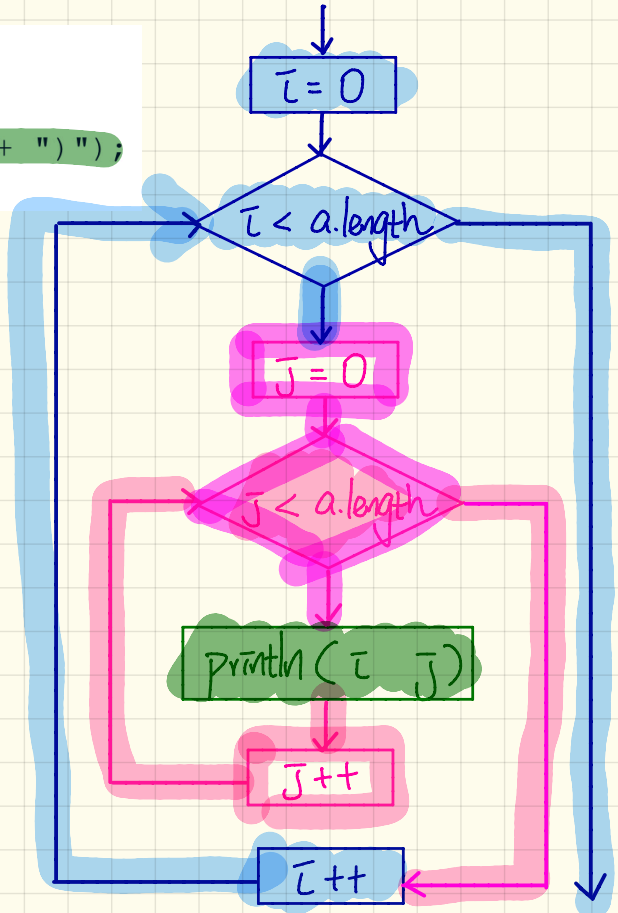
for (. .) {

  for (. .) {

    . —

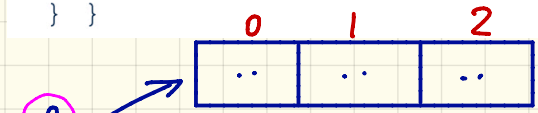
  }

}



Nested Loops : Tracing

```
for(int i = 0; i < a.length; i ++) {
  for(int j = 0; j < a.length; j ++) {
    System.out.println("(" + i + ", " + j + ")");
  } }
```



i	j
0	0
	1
	2
1	0
	1
	2
2	0
	1
	2

Single iteration of outer loop

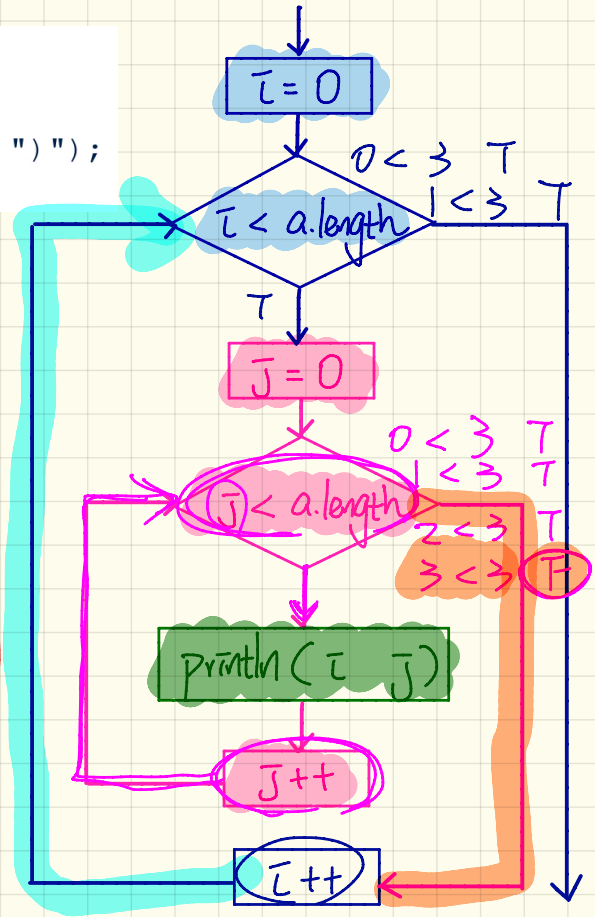
3 iterations of inner loop of outer loop

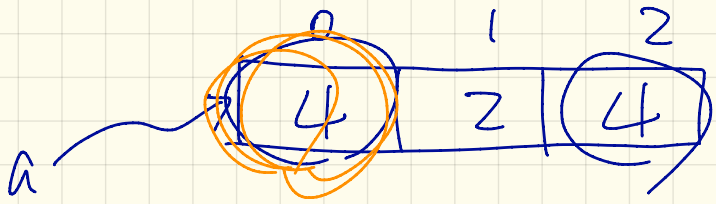
Output
(0, 0)
(0, 1)
(0, 2)
(1, 0)
(1, 1)
(1, 2)
(2, 0)
(2, 1)
(2, 2)

1st iteration of outer loop

2nd

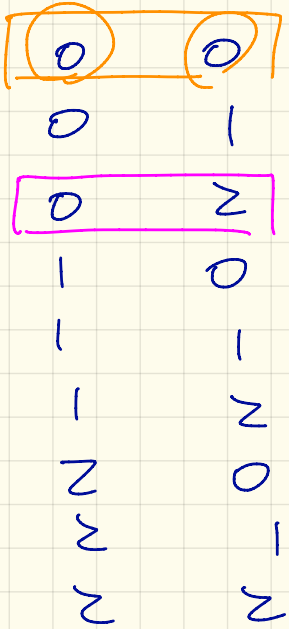
3rd





duplicates? True

a[0] == a[2]

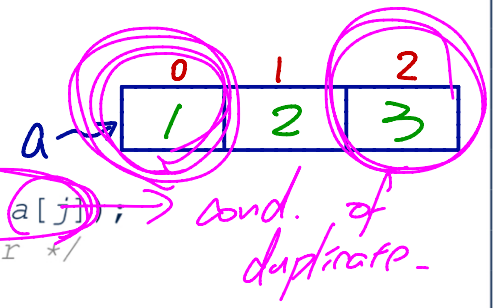


a[0] == a[0] → true

a[0] == a[2] → duplicates

Finding Duplicates: No Duplicates, Redundant Scan

```
1  /* Version 1 with redundant scan */
2  int[] a = {1, 2, 3}; /* no duplicates */
3  boolean hasDup = false;
4  for(int i = 0; i < a.length; i ++) {
5    for(int j = 0; j < a.length; j ++) {
6      hasDup = hasDup || (i != j && a[i] == a[j]);
7    } /* end inner for */ } /* end outer for */
8  System.out.println(hasDup);
```



i	j	i != j	a[i]	a[j]	a[i] == a[j]	hasDup
0	0	false	1	1	true	false
0	1	true	1	2	false	false
0	2	true	1	3	false	false
1	0	true	2	1	false	false
1	1	false	2	2	true	false
1	2	true	2	3	false	false
2	0	true	3	1	false	false
2	1	true	3	2	false	false
2	2	false	3	3	true	false

Wednesday February 27

Lecture 14

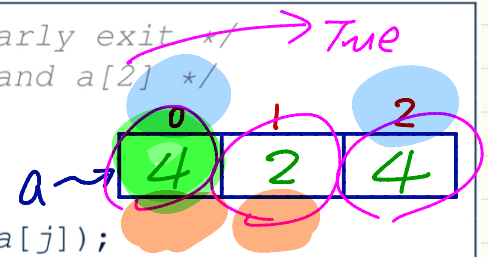
Finding Duplicates: No Duplicates, Redundant Scan

```
1  /* Version 1 with redundant scan */
2  int[] a = {1, 2, 3}; /* no duplicates */
3  boolean hasDup = false;
4  for(int i = 0; i < a.length; i ++) {
5    for(int j = 0; j < a.length; j ++) {
6      hasDup = hasDup || (i != j && a[i] == a[j]);
7    } /* end inner for */ } /* end outer for */
8  System.out.println(hasDup);
```

i	j	i != j	a[i]	a[j]	a[i] == a[j]	hasDup
0	0	false	1	1	true	false
0	1	true	1	2	false	false
0	2	true	1	3	false	false
1	0	true	2	1	false	false
1	1	false	2	2	true	false
1	2	true	2	3	false	false
2	0	true	3	1	false	false
2	1	true	3	2	false	false
2	2	false	3	3	true	false

Finding Duplicates : With Duplicates, Redundant Scan with no Early Exit

```
1  /* Version 1 with redundant scan and no early exit */
2  int[] a = {4, 2, 4}; /* duplicates: a[0] and a[2] */
3  boolean hasDup = false;
4  for(int i = 0; i < a.length; i ++) {
5    for(int j = 0; j < a.length; j ++) {
6      hasDup = hasDup || (i != j && a[i] == a[j]);
7    } /* end inner for */ } /* end outer for */
8  System.out.println(hasDup);
```



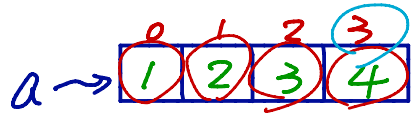
i	j	i != j	a[i]	a[j]	a[i] == a[j]	hasDup
0	0	false	4	4	true	false
0	1	true	4	2	false	false
0	2	true	4	4	true	true
1	0	true	2	4	false	true
1	1	false	2	2	true	true
1	2	true	2	4	false	true
2	0	true	4	4	true	true
2	1	true	4	2	false	true
2	2	false	4	4	true	true

N!

Finding Duplicates: No Duplicates, Non-Redundant Scan

have not found dup.

```
1  /* Version 3 with no redundant scan */
2  int[] a = {1, 2, 3, 4}; /* no duplicates */
3  boolean hasDup = false;
4  for(int i = 0; i < a.length && !hasDup; i ++) {
5    for(int j = i + 1; j < a.length && !hasDup; j ++) {
6      hasDup = a[i] == a[j];
7    } /* end inner for */ } /* end outer for */
8  System.out.println(hasDup);
```



0 1
0 2
0 3

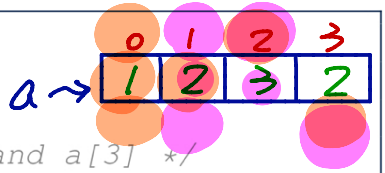
i	j	a[i]	a[j]	a[i] == a[j]	hasDup
0	1	1	2	false	false
0	2	1	3	false	false
0	3	1	4	false	false
1	2	2	3	false	false
1	3	2	4	false	false
2	3	3	4	false	false

1 2
1 3

(0,0) X
3 comb.
(1,0) X
2 comb.
1 comb.

Finding Duplicates: With Duplicates, Non-Redundant Scan with Early Exit

```
1  /* Version 3 with no redundant scan:
2   * array with duplicates causes early exit
3   */
4  int[] a = {1, 2, 3, 2}; /* duplicates: a[1] and a[3] */
5  boolean hasDup = false;
6  for(int i = 0; i < a.length && !hasDup; i ++) {
7    for(int j = i + 1; j < a.length && !hasDup; j ++) {
8      hasDup = a[i] == a[j];
9    } /* end inner for */ } /* end outer for */
10 System.out.println(hasDup);
```



i	j	a[i]	a[j]	a[i] == a[j]	hasDup
0	1	1	2	false	false
0	2	1	3	false	false
0	3	1	2	false	false
1	2	2	3	false	false
1	3	2	2	true	true

we found a duplicate

Common Errors (1) Improper Initialization of Loop Counter

```
boolean userWantsToContinue;
while (userWantsToContinue) {
  /* some computations here */
  String answer = input.nextLine();
  userWantsToContinue = answer.equals("Y");
}
```

= false;

false by default

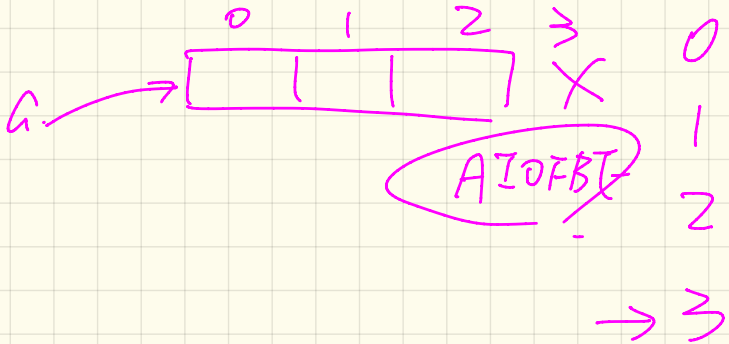
default value

Common Errors (2) Improper Stay Condition

3 <= 3

```
for (int i = 0; i <= a.length; i ++) {
  System.out.println(a[i]);
}
```

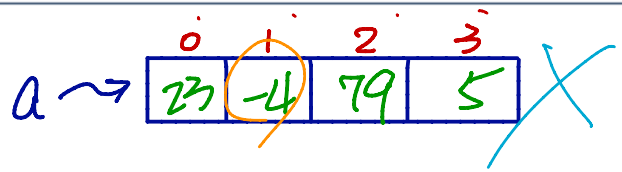
a[3]



Common Errors (3) Improper Update to Loop Counter

Does the following loop print all slots of array `a`?

```
int i = 0;
while (i < a.length) {
    i ++;
    System.out.println(a[i]);
}
```



i	a[i]
→ 0	
1	a[1] -4
2	a[2] 79
3	a[3] 5
4	a[4] X

AIOBE.

Common Errors (4) Improper Update of Stay Condition

```
1 String answer = input.nextLine();
2 boolean userWantsToContinue = answer.equals("Y");
3 while (userWantsToContinue) { /* stay condition (SC) */
4   /* some computations here */
5   answer = input.nextLine();
6 }
```

"N"

userWantsToContinue = answer.equals("Y");

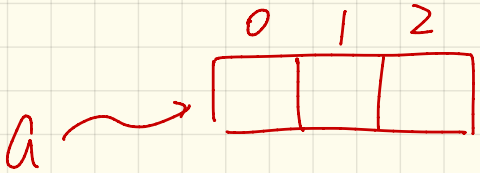
Common Errors (5) Improper Start Value of Loop Counter

```
int i = a.length - 1;
while (i >= 0) {
  System.out.println(a[i]); i --; }
while (i < a.length) {
  System.out.println(a[i]); i ++; }
```

2

when exiting the 1st loop i == -1

-1 < 3 T



i

2

1

0 → 0 >= 0 T

-1

a[-1] → AIOOBE

Common Errors (6) Misplaced Semicolon

Semicolon (;) in Java marks *the end of a statement* (e.g., assignment, `if` statement, `for`, `while`).

```
int[] ia = {1, 2, 3, 4};
for (int i = 0; i < 10; i ++); {
  System.out.println("Hello!");
}
```

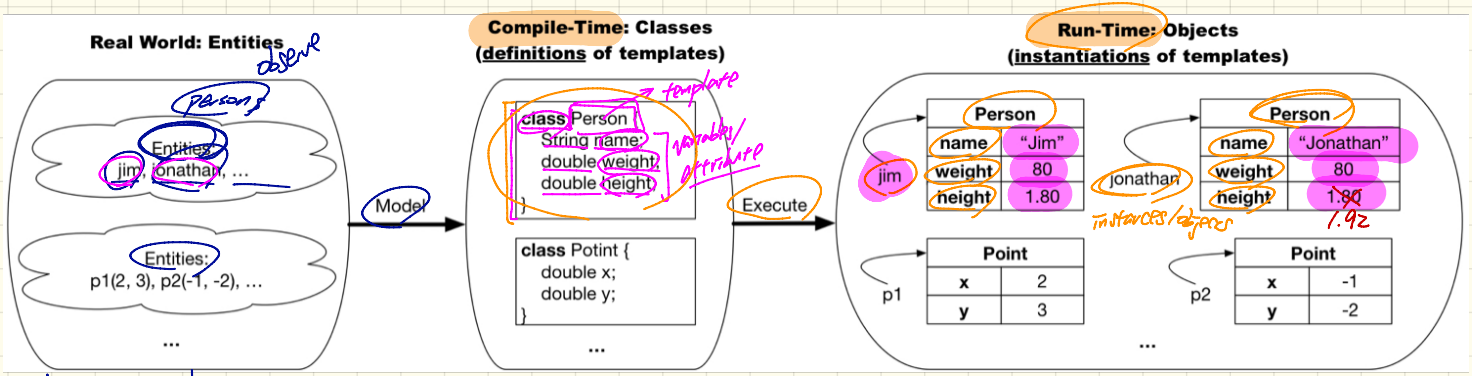
body of for loop is empty

not the body of for loop.

Output?

Hello.

Entities, Classes, Objects



object-oriented

Monday March 4

Lecture 15

Object-Oriented Programming (OOP)

- Templates (compile-time Java classes)
  - ~ attributes (variables)
  - ~ methods
    - → constructor
    - mutator
    - accessor
- Instances / Entities (runtime objects)
  - ~ calling constructor to create objects
  - ~ use of "dot notation" to
    - get attribute values
    - call accessor or mutator

OOP

Model: From Entities to Classes

Identify Critical Nouns & Verbs

attributes / classes

methods

class/template

Example 1

A person is a being, such as a human, that has certain attributes and behaviour constituting personhood: a person ages and grows on their heights and weights.

Example 2

Points on a two-dimensional plane are identified by their signed distances from the X- and Y-axises. A point may move arbitrarily towards any direction on the plane. Given two points, we are often interested in knowing the distance between them.

```java
public class Person {
    /*
     * Attributes.
     * These are variable declared at the class level.
     * All methods may use them.
     */
    int age;
    String nationality;
    double weight; /* kg */
    double height; /* meters */

    /*
     * Constructors.
     */
    Person(int newAge, double newWeight, double newHeight) {
        age = newAge;
        weight = newWeight;
        height = newHeight;
    }
}
```

attributes

parameters

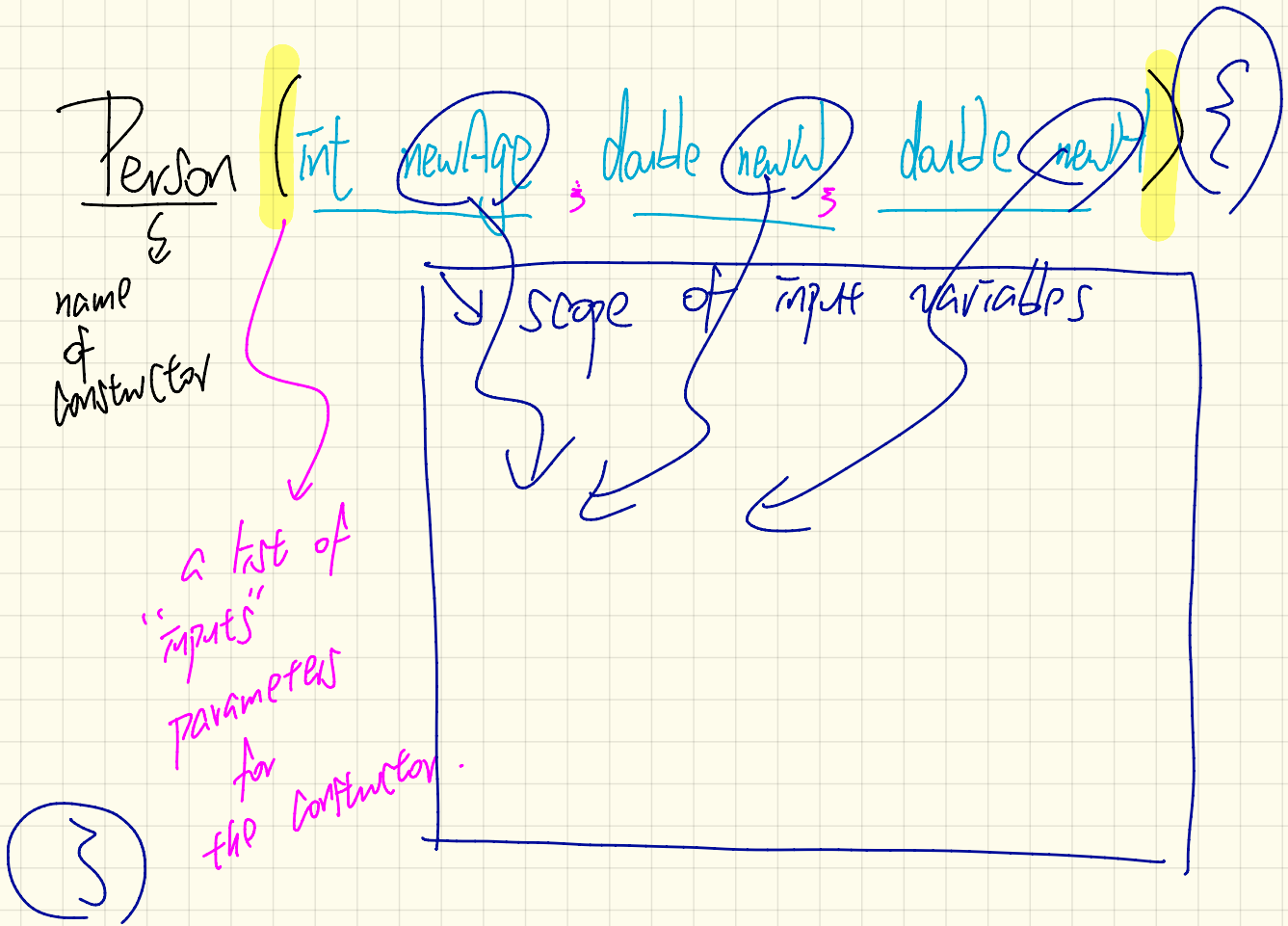
definition of constructor

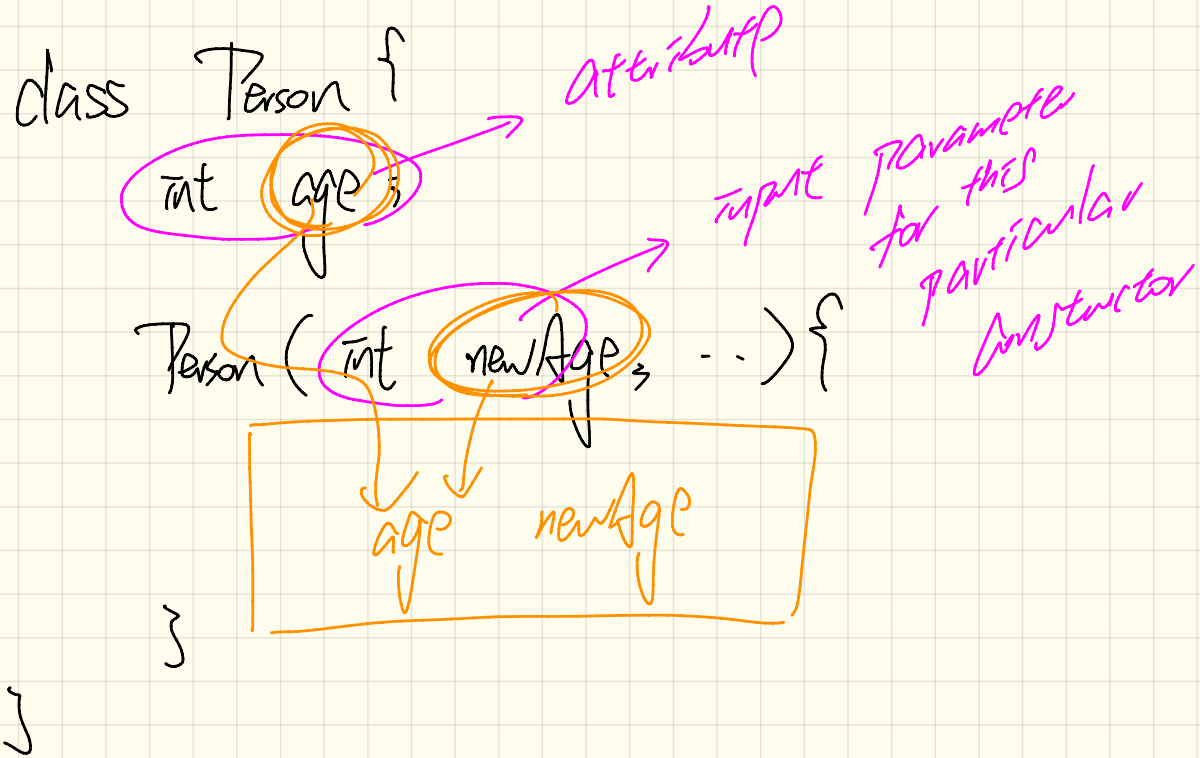
Person
age
nat.
w.
h.

use of constructor

```java
public class Tester {
    public static void main(String[] args) {
        Person jim = new Person(45, 72, 172);
    }
}
```

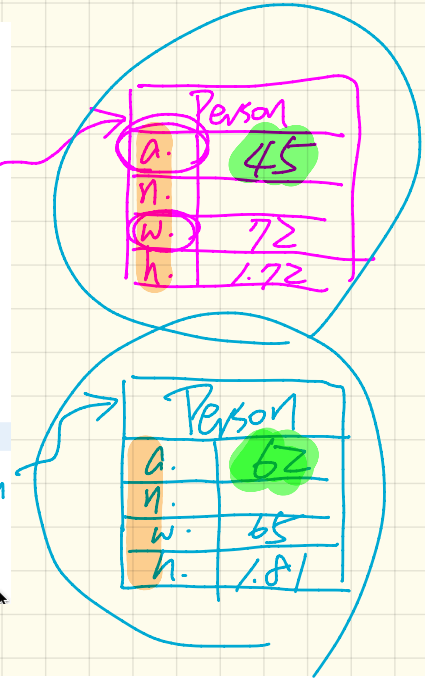
create a new object


Person
(
int newAge
,
double newW
,
double newH
)
{
name of constructor
scope of input variables
a list of "inputs" parameters for the constructor.
}


class Person {
int age;
attribute
Person (int newAge, . .) {
input parameter for this particular constructor
age
newAge
}
}

```java
public class Person {
    /*
     * Attributes.
     * These are variable declared at the class level.
     * All methods may use them.
     */
    int age;
    String nationatlity;
    double weight; /* kg */
    double height; /* meters */

    /*
     * Constructors.
     */
    public Person(int newAge, double newWeight, double newHeight) {
        age = newAge;
        weight = newWeight;
        height = newHeight;
    }
}
```

define the constructor

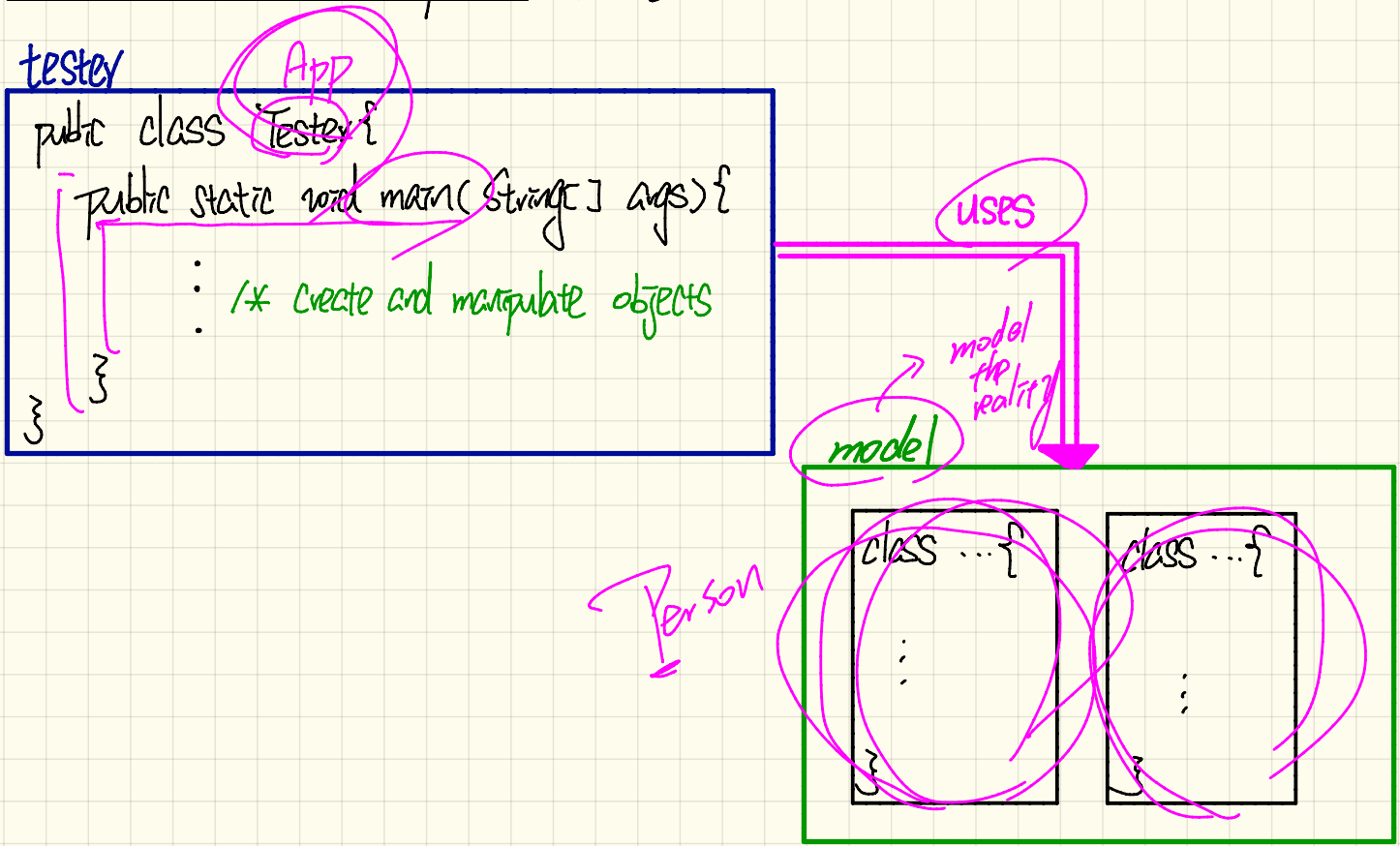



```java
public class Tester {

    public static void main(String[] args) {
        Person jim = new Person(45, 72, 1.72);
        Person jonathan = new Person(62, 65, 1.81);
    }

}
```

Jim.age
Jonathan.age
attribute name
object names

two calls to constructor

Test Driven Development (TDD)

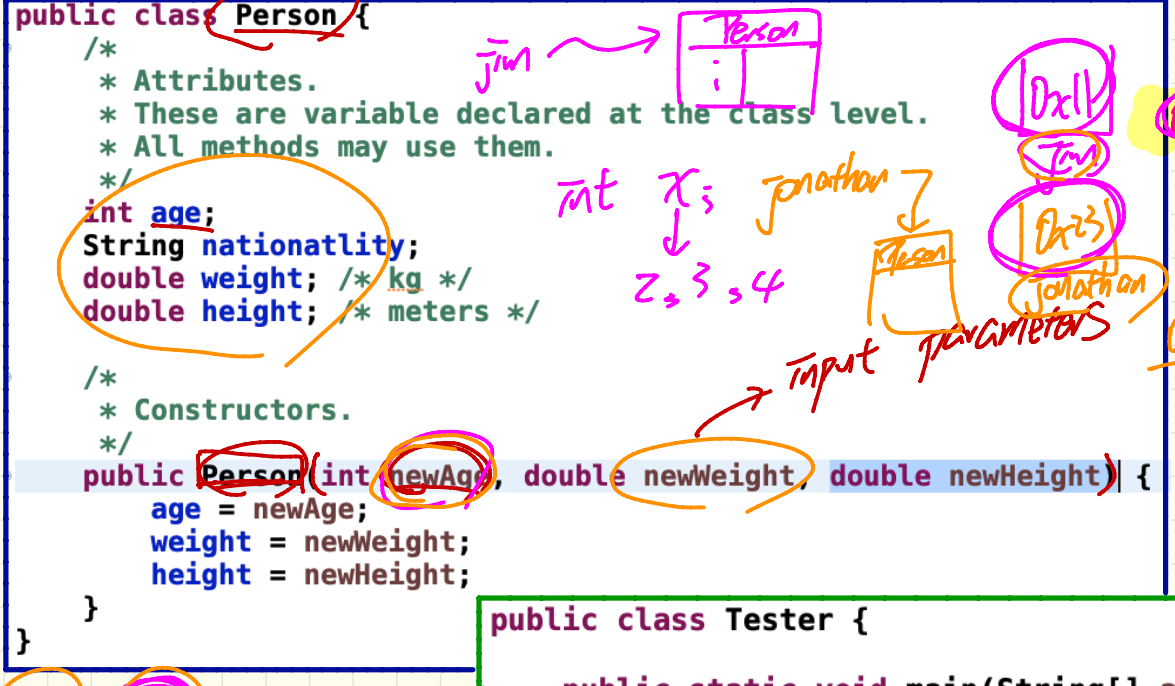


Wednesday March 6

Lecture 16

Constructors

```java
public class Person {
    /*
     * Attributes.
     * These are variable declared at the class level.
     * All methods may use them.
     */
    int age;
    String nationality;
    double weight; /* kg */
    double height; /* meters */

    /*
     * Constructors.
     */
    public Person(int newAge, double newWeight, double newHeight) {
        age = newAge;
        weight = newWeight;
        height = newHeight;
    }
}
```




```java
public class Tester {

    public static void main(String[] args) {
        Person jim = new Person(45, 72, 1.72);
        Person jonathan = new Person(62, 65, 1.81);
    }
}
```

Person jim = new Person(...);

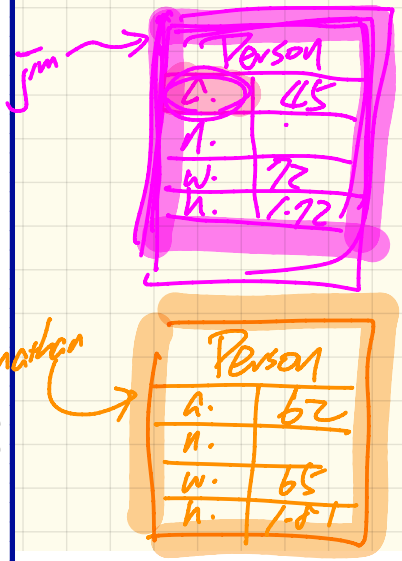
jim == jonathan false

Constructors using this keyword

```java
public class Person {
    /*
     * Attributes
     */
    int age;
    String nationality;
    double weight; /* kg */
    double height; /* meters */

    /*
     * Constructors
     */
    Person (int age, double weight, double height) {
        this.age = age;
        this.weight = weight;
        this.height = height;
    }
```

this.age = 45;

this.age = 62;



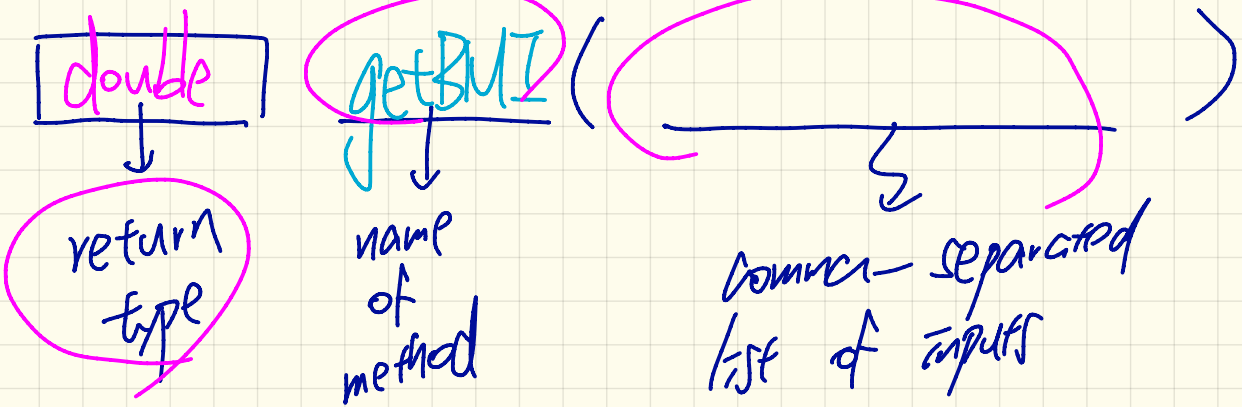

```java
    public static void main(String[] args) {
        Person jim = new Person(45, 72, 1.72);
        Person jonathan = new Person(62, 65, 1.81);
    }

}
```

Constructor – create new objects

- Accessor | method – inquire about the object
- Mutator | method – modify attributes

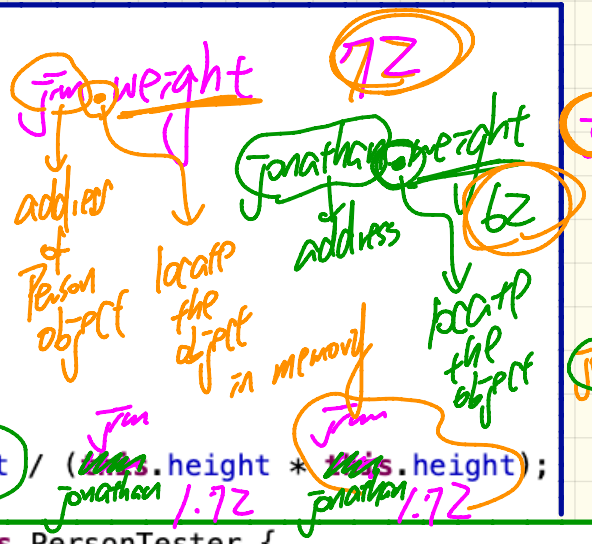
named block of method

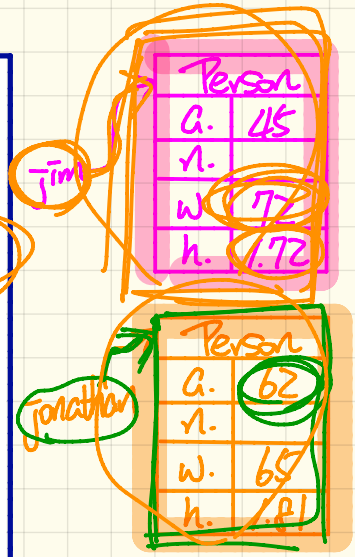
Method (accessor / mutator)



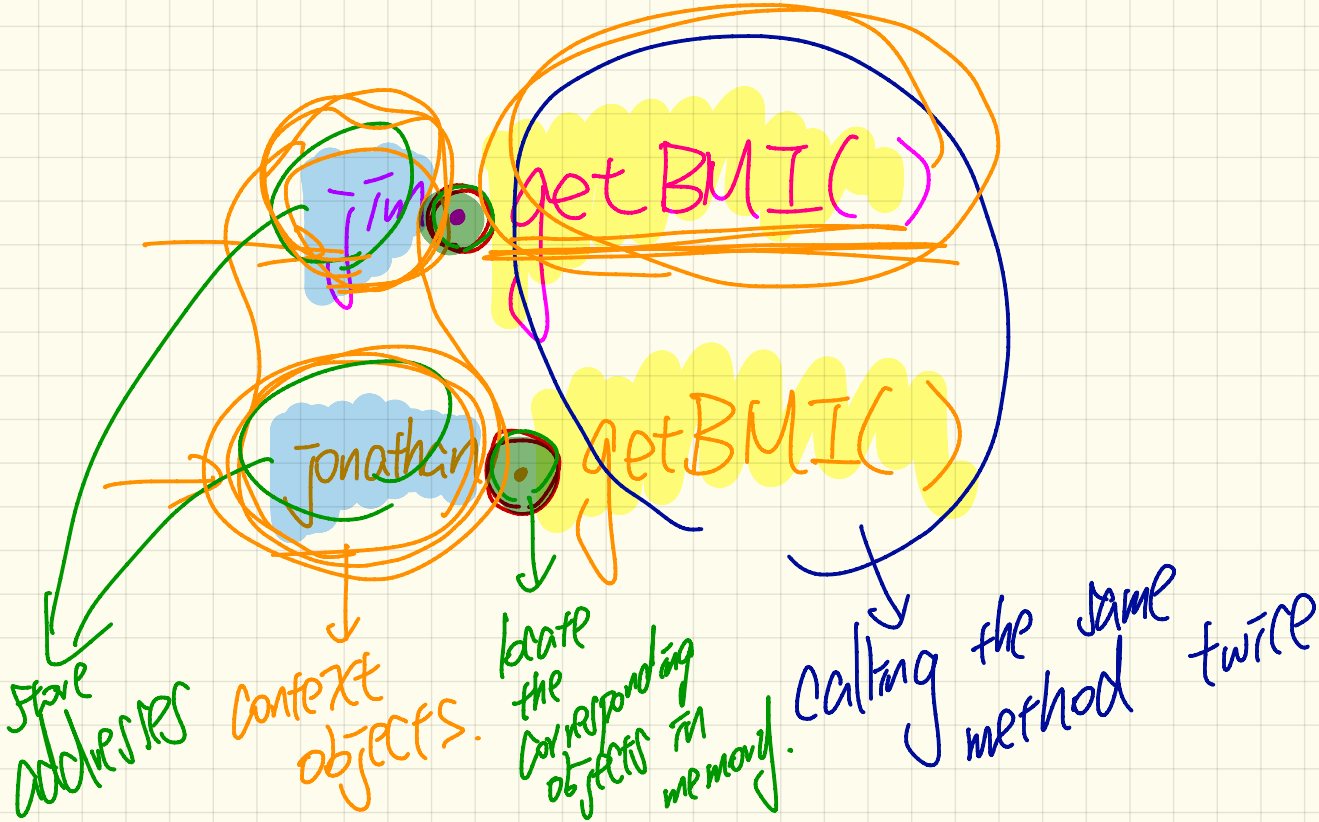
Accessors

```java
public class Person {
    /*
     * Attributes
     */
    int age;
    String nationality;
    double weight; /* kg */
    double height; /* meters */

    /*
     * Accessors
     */
    double getBMI() {
        double bmi = jim.weight / (jonathan.height * jonathan.height);
        return bmi;
    }
```





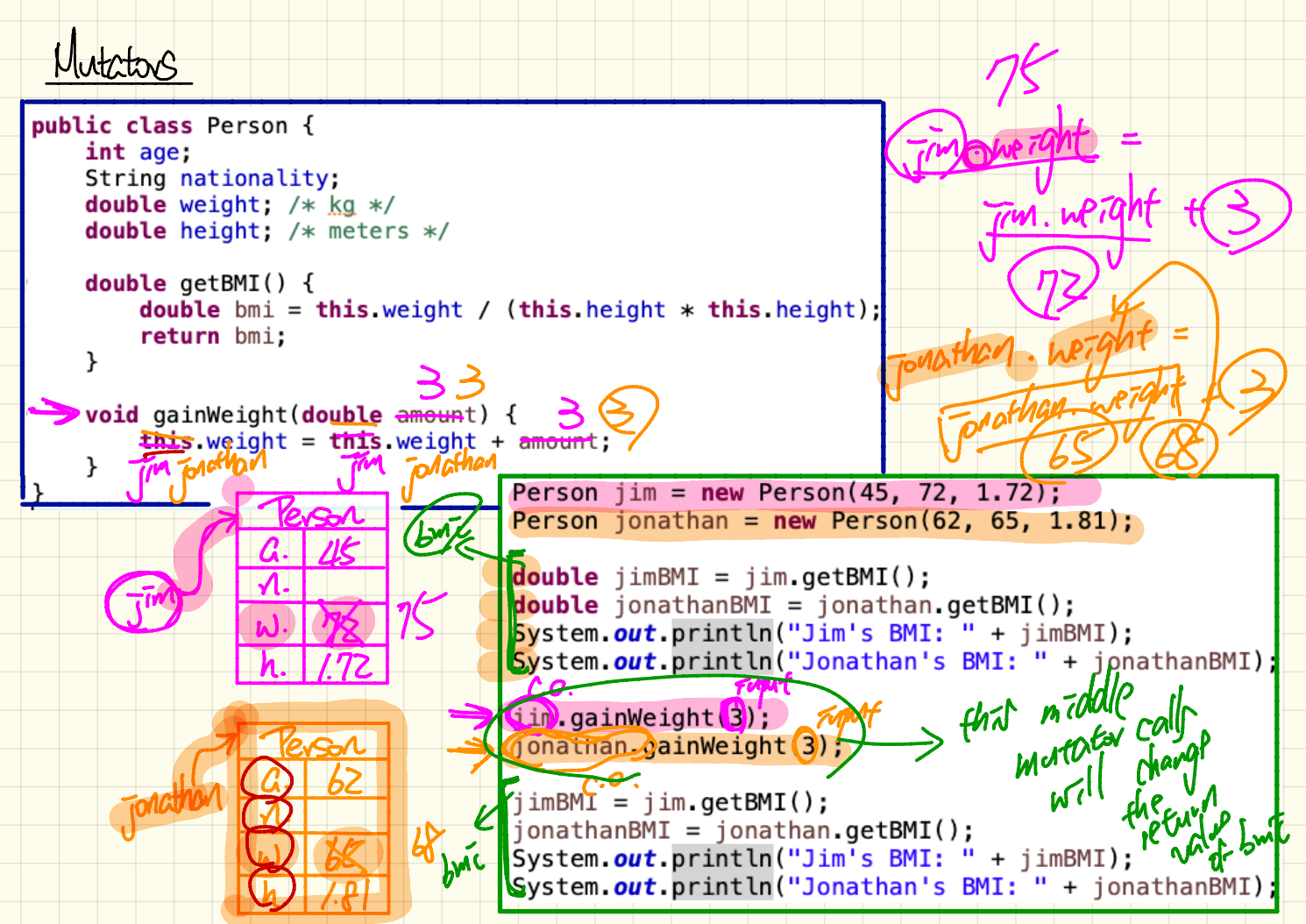

```java
public class PersonTester {

    public static void main(String[] args) {
        Person jim = new Person(45, 72, 1.72);
        Person jonathan = new Person(62, 65, 1.81);

        double jimBMI = jim.getBMI();
        double jonathanBMI = jonathan.getBMI();
        System.out.println("Jim's BMI: " + jimBMI);
        System.out.println("Jonathan's BMI: " + jonathanBMI);
```


Tim
getBMI()
Jonathan
getBMI()
store addresses
context objects.
locate the corresponding objects in memory.
calling the same method twice

Mutators

```java
public class Person {
    int age;
    String nationality;
    double weight; /* kg */
    double height; /* meters */

    double getBMI() {
        double bmi = this.weight / (this.height * this.height);
        return bmi;
    }

    void gainWeight(double amount) {
        this.weight = this.weight + amount;
    }
}
```

```java
Person jim = new Person(45, 72, 1.72);
Person jonathan = new Person(62, 65, 1.81);

double jimBMI = jim.getBMI();
double jonathanBMI = jonathan.getBMI();
System.out.println("Jim's BMI: " + jimBMI);
System.out.println("Jonathan's BMI: " + jonathanBMI);

jim.gainWeight(3);
jonathan.gainWeight(3);

jimBMI = jim.getBMI();
jonathanBMI = jonathan.getBMI();
System.out.println("Jim's BMI: " + jimBMI);
System.out.println("Jonathan's BMI: " + jonathanBMI);
```

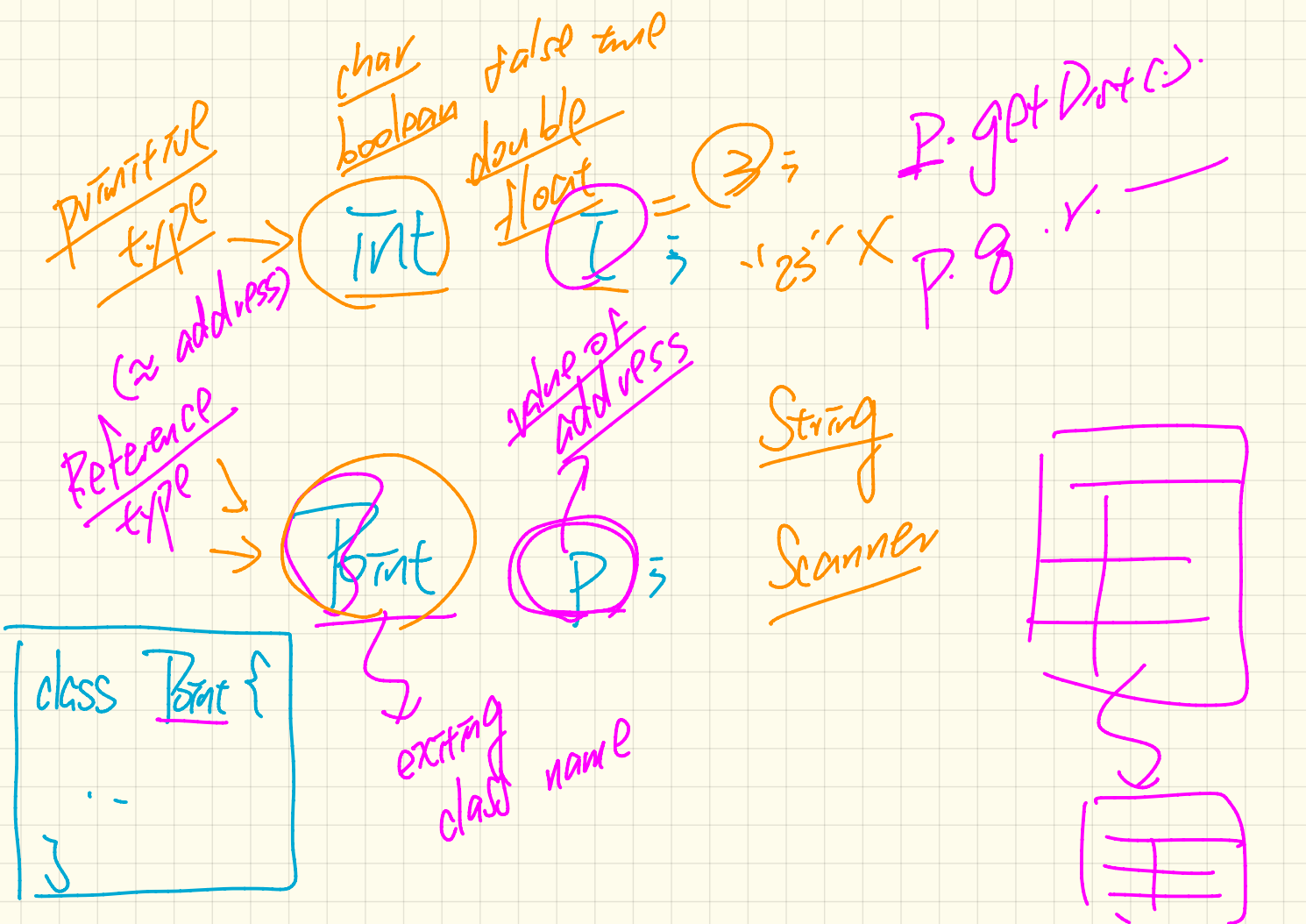


Monday March 11

Lecture 17

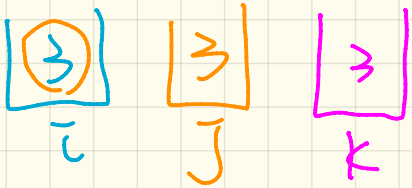
Exam: April 7

Last Class: April 3


primitive type
char
boolean
false true
double
float
int
i
3;
i = '23' X
p.getDist();
p.g
(≈ address)
Reference type
Point
value of address
p
String
Scanner
class Point {
}
existing class name

Copy of Variables : Primitive Type

```
1  int i = 3;
2  int j = i;   System.out.println(i == j);
3  int k = 3;   System.out.println(k == i && k == j);
```



Copy of Variable : Reference Type

```
1  Point p1 = new Point(2, 3);
2  Point p2 = p1;   System.out.println(p1 == p2);
3  Point p3 = new Point(2, 3);
4  Systme.out.println(p3 == p1 || p3 == p2);
5  Systme.out.println(p3.x == p1.x && p3.y == p1.y);
6  Systme.out.println(p3.x == p2.x && p3.y == p2.y);
```

are p1 and p2 pointing to the same object?

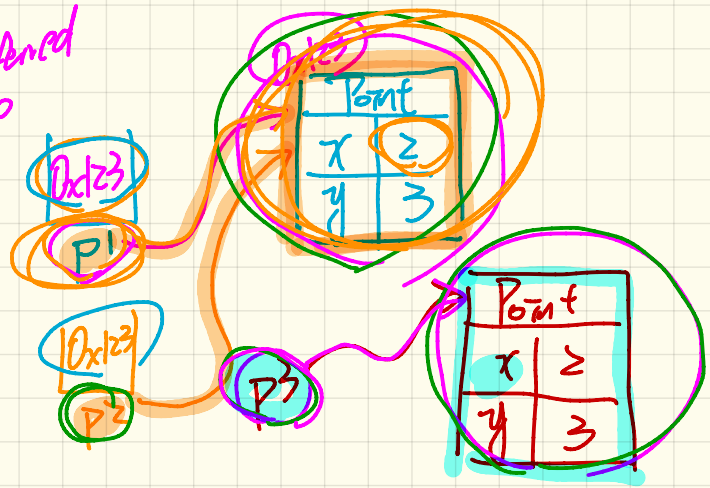
T

F F

T T

```
class Point {
   double x;
   double y;
   Point(double x, double y) {
      this.x = x;
      this.y = y;
   }
}
```

lookup the object being referred to



```
Point p1;
Point p2;
...
p1 == p2
```

p1 and p2 point to the same object?

```
p1.x == p2.x && p1.y == p2.y
```

Problem: Consider assignments to *primitive* variables:

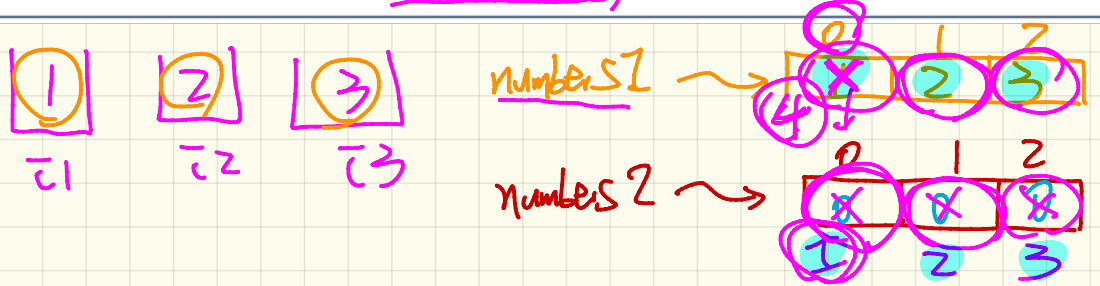
```
1  int i1 = 1;
2  int i2 = 2;
3  int i3 = 3;
4  int[] numbers1 = {i1, i2, i3};
5  int[] numbers2 = new int[numbers1.length];
6  for(int i = 0; i < numbers1.length; i ++) {
7    numbers2[i] = numbers1[i];
8  }
9  numbers1[0] = 4;
10 System.out.println(numbers1[0]);
11 System.out.println(numbers2[0]);
```

0 1 2

ns2[0] = ns1[0]
ns2[1] = ns1[1]

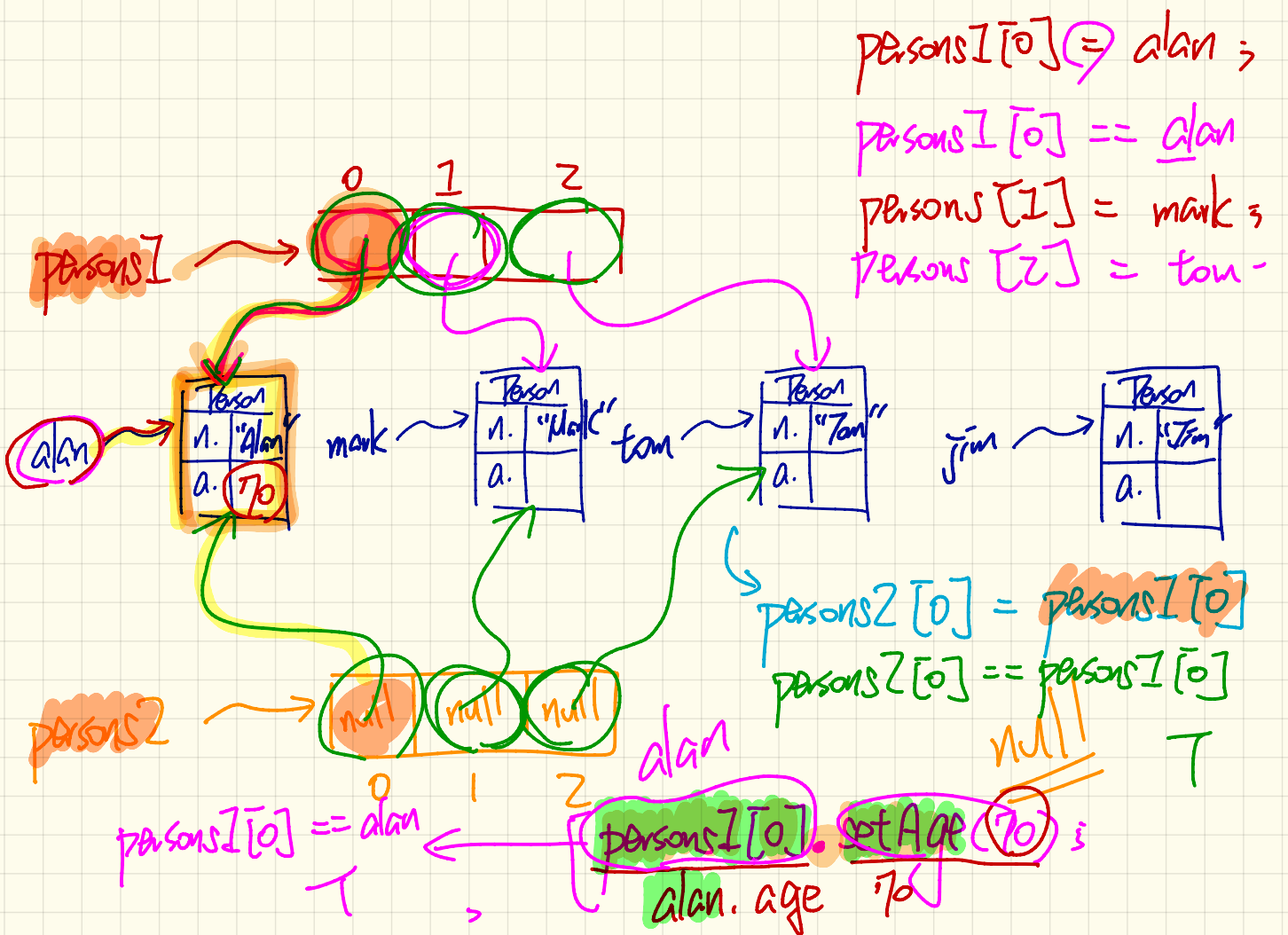
4

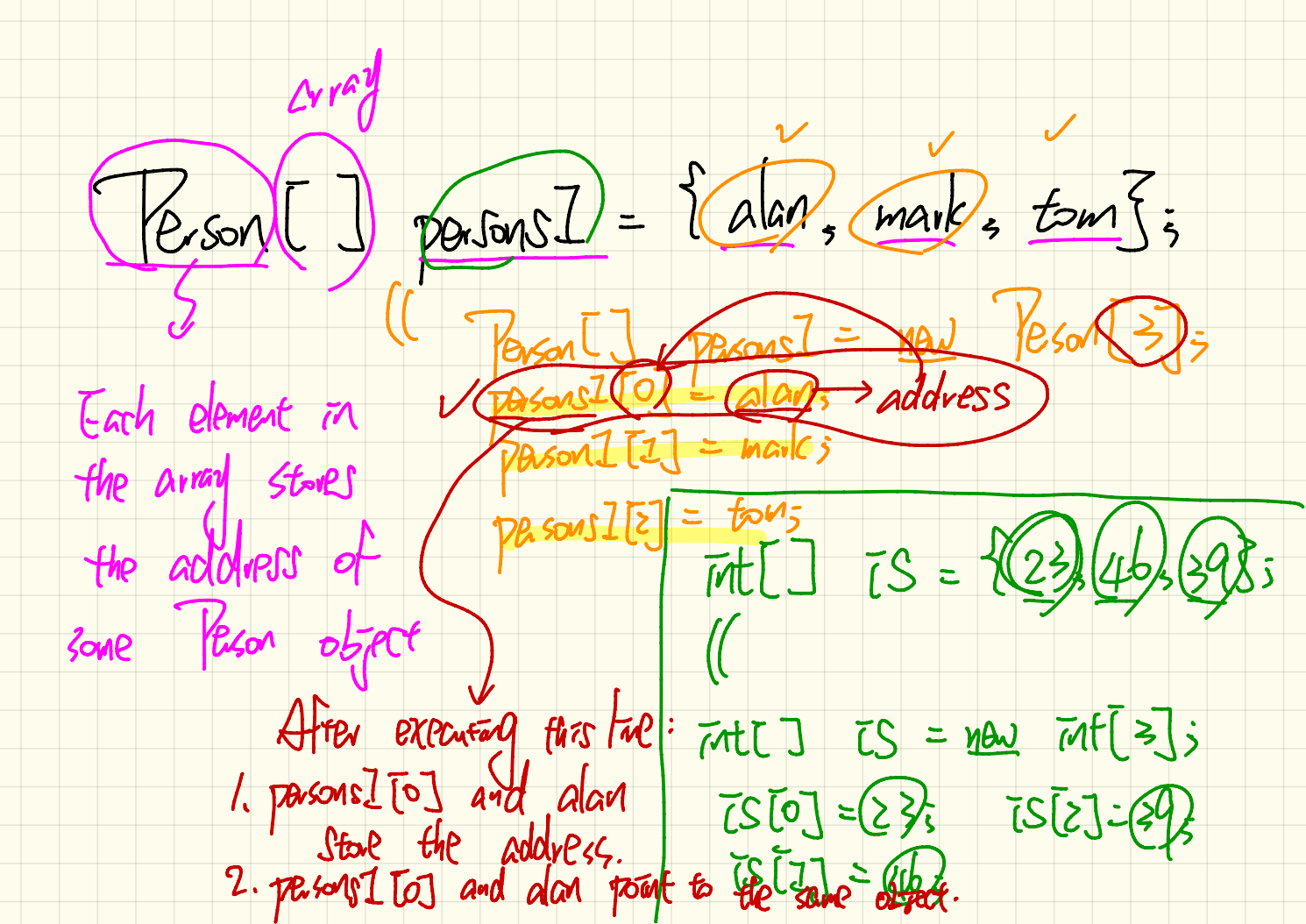
1



Problem: Consider assignments to *reference* variables:

```
1  Person alan = new Person("Alan");
2  Person mark = new Person("Mark");
3  Person tom = new Person("Tom");
4  Person jim = new Person("Jim");
5  Person[] persons1 = {alan, mark, tom};
6  Person[] persons2 = new Person[persons1.length];
7  for(int i = 0; i < persons1.length; i ++) {
8   persons2[i] = persons1[i]; }
9  persons1[0].setAge(70);
10 System.out.println(jim.age);
11 System.out.println(alan.age);
12 System.out.println(persons2[0].age);
13 persons1[0] = jim;
14 persons1[0].setAge(75);
15 System.out.println(jim.age);
16 System.out.println(alan.age);
17 System.out.println(persons2[0].age);
```


persons1[0] = alan;
persons1[0] == alan
persons1[1] = mark;
persons1[2] = tom;
persons1
0
1
2
alan
Person
n. "Alan"
a. 70
mark
Person
n. "Mark"
a.
tom
Person
n. "Tom"
a.
jim
Person
n. "Jim"
a.
persons2
null
null
null
persons2[0] = persons1[0]
persons2[0] == persons1[0]
T
null
alan
persons1[0].setAge(70);
alan.age 70
persons1[0] == alan
T


array
Person[] persons1 = {alan, mark, tom};
Each element in the array stores the address of some Person object
Person[] persons1 = new Person[3];
persons1[0] = alan; → address
person1[1] = mark;
persons1[2] = tom;
int[] iS = {23, 46, 39};
int[] iS = new int[3];
iS[0] = 23; iS[2] = 39;
iS[1] = 46;
After executing this line:
1. persons1[0] and alan store the address.
2. persons1[0] and alan point to the same object.

Wednesday March 13

Lecture 18

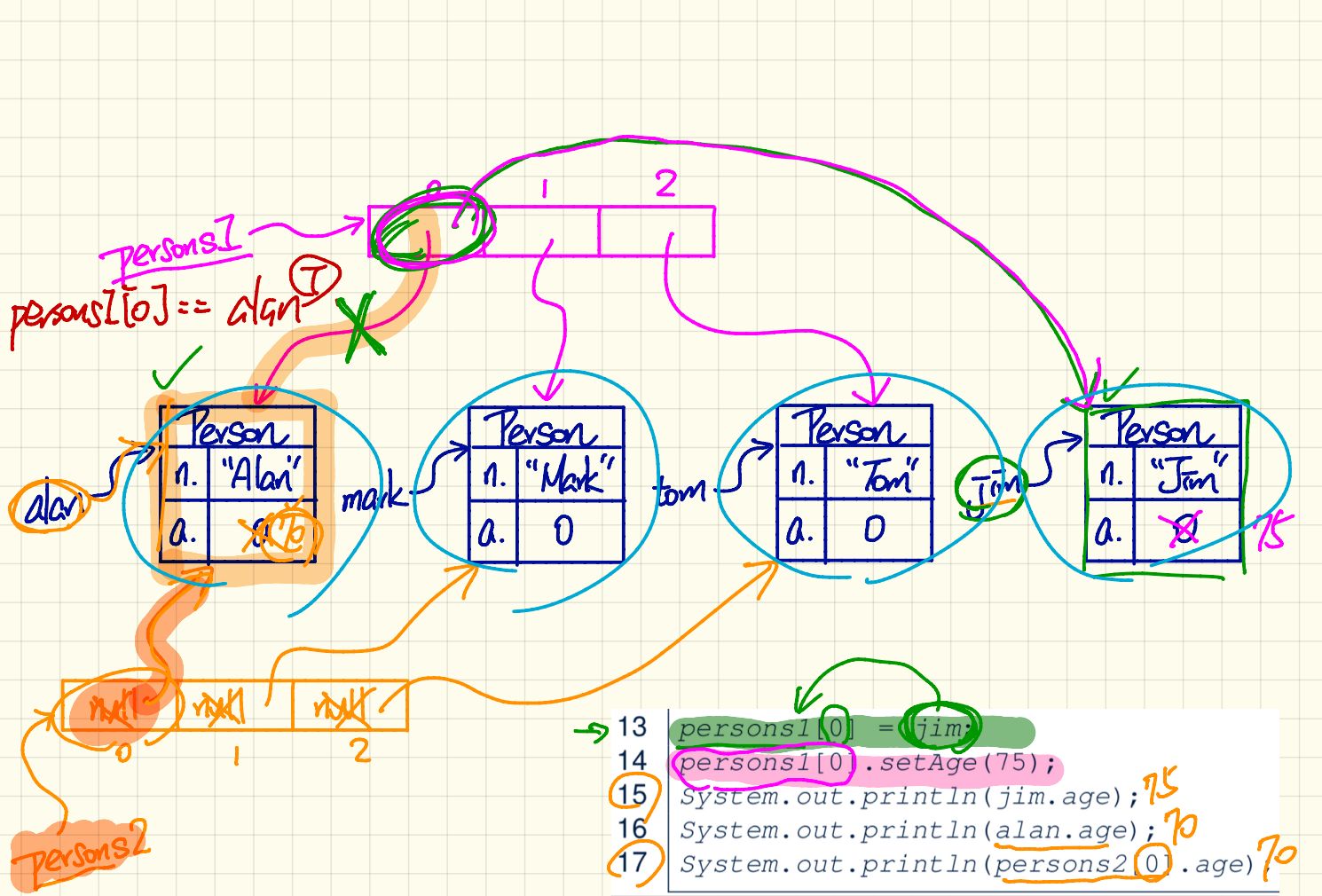
- Lab 6
- Lab Test 3
  - ~ Guide
  - ~ Practice Test
- Tutorial Videos

```
1  Person alan = new Person("Alan");
2  Person mark = new Person("Mark");
3  Person tom = new Person("Tom");
4  Person jim = new Person("Jim");
5  Person[] persons1 = {alan, mark, tom};
6  Person[] persons2 = new Person[persons1.length];
7  for(int i = 0; i < persons1.length; i ++) {
8    persons2[i] = persons1[i]; }
9  persons1[0].setAge(70);
10 System.out.println(jim.age);
11 System.out.println(alan.age);
12 System.out.println(persons2[0].age);
13 persons1[0] = jim;
14 persons1[0].setAge(75);
15 System.out.println(jim.age);
16 System.out.println(alan.age);
17 System.out.println(persons2[0].age);
```

alan.age == 0

0
1
2

class Person {
String name
int age
}


persons1
0
1
2
persons1[0] == alan?
Person
n. "Alan"
a. 0
Person
n. "Mark"
a. 0
Person
n. "Tom"
a. 0
Person
n. "Jim"
a. 75
alan
mark
tom
Jim
persons2
0
1
2
13 persons1[0] = jim;
14 persons1[0].setAge(75);
15 System.out.println(jim.age);
16 System.out.println(alan.age);
17 System.out.println(persons2[0].age)
75
70
70

Person[] persons;

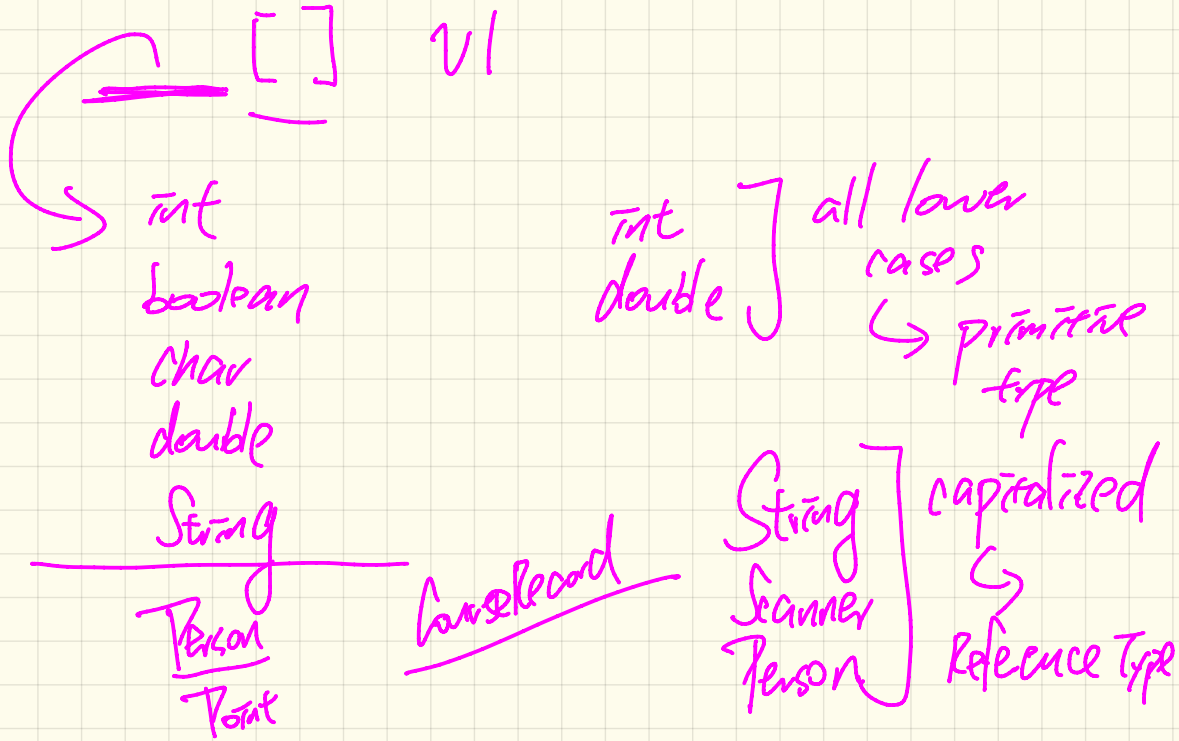
↳ as if:

Person persons[0]

Person persons[1]

⋮

Person persons[persons.length - 1]


[ ]
vl
int
boolean
char
double
String
Person
Point
Camelcase
int
double
all lower cases
primitive type
String
Scanner
Person
capitalized
Reference Type

String s1 = null;

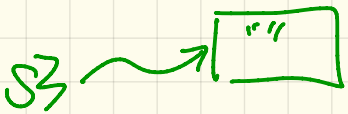
↳ store null address to begin with.

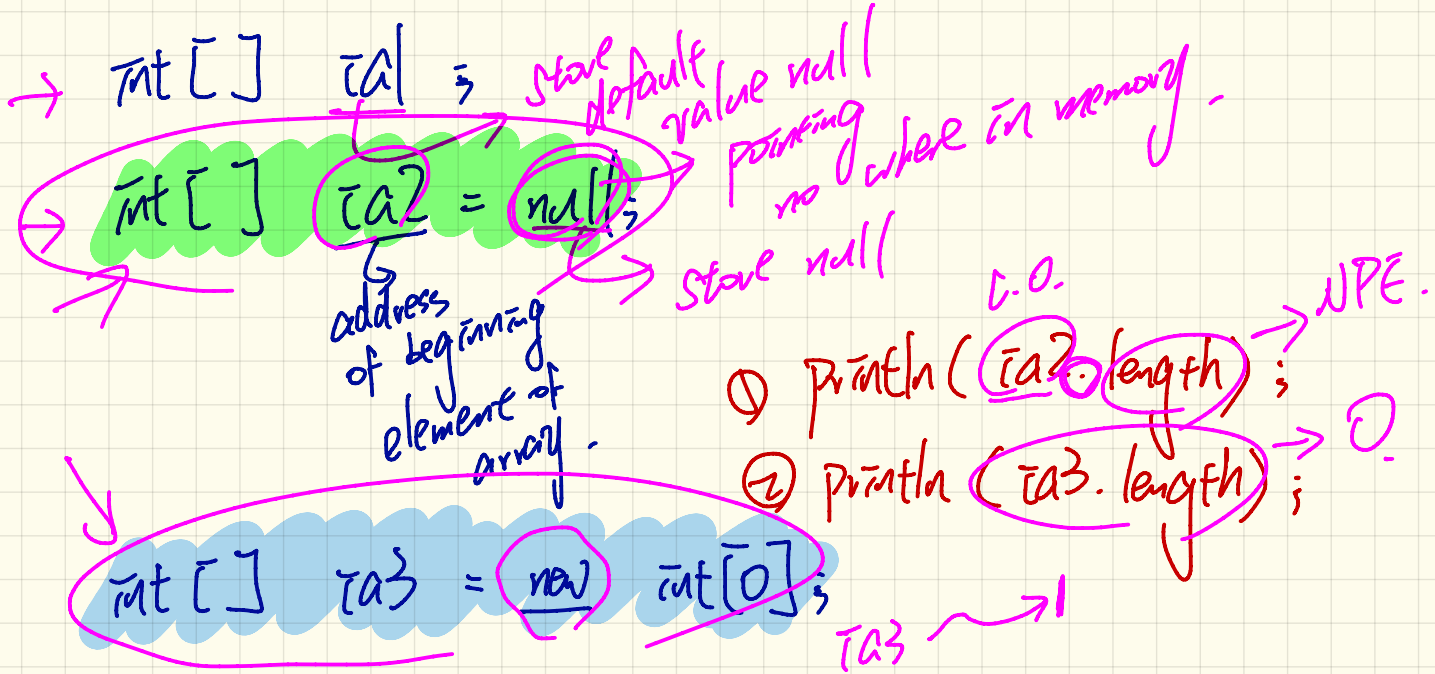
String s2;

↳ store default value null

String s3 = "";

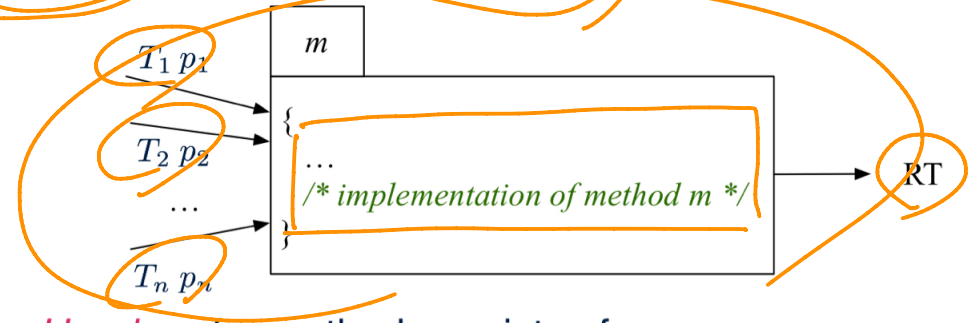
↳ empty string. "" " "




int [] ia ;
store default value null
int [] ia2 = null;
pointing no where in memory.
store null
address of beginning element of array.
C.O.
NPE.
① println(ia2.length);
② println(ia3.length);
0.
int [] ia3 = new int[0];
ia3

What is a method?

- A *method* is a named block of code, *reusable* via its name.



- The *Header* of a method consists of:
  - Return type [*RT* (which can be `void`)]
  - Name of method [*m*]
  - Zero or more *parameter names* [$p_1, p_2, \ldots, p_n$]
  - The corresponding *parameter types* [$T_1, T_2, \ldots, T_n$]
- A call to method *m* has the form: $m(a_1, a_2, \ldots, a_n)$
  Types of *argument values* $a_1, a_2, \ldots, a_n$ must match the the corresponding parameter types $T_1, T_2, \ldots, T_n$.

Parameters vs. Arguments

```
class Point {
   Point(double x, double y) {…}

   double getDistanceFrom(Point other) {…}

   void move(char direction, double units) {…}
}
```

parameters

Class/Template Definition

name of param.

parameters

```
class PointTester {
   static void main(String[] args) {
      Point p1 = new Point(2.5, -3.6);
      Point p2 = new Point(-4.8, 5.9);
      double dist1 = p1.getDistanceFrom(p2);
      double dist2 = p2.getDistanceFrom(p1);
      p1.move('R', 7.6);
   }
}
```

argument

argument

argument

double getDist(Point p1 , Point p2)

other

other

Method Usages

Kinds of Methods

1. *Constructor*
   - Same name as the class. No return type. *Initializes* attributes.
   - Called with the **new** keyword.
   - e.g., `Person jim = new Person(50, "British");`
2. *Mutator*
   - *Changes* (re-assigns) attributes
   - `void` return type
   - Cannot be used when a value is expected
   - e.g., `double h = jim.setHeight(78.5);` is illegal!
3. *Accessor*
   - *Uses* attributes for computations (without changing their values)
   - Any return type other than `void`
   - An explicit *`return` statement* (typically at the end of the method) returns the computation result to where the method is being used.
   - e.g., `double bmi = jim.getBMI();`
   - e.g., `println(p1.getDistanceFromOrigin());`

Use of Accessors vs. Mutators

```
class Person {
  void setWeight(double weight) {...}
  double getBMI() {...}
}
```

- Calls to *mutator methods* *cannot* be used as values.
  - e.g., `System.out.println(jim.setWeight(78.5));` ×
  - e.g., `double w = jim.setWeight(78.5);` ×
  - e.g., `jim.setWeight(78.5);` ✓
- Calls to *accessor methods* *should* be used as values.
  - e.g., `jim.getBMI();` ×
  - e.g., `System.out.println(jim.getBMI());` ✓
  - e.g., `double w = jim.getBMI();` ✓

Method Parameters

- **Principle 1:** A *constructor* needs an *input parameter* for every attribute that you wish to initialize.

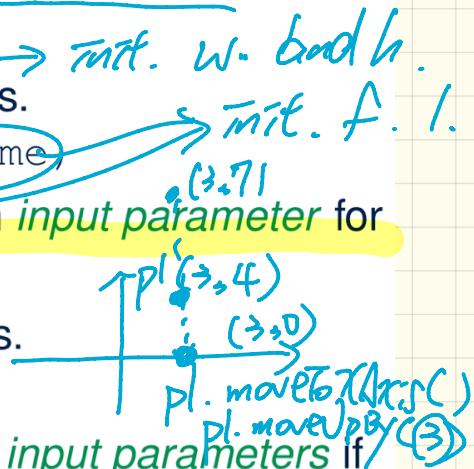
  e.g., `Person(double w, double h)` vs. `Person(String fName, String lName)`

- **Principle 2:** A *mutator* method needs an *input parameter* for every attribute that you wish to modify.

  e.g., In `Point`, `void moveToXAxis()` vs. `void moveUpBy(double unit)`

- **Principle 3:** An *accessor method* needs *input parameters* if the attributes alone are not sufficient for the intended computation to complete.

  e.g., In `Point`, `double getDistFromOrigin()` vs. `double getDistFrom(Point other)`



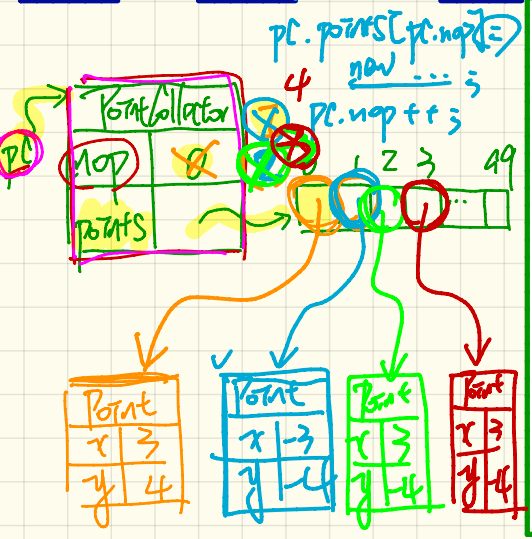
Monday March 18

Lecture 19

Programming Pattern: Mutator

```
class PointCollector {
  Point[] points; int nop; /* number of points */
  PointCollector() { points = new Point[100]; }
  void addPoint(double x, double y) {
    this.points[nop] = new Point(x, y); this.nop++; }
}
```

```
class PointCollectorTester {
  public static void main(String[] args) {
    PointCollector pc = new PointCollector();
    System.out.println(pc.nop);      /* 0 */
    pc.addPoint(3, 4);
    System.out.println(pc.nop);      /* 1 */
    pc.addPoint(-3, -4);
    System.out.println(pc.nop);      /* 2 */
    pc.addPoint(-3, -4);
    System.out.println(pc.nop);      /* 3 */
    pc.addPoint(3, -4);
    System.out.println(pc.nop);      /* 4 */
```



Circles.

the same

pc.add(Point(3,4)); Tutorial.

pc.ps[0].moveUp(3); null pointer exception

```
class PointCollector {
    int nop;
    Point[] ps;
    PointCollector() {
        this.ps = new Point[100];
    }
}
```

attribute nop is not initialized

default value: 0

pc → PC { nop: 1, ps: } → 0 1 2 .. 99

pc.ps[1].moveUp(3)

Point { x: 3, y: 4 }

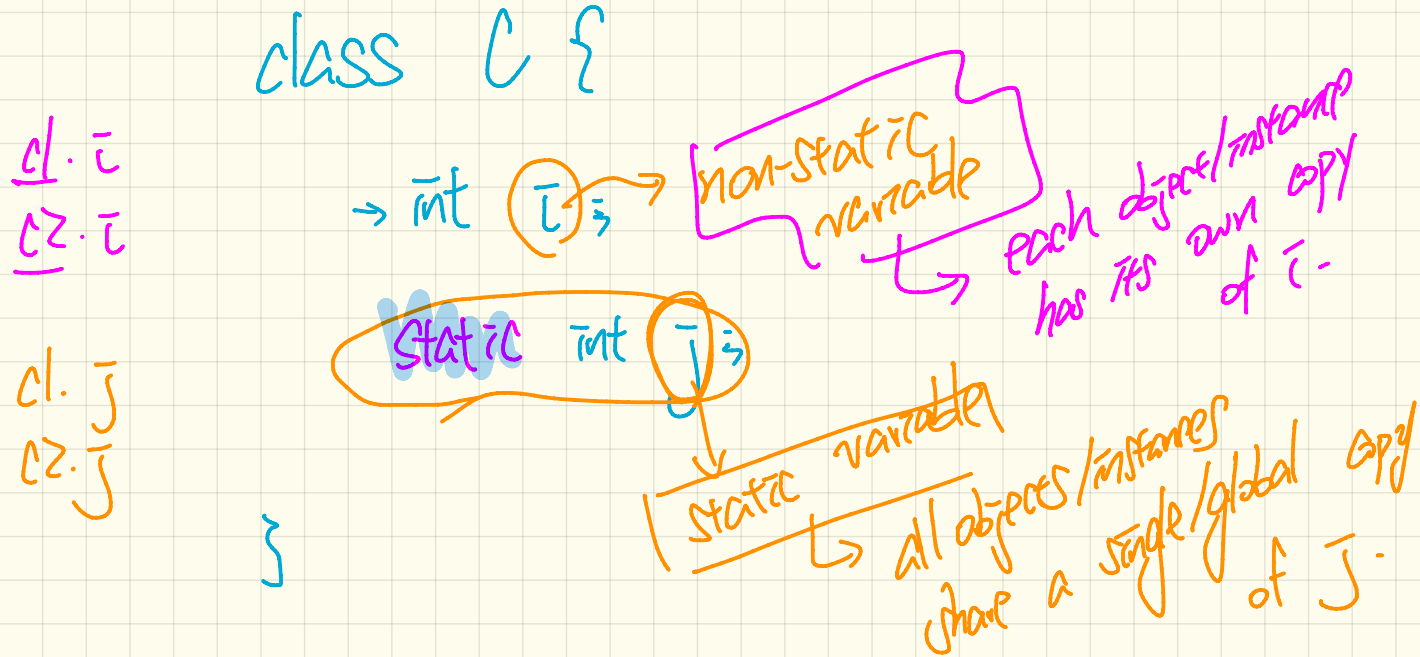
```
class PointCollector {
    int nop;
    Point[] ps;
    PointCollector() {
        this.ps = new Point[100];
        this.nop = 0;
    }
}
```

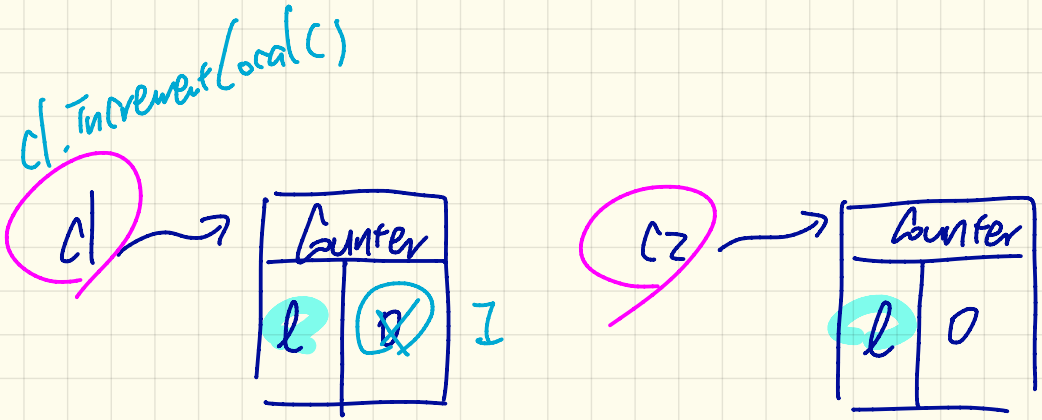
1.

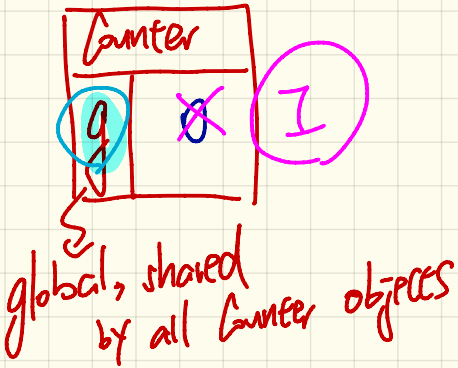

```
class PointCollector {
    int nop;
    Point[] ps;
    PointCollector() { ... }
    void addPoint(x, y) { ... }
}
```

```
pc this.ps[nop] = new Point(x, y);
this.nop++;
```

```
PointCollector pc =
    new PointCollector();
pc.addPoint(3, 4);
```

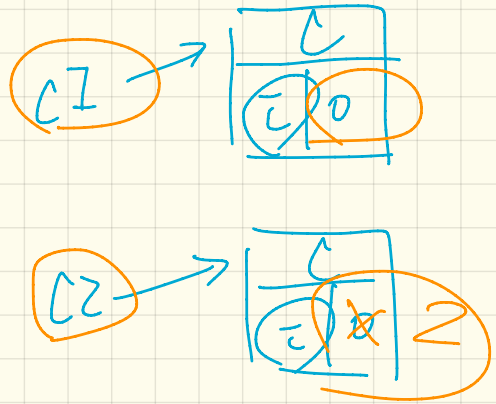

class C {
c1.i
c2.i
→ int i;
non-static variable
each object/instance has its own copy of i.
static int j;
c1.j
c2.j
static variable
all objects/instances share a single/global copy of j.
}


c1.Increment(local(c))
c1
Counter
l
x
1
c2
Counter
l
0


Counter
g
x
1
global, shared by all Counter objects

Wednesday March 20

Lecture 20



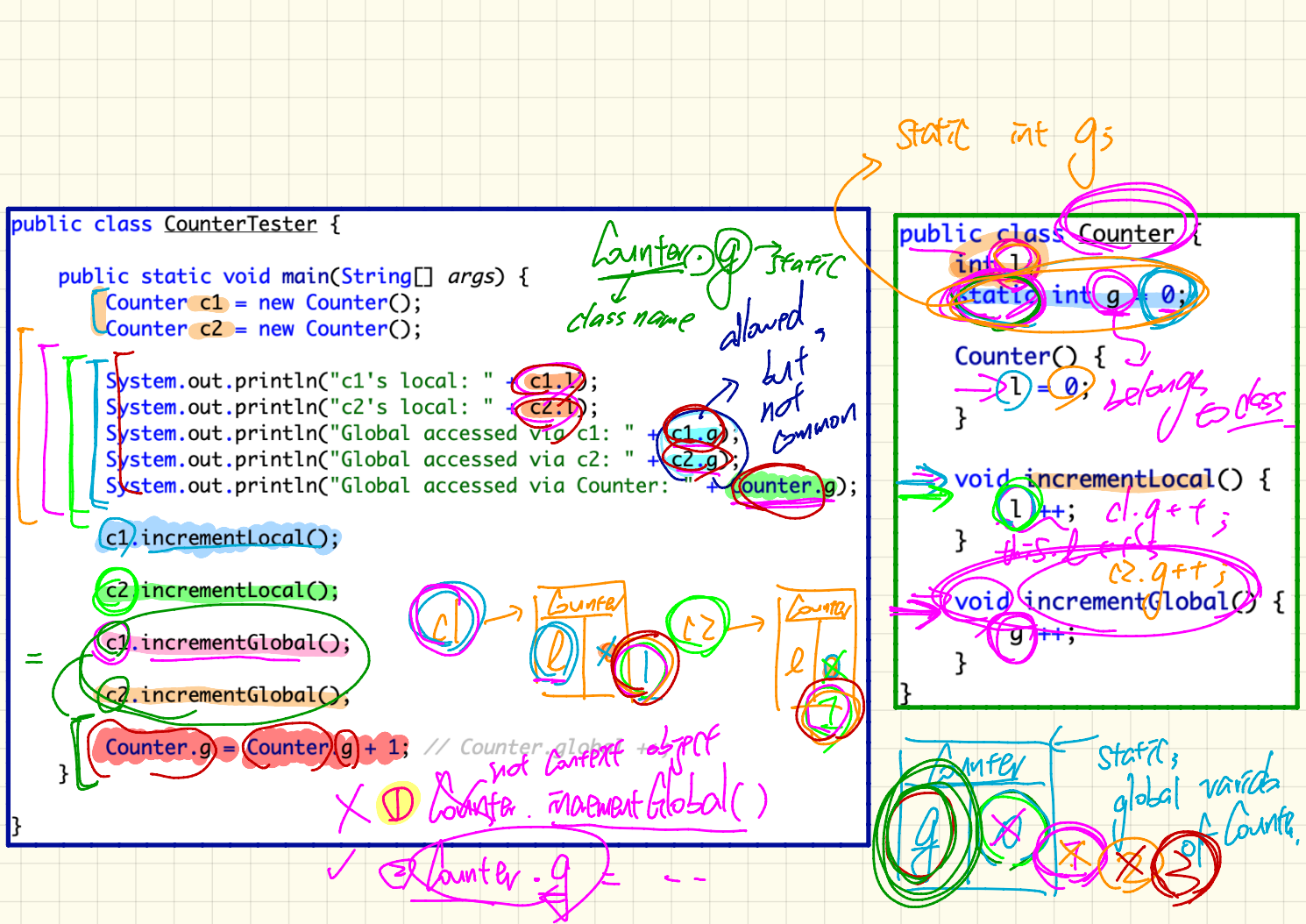

```
class C {
    int i;
    static int j;
}
```

non-static

c1.i

c2.i

all objects of type C share a single copy of j.

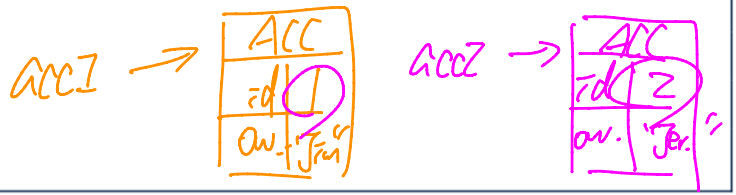

public class CounterTester {
public static void main(String[] args) {
Counter c1 = new Counter();
Counter c2 = new Counter();
System.out.println("c1's local: " + c1.l);
System.out.println("c2's local: " + c2.l);
System.out.println("Global accessed via c1: " + c1.g);
System.out.println("Global accessed via c2: " + c2.g);
System.out.println("Global accessed via Counter: " + Counter.g);
c1.incrementLocal();
c2.incrementLocal();
c1.incrementGlobal();
c2.incrementGlobal();
Counter.g = Counter.g + 1; // Counter.global
}
}
Counter.g
static
class name
allowed,
but
not
common
=
c1
Counter
l
g
1
c2
Counter
l
g
1
not context object
Counter.incrementGlobal()
Counter.g
static int g;
public class Counter {
int l;
static int g = 0;
Counter() {
l = 0;
}
belongs to class
void incrementLocal() {
l++;
c1.g++;
c2.g++;
}
void incrementGlobal() {
g++;
}
}
Counter
g
1
2
3
static;
global variable
Counter

Managing Account ID: Manually

```
class Account {
  int id;
  String owner;
  Account(int id, String owner) {
    this.id = id;
    this.owner = owner;
  }
}
```

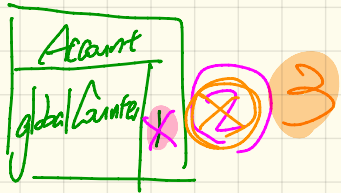
non-static

non-static




```
class AccountTester {
  Account acc1 = new Account(1, "Jim");
  Account acc2 = new Account(2, "Jeremy");
  System.out.println(acc1.id != acc2.id);
}
```

Managing Account ID: Automatically



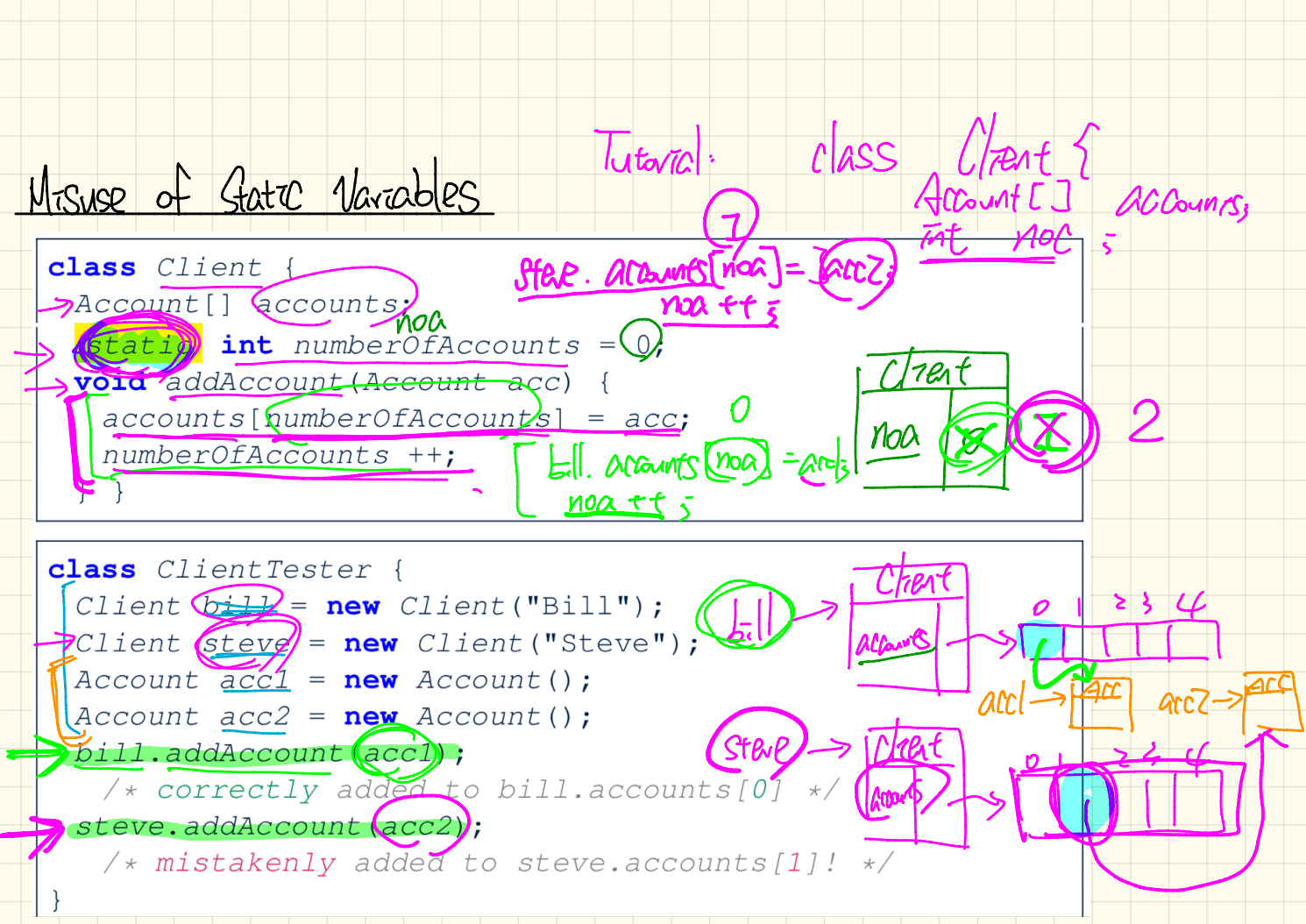

```
class Account {
  static int globalCounter = 1;
  int id; String owner;
  Account(String owner) {
    this.id = globalCounter; globalCounter ++;
    this.owner = owner; } }
```

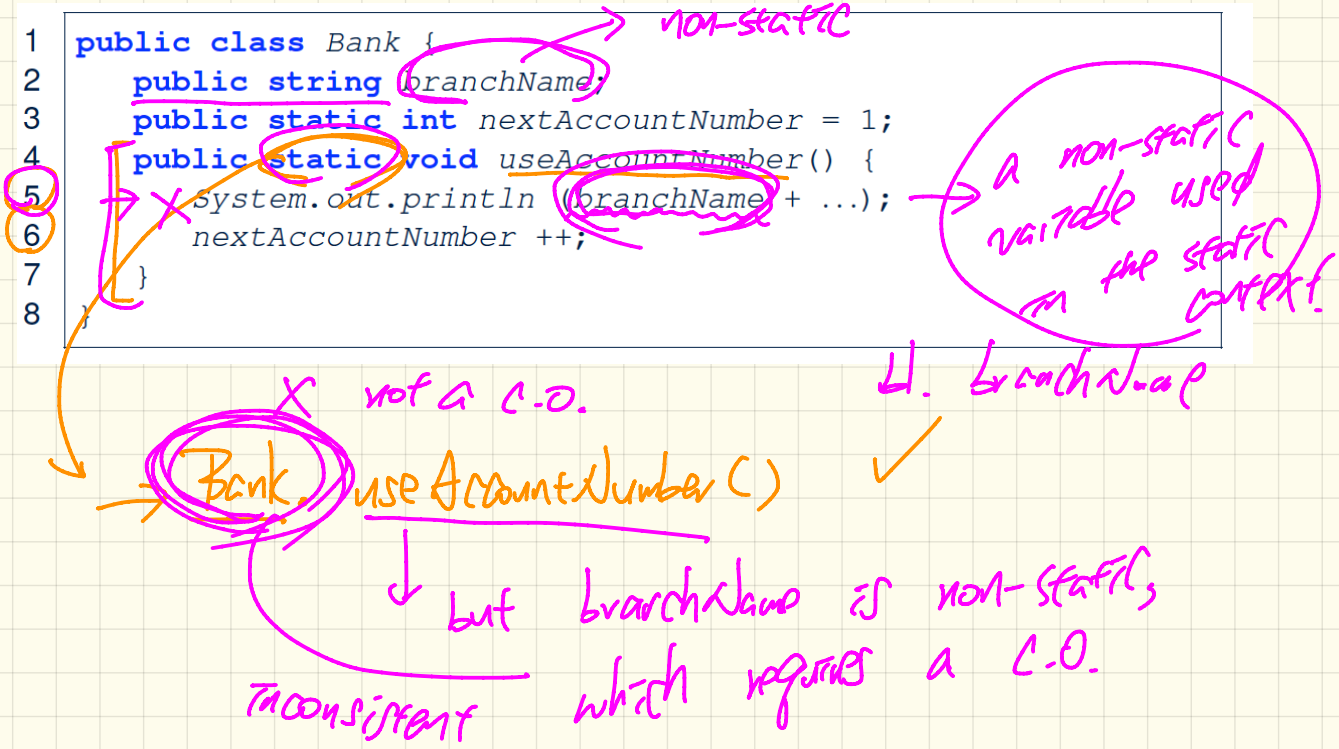

```
class AccountTester {
  Account acc1 = new Account("Jim");
  Account acc2 = new Account("Jeremy");
  System.out.println(acc1.id != acc2.id); }
```

Misuse of Static Variables

```
class Client {
  Account[] accounts;
  static int numberOfAccounts = 0;
  void addAccount(Account acc) {
    accounts[numberOfAccounts] = acc;
    numberOfAccounts ++;
  } }
```

```
class ClientTester {
  Client bill = new Client("Bill");
  Client steve = new Client("Steve");
  Account acc1 = new Account();
  Account acc2 = new Account();
  bill.addAccount(acc1);
    /* correctly added to bill.accounts[0] */
  steve.addAccount(acc2);
    /* mistakenly added to steve.accounts[1]! */
}
```




1 public class Bank {
2 public string branchName;
3 public static int nextAccountNumber = 1;
4 public static void useAccountNumber() {
5 System.out.println (branchName + ...);
6 nextAccountNumber ++;
7 }
8 }
non-static
a non-static variable used in the static context
Bank.useAccountNumber()
not a C.O.
but branchName is non-static, which requires a C.O.
inconsistent

Monday March 25

Lecture 21

Use of Static Variables: Common Errors

```
1  public class Bank {
2      public string branchName;
3      public static int nextAccountNumber = 1;
4      public static void useAccountNumber() {
5          System.out.println (branchName + ...);
6          nextAccountNumber ++;
7      }
8  }
```

non-static

static

this.branchName

Bank → does not store valid address of Bank object ∴ not compile

Bank.nextAccountNumber

Bank.useAccountNumber()

Bank: class name, not C.O.

Bank b = new Bank();

context object: b.branchName

Use of Static Variables: Common Errors

```
1  public class Bank {
2      public string branchName;
3      public static int nextAccountNumber = 1;
4      public static void useAccountNumber() {
5          System.out.println (branchName + ...);
6          nextAccountNumber ++;
7      }
8  }
```

static

```
1  public class Bank {
2      public string branchName;
3      public static int nextAccountNumber = 1;
4      public static void useAccountNumber() {
5          System.out.println (branchName + ...);
6          nextAccountNumber ++;
7      }
8  }
```

Fix 1: eliminate all non-static variables from static methods.

not appropriate

Fix 2: change all non-static variables to static

compile but this means all the Bank objects have/share the same branchName

Programming Pattern: Mutator

```
class PointCollector {
  Point[] points; int nop; /* number of points */
  PointCollector() { points = new Point[100]; }
  void addPoint(double x, double y) {
    points[nop] = new Point(x, y); nop++; }
```

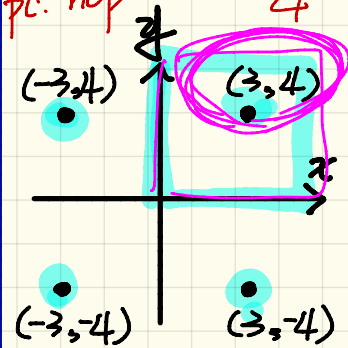
```
class PointCollectorTester {
  public static void main(String[] args) {
    PointCollector pc = new PointCollector();
    System.out.println(pc.nop);  /* 0 */
    pc.addPoint(3, 4);
    System.out.println(pc.nop);  /* 1 */
    pc.addPoint(-3, 4);
    System.out.println(pc.nop);  /* 2 */
    pc.addPoint(-3, -4);
    System.out.println(pc.nop);  /* 3 */
    pc.addPoint(3, -4);
    System.out.println(pc.nop);  /* 4 */
```

Programming Pattern: Accessor

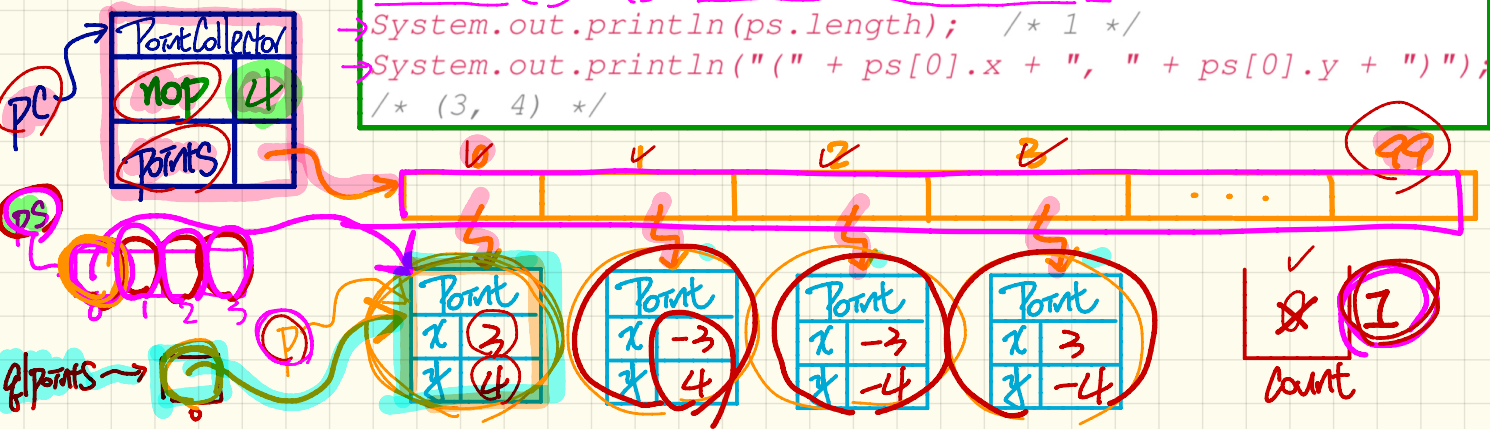
```
Point[] getPointsInQuadrantI() {
  Point[] ps = new Point[nop];
  int count = 0; /* number of points in Quadrant I */
  for(int i = 0; i < nop; i ++) {
    Point p = points[i];
    if(p.x > 0 && p.y > 0) { ps[count] = p; count ++; } }
  Point[] q1Points = new Point[count];
  /* ps contains null if count < nop */
  for(int i = 0; i < count; i ++) { q1Points[i] = ps[i]; }
  return q1Points;
} }
```

return ps; ✗ return points; ✗

pc.points.length 100
pc.nop 4

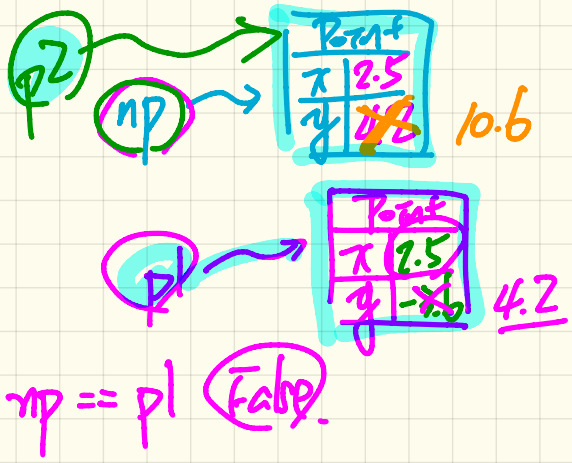


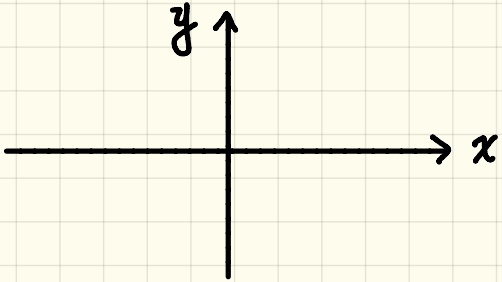

```
Point[] ps = pc.getPointsInQuadrantI();
System.out.println(ps.length);  /* 1 */
System.out.println("(" + ps[0].x + ", " + ps[0].y + ")");
/* (3, 4) */
```



Return Type: Reference Type

```
class Point {
  Point(double x, double y) {…}

  void moveUpBy(double units) {
    this.y = this.y + units;
  }

  Point movedUpBy(double units) {
    Point np = new Point(this.x, this.y);
    np.moveUpBy(units);
    return np;
  }
}
```






```
class PointTester {
  static void main(String[] args) {
    Point p1 = new Point(2.5, -3.6);
    p1.moveUp(7.8);
    Point p2 = p1.movedUpBy(6.4);
    System.out.println(p1 == p2);
  }
}
```

False.

Wednesday March 27

Lecture 22

API: Math

utilities class

overloading: multiple methods have same name, but distinct types.

Modifier and Type	Method and Description
static double	abs(double a) (v1) Returns the absolute value of a double value.
static float	abs(float a) (v2) Returns the absolute value of a float value.
static int	abs(int a) (v3) Returns the absolute value of an int value.
static long	abs(long a) (v4) Returns the absolute value of a long value.

call the method using class name. abs is an accessor

identical method name

return type.

Math.abs(-2.0) → 2.0 (param. type)

Definition

API of Math

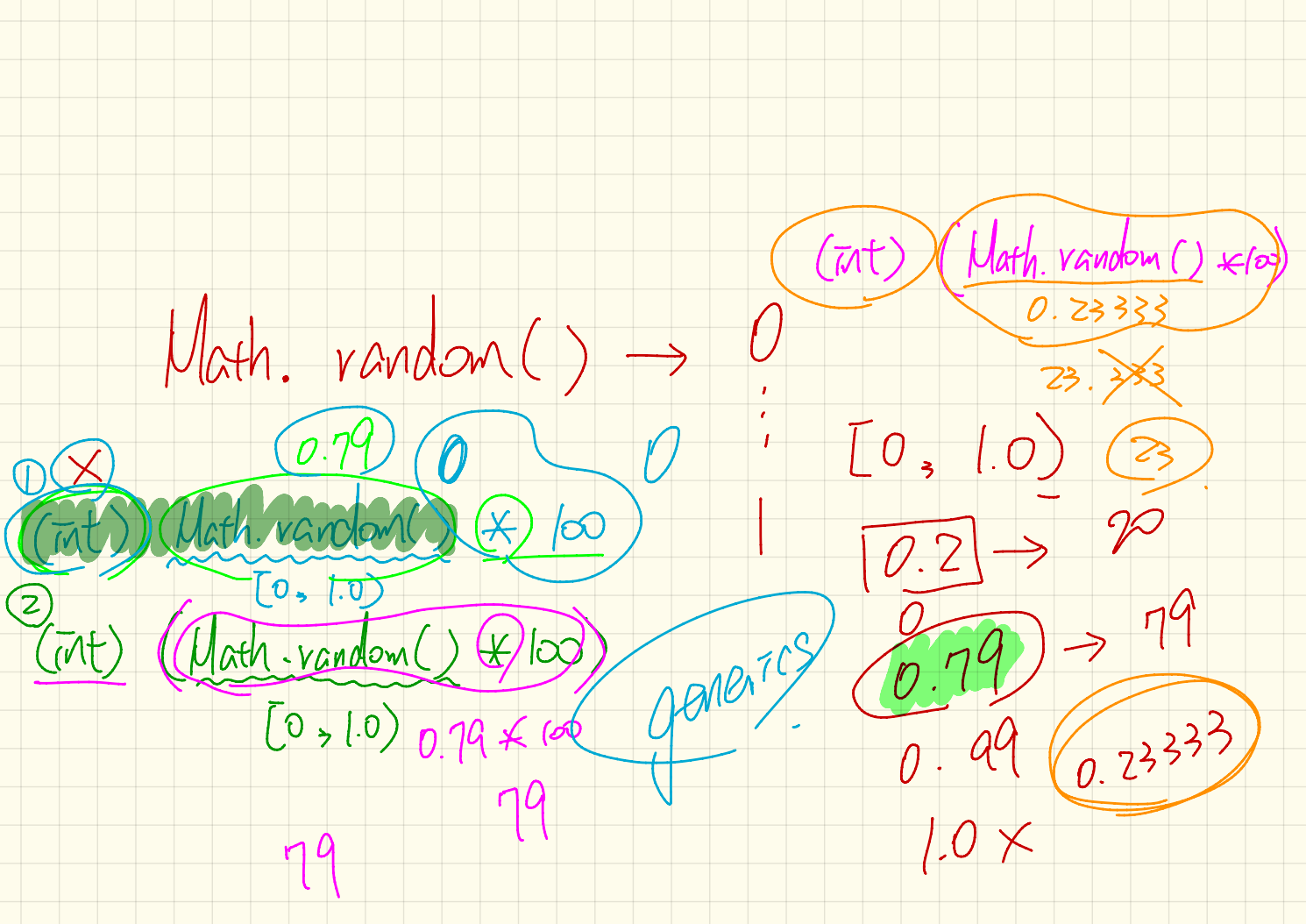
static double abs(double x)

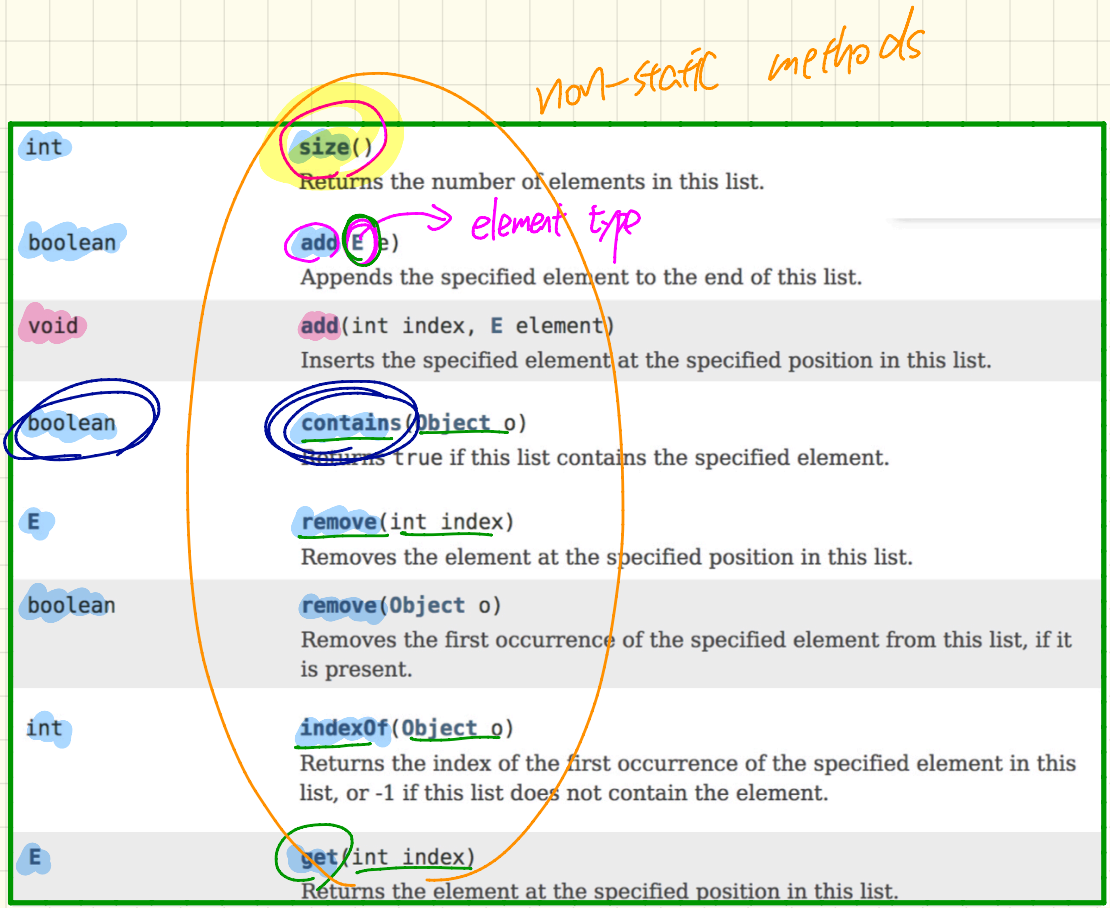
parameter

Usage

Math.abs(-2.4)

argument.


Math. random() → 0 ⋮ 1
(int) (Math. random() * 100)
0.79
0
0
[0, 1.0)
① X
(int) (Math. random() * 100)
② (int) (Math. random() * 100)
[0, 1.0) 0.79 * 100
79
79
generics
(int) (Math. random() * 100)
0.23333
23.333
[0, 1.0)
23
0.2 → 20
0
0.79 → 79
0.99
1.0 X
0.23333



int	**size**() Returns the number of elements in this list.
boolean	**add**(E e) Appends the specified element to the end of this list.
void	**add**(int index, E element) Inserts the specified element at the specified position in this list.
boolean	**contains**(Object o) Returns true if this list contains the specified element.
E	**remove**(int index) Removes the element at the specified position in this list.
boolean	**remove**(Object o) Removes the first occurrence of the specified element from this list, if it is present.
int	**indexOf**(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
E	**get**(int index) Returns the element at the specified position in this list.

API: ArrayList

ArrayList<String> list1;

ArrayList<Person> list2;

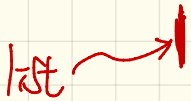
ArrayList<String> list = new ...

list.size()

list.length ✗ does not compile

∴ length is not part of the API of ArrayList

Use of ArrayList




```
ArrayList<String> list = new ArrayList<String>();
println(list.size());
println(list.contains("A"));
println(list.indexOf("A"));
list.add("A");
list.add("B");
println(list.contains("A")); println(list.contains("B")); println(list.contains("C"));
println(list.indexOf("A")); println(list.indexOf("B")); println(list.indexOf("C"));
list.add(1, "C");
println(list.contains("A")); println(list.contains("B")); println(list.contains("C"));
println(list.indexOf("A")); println(list.indexOf("B")); println(list.indexOf("C"));
list.remove("C");
println(list.contains("A")); println(list.contains("B")); println(list.contains("C"));
println(list.indexOf("A")); println(list.indexOf("B")); println(list.indexOf("C"));

for(int i = 0; i < list.size(); i ++) {
  println(list.get(i));
}
```

Monday April 1

Lecture 23

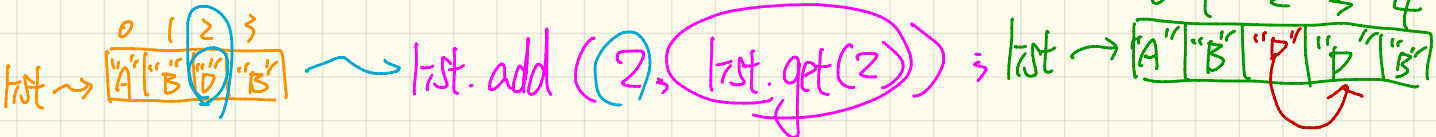
Lab 7 (not to be graded no submission)

§

API

M, T, W no attendance of lab sessions

(W)



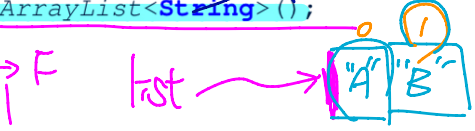
API: ArrayList

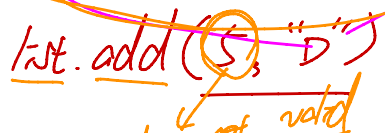
int	**size()** Returns the number of elements in this list.
boolean	**add(E e)** Appends the specified element to the end of this list.
void	**add(int index, E element)** Inserts the specified element at the specified position in this list.
boolean	**contains(Object o)** Returns true if this list contains the specified element.
E	**remove(int index)** Removes the element at the specified position in this list.
boolean	**remove(Object o)** Removes the first occurrence of the specified element from this list, if it is present.
int	**indexOf(Object o)** Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
E	**get(int index)** Returns the element at the specified position in this list.

when duplicates exist

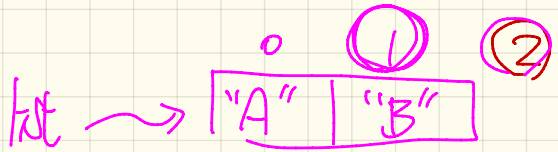
Use of ArrayList


```
ArrayList<String> list = new ArrayList<String>();
println(list.size());
println(list.contains("A"));
println(list.indexOf("A"));
list.add("A");
list.add("B");
println(list.contains("A")); println(list.contains("B")); println(list.contains("C"));
println(list.indexOf("A")); println(list.indexOf("B")); println(list.indexOf("C"));
list.add(1, "C");
println(list.contains("A")); println(list.contains("B")); println(list.contains("C"));
println(list.indexOf("A")); println(list.indexOf("B")); println(list.indexOf("C"));
list.remove("C");
println(list.contains("A")); println(list.contains("B")); println(list.contains("C"));
println(list.indexOf("A")); println(list.indexOf("B")); println(list.indexOf("C"));

for(int i = 0; i < list.size(); i ++) {
  println(list.get(i));
}
```





ArrayList<String> list = new ArrayList<>();



add to end "D"

(Approach 1)

list.add("D") void add(int index, element)

insert to the beginning: list.add(0, __)

insert to the middle: list.add(i, __)

i = 1, 2, ..., list.size()-1

list.add(i, __)

i → list.size() exception.

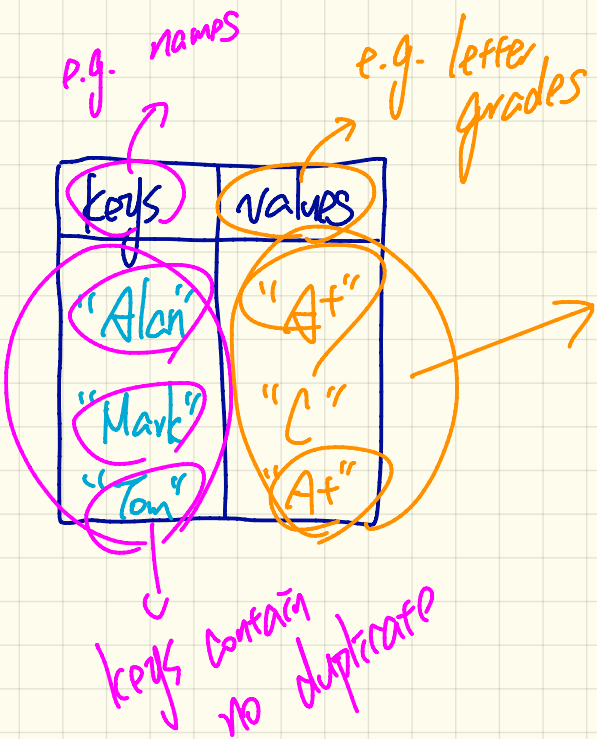
insert to the end: list.add(list.size(), __)

int	**size**() Returns the number of keys in this hashtable.
boolean	**containsKey**(**Object** key) Tests if the specified object is a key in this hashtable.
boolean	**containsValue**(**Object** value) Returns true if this hashtable maps one or more keys to this value.
V	**get**(**Object** key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
V	**put**(**K** key, **V** value) Maps the specified key to the specified value in this hashtable.
V	**remove**(**Object** key) Removes the key (and its corresponding value) from this hashtable.

API: HashTable

Hash table

a collection of entries
- key
- value



Wednesday April 3

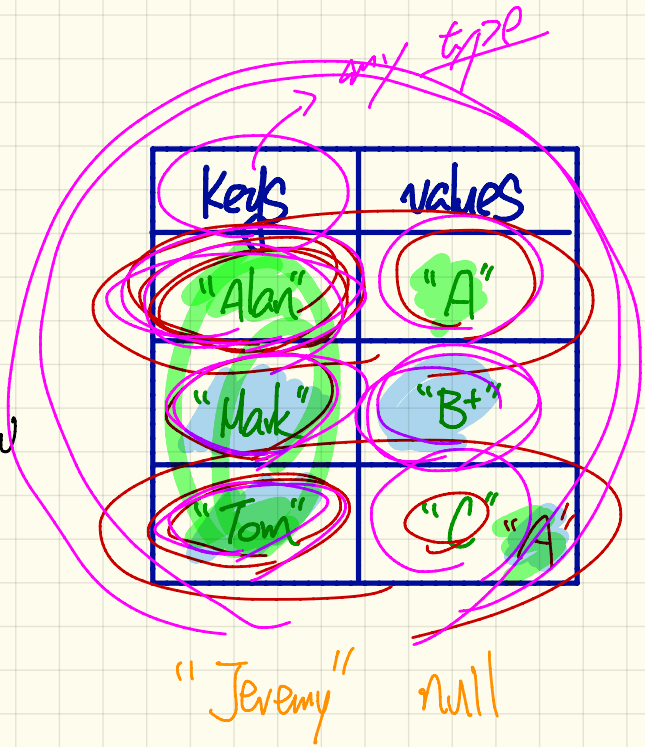
Lecture 24

- Quiz 5 , 6

- Lab Test 3

Hashtable

- 2-column table
- keys contain no duplicates
- values may contain duplicates
- a key is used to identify a row



API: HashTable

int	**size()** Returns the number of keys in this hashtable.
boolean	**containsKey(Object key)** Tests if the specified object is a key in this hashtable.
boolean	**containsValue(Object value)** Returns true if this hashtable maps one or more keys to this value.
V	**get(Object key)** Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
V	**put(K key, V value)** Maps the specified key to the specified value in this hashtable.
V	**remove(Object key)** Removes the key (and its corresponding value) from this hashtable.

Exam

LAS A
LAS B
LAS C

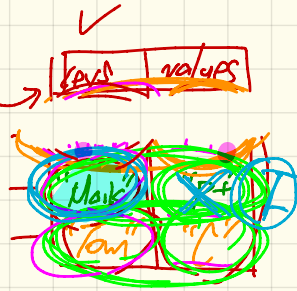
} You will be assigned to a room based on LAST NAMES.

Use of HashTable

key

value

```
Hashtable<String, String> grades = new Hashtable<String, String>();
System.out.println("Size of table: " + grades.size());
System.out.println("Key Alan exists: " + grades.containsKey("Alan"));
System.out.println("Value B+ exists: " + grades.containsValue("B+"));
grades.put("Alan", "A");
grades.put("Mark", "B+");
grades.put("Tom", "C");
System.out.println("Size of table: " + grades.size());
System.out.println("Key Alan exists: " + grades.containsKey("Alan"));
System.out.println("Key Mark exists: " + grades.containsKey("Mark"));
System.out.println("Key Tom exists: " + grades.containsKey("Tom"));
System.out.println("Key Simon exists: " + grades.containsKey("Simon"));
System.out.println("Value A exists: " + grades.containsValue("A"));
System.out.println("Value B+ exists: " + grades.containsValue("B+"));
System.out.println("Value C exists: " + grades.containsValue("C"));
System.out.println("Value A+ exists: " + grades.containsValue("A+"));
System.out.println("Value of existing key Alan: " + grades.get("Alan"));
System.out.println("Value of existing key Mark: " + grades.get("Mark"));
System.out.println("Value of existing key Tom: " + grades.get("Tom"));
System.out.println("Value of non-existing key Simon: " + grades.get("Simon"));
grades.put("Mark", "F");
System.out.println("Value of existing key Mark: " + grades.get("Mark"));
grades.remove("Alan");
System.out.println("Key Alan exists: " + grades.containsKey("Alan"));
System.out.println("Value of non-existing key Alan: " + grades.get("Alan"));
```



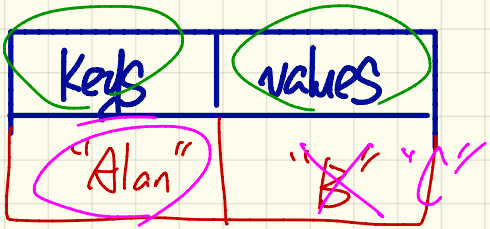
Friday April 5

Review Lecture

Hashtable

- 2-column table
- keys contain no duplicates
- values may contain duplicates
- a key is used to identify a row

```
grades.put("Alan", "B");
grades.put("Alan", "C");
```



Use of HashTable

keys

values

Hashtable<String, Birthday> book = new

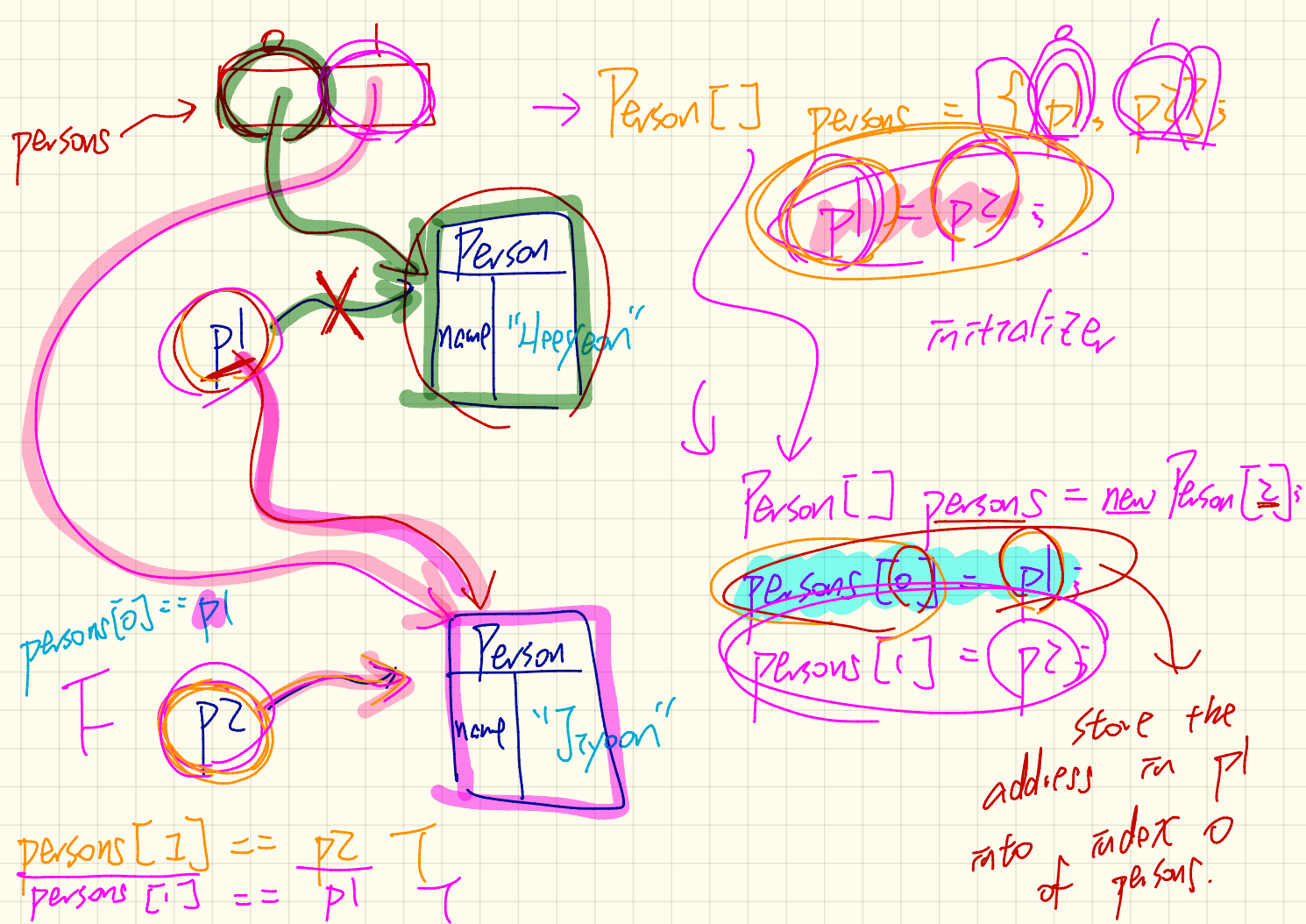
name

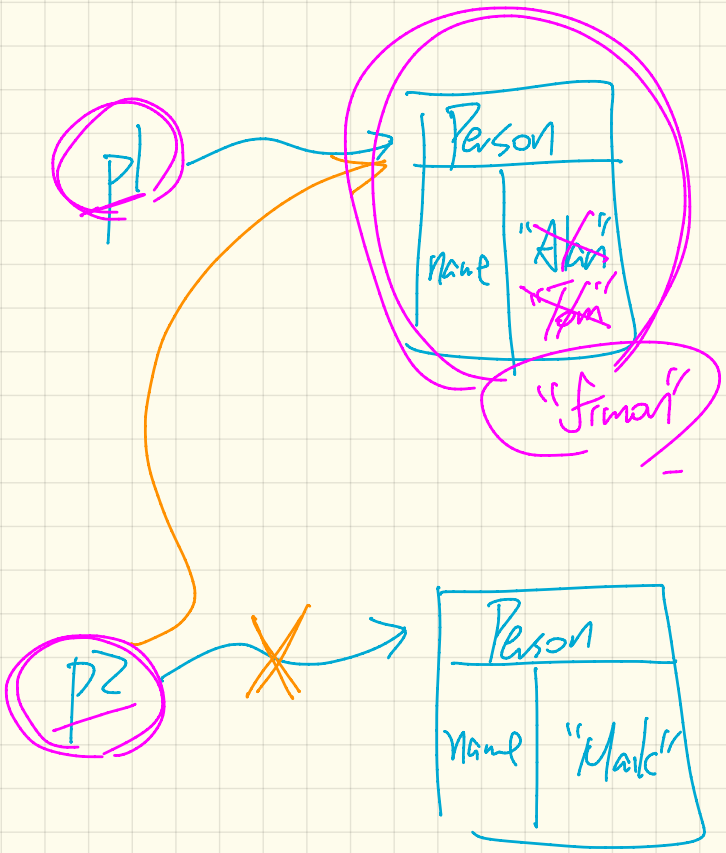
Birthday

ks

vs

```
Hashtable<String, String> grades = new Hashtable<String, String>();
System.out.println("Size of table: " + grades.size());
System.out.println("Key Alan exists: " + grades.containsKey("Alan"));
System.out.println("Value B+ exists: " + grades.containsValue("B+"));
grades.put("Alan", "A");
grades.put("Mark", "B+");
grades.put("Tom", "C");
System.out.println("Size of table: " + grades.size());
System.out.println("Key Alan exists: " + grades.containsKey("Alan"));
System.out.println("Key Mark exists: " + grades.containsKey("Mark"));
System.out.println("Key Tom exists: " + grades.containsKey("Tom"));
System.out.println("Key Simon exists: " + grades.containsKey("Simon"));
System.out.println("Value A exists: " + grades.containsValue("A"));
System.out.println("Value B+ exists: " + grades.containsValue("B+"));
System.out.println("Value C exists: " + grades.containsValue("C"));
System.out.println("Value A+ exists: " + grades.containsValue("A+"));
System.out.println("Value of existing key Alan: " + grades.get("Alan"));
System.out.println("Value of existing key Mark: " + grades.get("Mark"));
System.out.println("Value of existing key Tom: " + grades.get("Tom"));
System.out.println("Value of non-existing key Simon: " + grades.get("Simon"));
grades.put("Mark", "F");
System.out.println("Value of existing key Mark: " + grades.get("Mark"));
grades.remove("Alan");
System.out.println("Key Alan exists: " + grades.containsKey("Alan"));
System.out.println("Value of non-existing key Alan: " + grades.get("Alan"));
```


persons
Person[]
persons = {p1, p2};
p1 = p2;
Initialize
Person
name "Heeyeon"
p1
Person[] persons = new Person[2];
persons[0] = p1;
persons[1] = p2;
persons[0] == p1
Person
name "Jiyoon"
p2
persons[1] == p2
persons[1] == p1
store the address in p1 into index 0 of persons.



→ p1.setName("Tom")

p2 = p1;

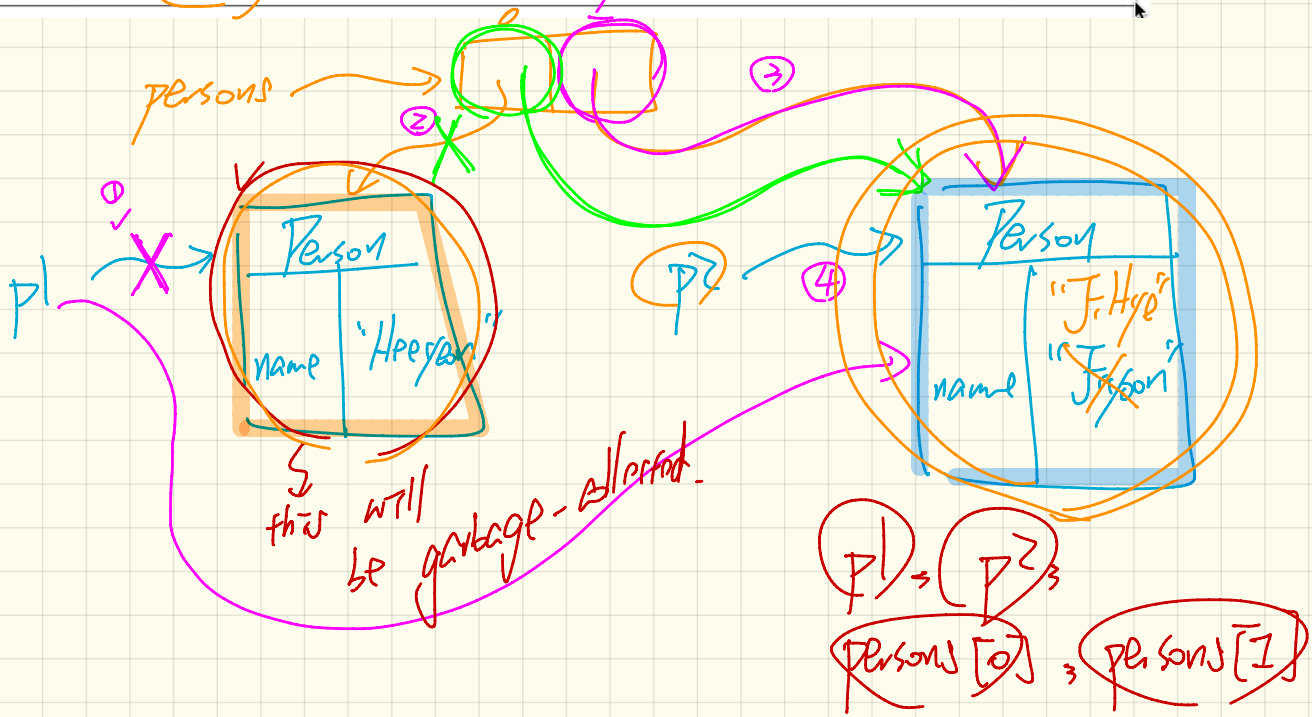
p1 == p2 (T)

p1.setName("Simon")

p2.getName()

↳ "Simon"

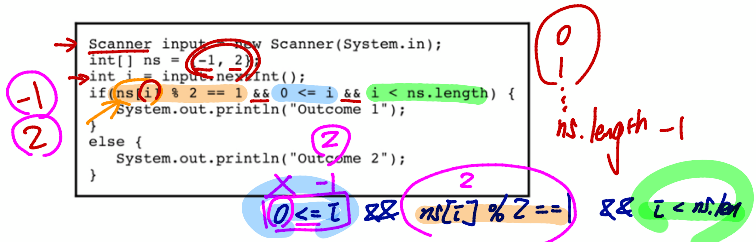
```
1  Person p1 = new Person("Heeyeon");
2  Person p2 = new Person("Jiyoon");
3  Person[] persons = {p1, p2};
4  p1 = persons[1];
5  persons[0] = p2;
6  p2.setName("Jihye");
7  System.out.println(p1.name);
```



Correct answers are in bold green.

Wrong answers are in bold red.

1. [1 mark, id = 111] Consider the following fragment of code:



When running the above program, which of the following value(s) of variable **i** will result in an *ArrayIndexOutOfBoundsException*? Chose the **best** answer.

A. [id = 1] -1

B. [id = 2] 0

C. [id = 3] 1

D. [id = 4] 2

E. [id = 5] None of the above answers is correct.

F. [id = 6] **More than one of the above answers are correct.**

---

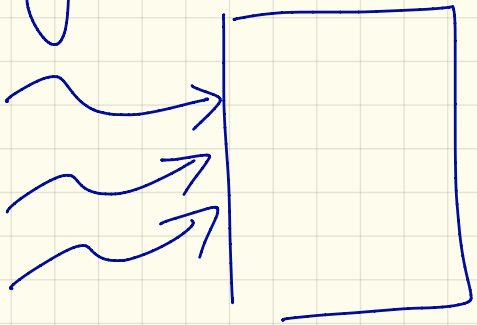
2. [1 mark, id = 211] Consider the following fragment of code:

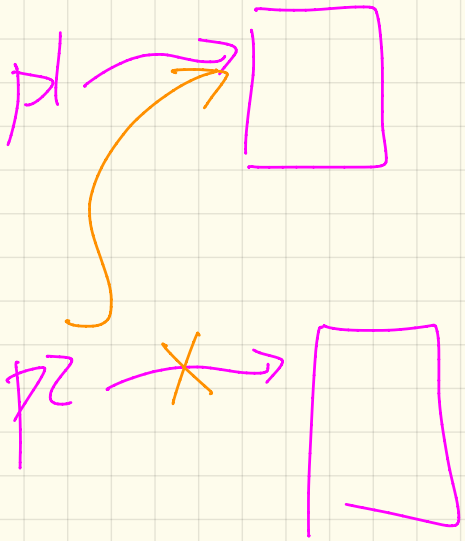
Written Questions (10%)

getFS(0) → {}
(2) → {1, 1}

```
int[] getFibSeq(int i) {

}
```

aliasing





Person p1 = new . . . .

Person p2 = new . . . .

p1 = p2 ;

① p1 = p2

② p2 = p1

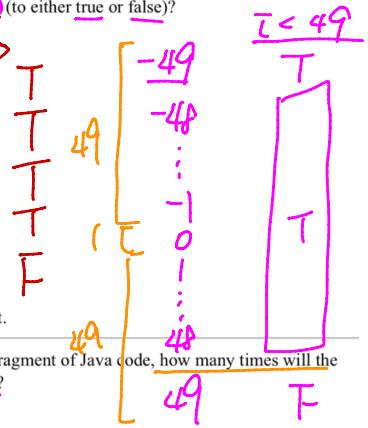
Solutions of EECS1021 Quiz 3 for chiddy00

Correct answers are in bold green.

Wrong answers are in bold red.

1. [1 mark, id = 111] When executing the following fragment of Java code, how many times will the stay condition (i.e., i < 49) of the loop be evaluated (to either true or false)?

```
for(int i = -49; i < 49; i ++) {
  System.out.println("Outcome");
}
```



A. [id = 1] **99**
B. [id = 2] **98**
C. [id = 3] 97
D. [id = 4] 100
E. [id = 5] 101
F. [id = 6] 0
G. [id = 7] 1
H. [id = 8] None of the above answers is correct.

---

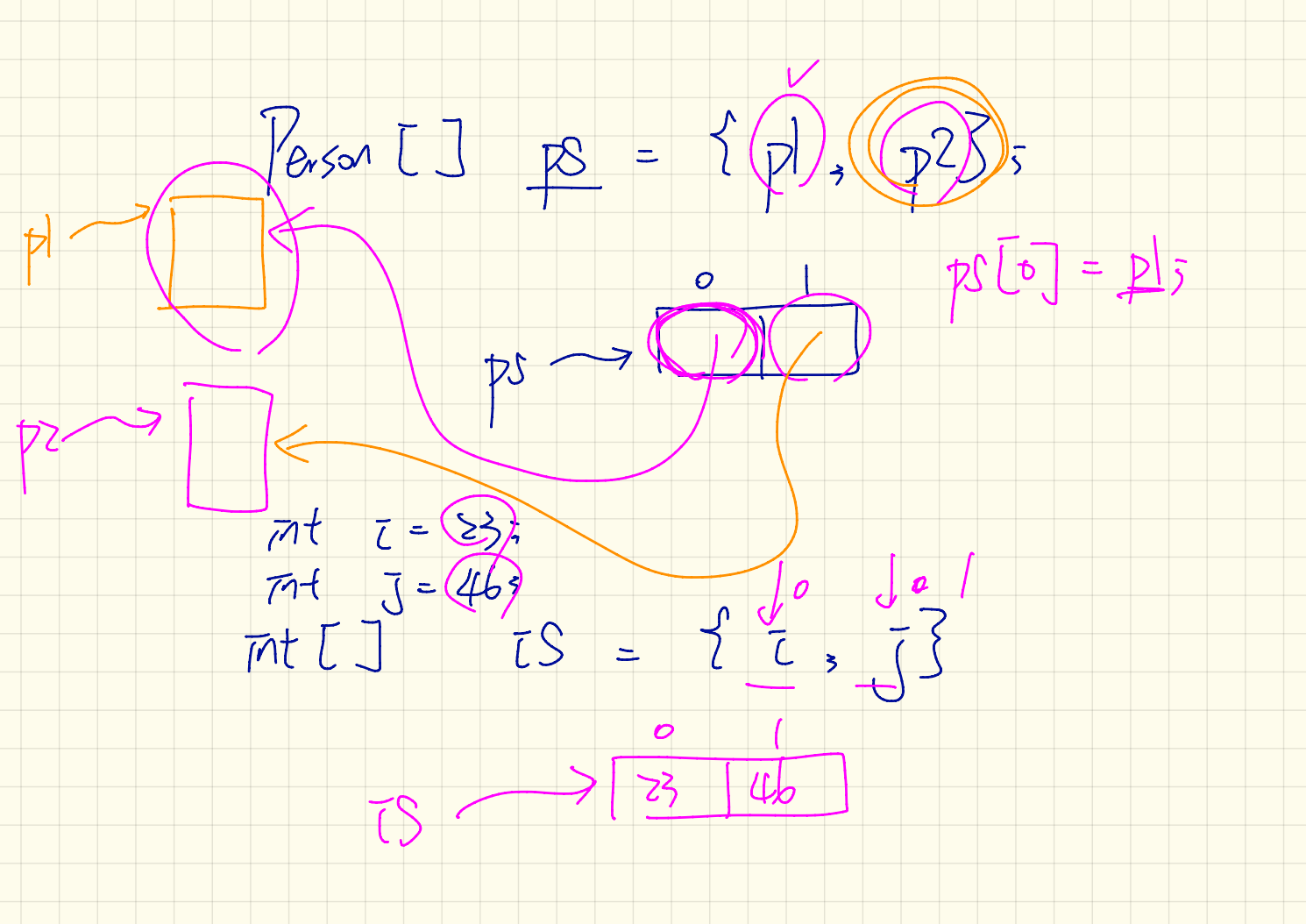
2. [1 mark, id = 211] When executing the following fragment of Java code, how many times will the body of loop (i.e., the print statement) be executed?

```
for(int i = -49; i < 49; i ++) {
  System.out.println("Outcome");
}
```

A. [id = 1] 99
B. [id = 2] **98**
C. [id = 3] **97**
D. [id = 4] 100
E. [id = 5] 101
F. [id = 6] 0
G. [id = 7] 1
H. [id = 8] None of the above answers is correct.

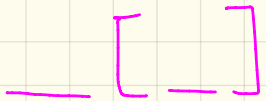
---

3. [1 mark, id = 311] When executing the following fragment of Java code, how many times will the stay condition (i.e., i < 49) of the loop be evaluated to false?

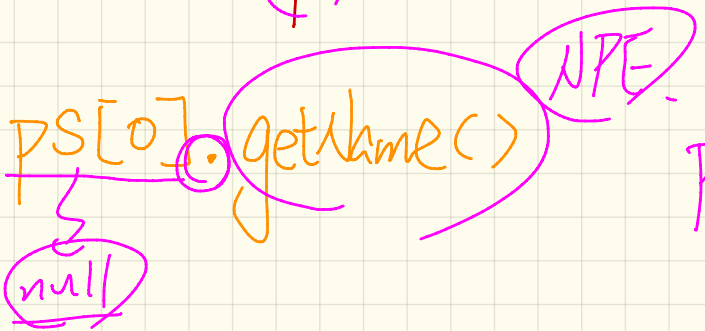

Person[] ps = {p1, p2};
p1
ps
0
1
ps[0] = p1;
p2
int i = 23;
int j = 46;
int[] is = {i, j};
0
1
0
1
is
23
46

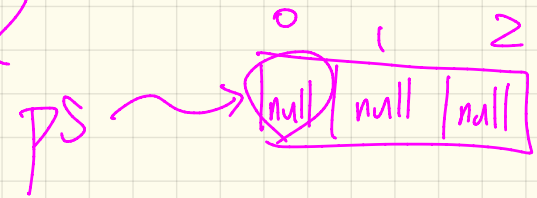
NullPointerException

ArrayIndexOutOfBoundsEx.

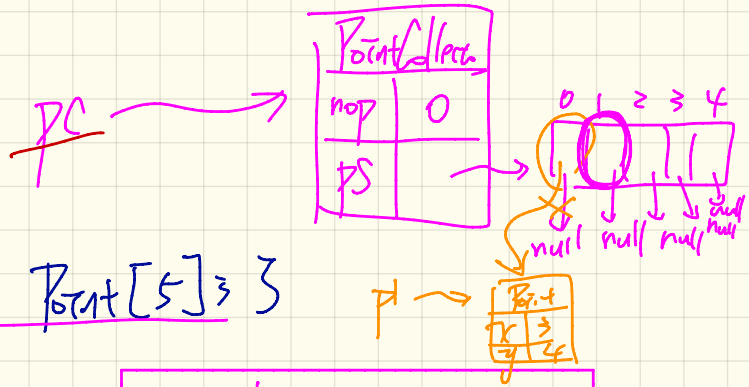


```
Person[] ps = new Person[3];
```





```
class PointCollector {
    Point[] ps;
    int nop;
    PointCollector() { ps = new Point[5]; }
    void addPoint(Point p){
        ps[nop] = p;
        nop++;
    }

    String getDesc() {
        String s = "";
        for (int i = 0; i < ps.length; i++){
            s += ps[i].getX() + "," + ps[i].getY();
        }
        return s;
    }
}
```



this.nop nop

2nd iteration: i == 1 NPE

END OF NOTES

ALL THE BEST !