EECS1022
Programming for Mobile Computing

Winter 2021

Instructor: Jackie Wang

Lecture 1

Part A

Elementary Programming - Development Process

Separation of Concerns

junit_tests

- Expected vs. Actual Values
- Methods
  * calling methods from model
  * containing no print statements
  * assertions

use

model

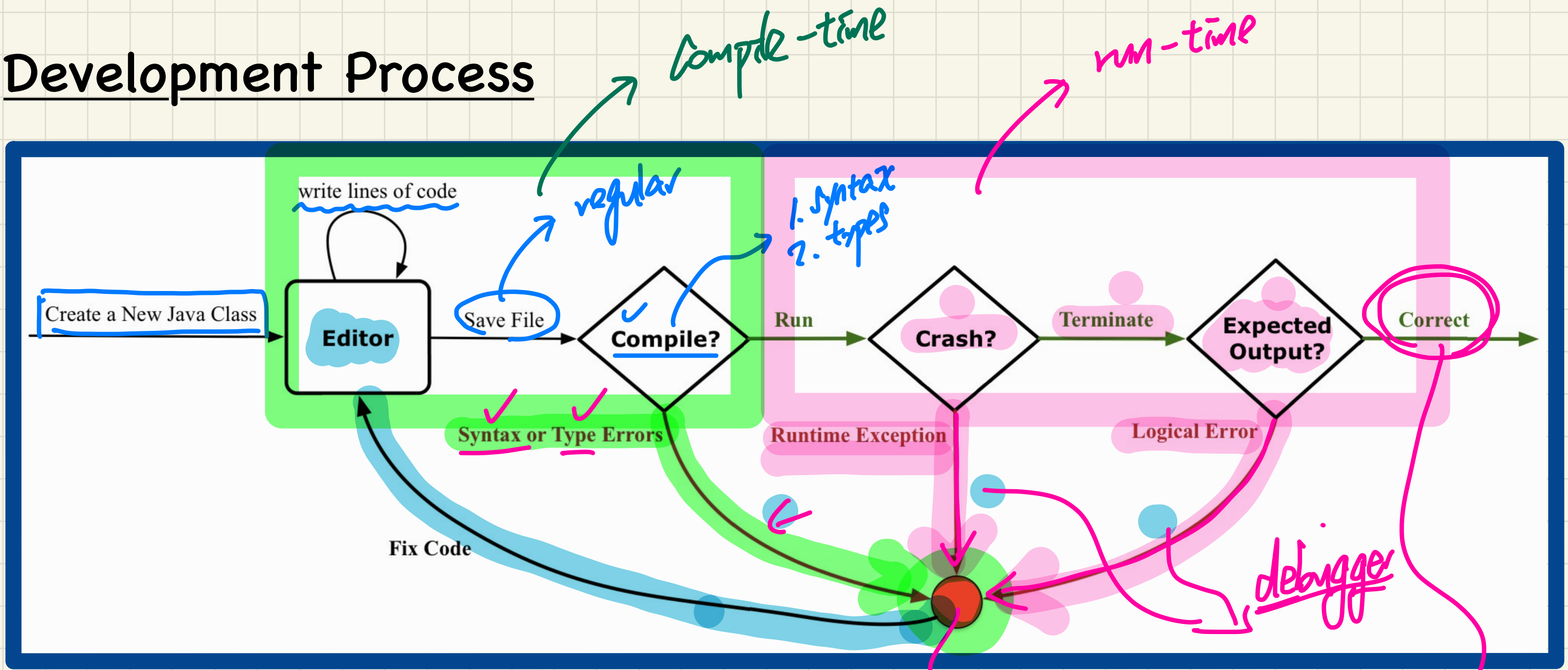
- Classes & Methods
- Methods
  * containing no print statements
  * return statements

use

console_apps

- main method
  * calling methods from model
  * containing print statements
  * containg no return statements

Development Process



error state
(something wrong).

1. compiles
2. terminates
3. output expected

Error at Compile Time: Syntax Errors (1)

CompileTimeSyntaxError1.java

```java
public class CompileTimeSyntaxError1 {
    public static void main(String[] args) {
        // Syntax Error: missing semicolon
        System.out.println("Hello")
    }
}
```

Error at Compile Time: Syntax Errors (2)

CompileTimeSyntaxError2.java

```java
public class CompileTimeSyntaxError2 {
    public static void main(String[] args) {
        // Syntax Error: missing ending double quote
        System.out.println("Hello);
    }
}
```

Error at Compile Time: Syntax Errors (3)

{ } ( )
[ ]

CompileTimeSyntaxError3.java

```
public class CompileTimeSyntaxError3 {
    public static void main(String[] args) {
        System.out.println("Hello");

        /* Error 3: missing ending curly bracket */
}
```

3

Error at Compile Time: Syntax Errors (4)

CompileTimeSyntaxError4.java

```java
public class CompileTimeSyntaxError4 {
    public static void main(String[] args) {
        System.out.println("Hello");

        /* Error 3: extra ending curly bracket */
    }
}
}
```

no opening { to match

Error at Compile Time: Type Errors (1)

CompileTimeTypeError1.java

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

i.

not a number.

✓

*: multiplication

1. Fix: 46

2. Fix: int i = 46;

Error at Compile Time: Type Errors (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);
    }
}
```

int j = i * 2;

undeclared

⇒ unknown.

Error at Run Time: Exception

no compile-time error ⇒ runnable. executable.

RunTimeException.java

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

division

math: undefined

prog: crash.

Error at 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();
    }
}
```

logical error
wrong
formula.

radius * radius * 3.14

1. Compiles
2. terminates without crashing
3. output is wrong.

Document Your Code

Single-Lined Comments: [Eclipse: `Ctrl + /`]

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

Multiple-Lined Comments: [Eclipse: `Ctrl + /`]

```
/* 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.
 */
```

Lecture 1

Part B

Elementary Programming - Literals, Operations

'␣␣' ✗

'' ✗ "" ✓

'␣' → character

"␣" → string.

$\underset{0}{\_}.23$ ✓ $23.\underset{0}{\_}$ ✓

Operator, Operands, Operation

numerical operation

3 * 7

binary operators

numerical

-(24 / 4)

unary op. (numerical)

x < y

relational

relational operation

!(x >= y)

not

unary op. (logical)

logical (boolean) operation

>
<=
>=

¬ negation
!

&& and
|| or

binary, logical.

- overloaded

e.g. -2 (unary)

e.g. 2-3 (binary)

- An operation consists of an operator and one or more operands.
- An operator has one or more applicable operands. (unary vs. binary)
- An operation produces a value~~s~~ of certain type.

↳ op, operands

Division

Case 1

both operands are integers

$23 \,\%\, 4$ → modulo remainder

$\underset{x}{23} \,/\, \underset{y}{4}$ → quotient

5 with remainder $\boxed{3}$

Given two integers x, y

$$\underset{23}{x} == \underset{4}{y} * \underbrace{(x / y)}_{5} + \underbrace{(x \% y)}_{3}$$

Case 2

at least one operand is floating-point

$23.0 \,/\, 4$ → precise result

$23 \,/\, 4.0$

$23.0 \,/\, 4.0$

$\boxed{5.75}$

Lecture 1

Part C

Elementary Programming -
Data Types
Assignments, Constants vs. Variables

Data Type Declarations

int i = ?

double d = ?

boolean b = ?

char c = ?

String s = ?

data type

variable names.

Consequence of declaring variable with name i of type int:

At runtime, only integer values can be stored in i.

i = "1022" ×

once declared, cannot change the type of a variable.

once initialized, ~~cannot be reassigned.~~

type of named constant

name of constant

initialization of named constant (mandatory)

final double pi $\boxed{= 3.14}$;

"23.4" ✗

'a' ✗

incompatible ⇒ type errors.

double radius $\boxed{= 23.4}$;

name of variable

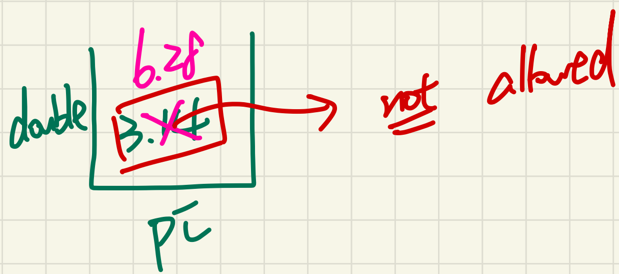
initialization of variable (optional)

Constant: Initialization vs. Re-Assignments

ConstantCannotBeReassigned.java

```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;
    }
}
```

re-assignment



Assignment: Change of Stored Value

trace.

- type
- target LHS
- source RHS.

LHS target of assignment

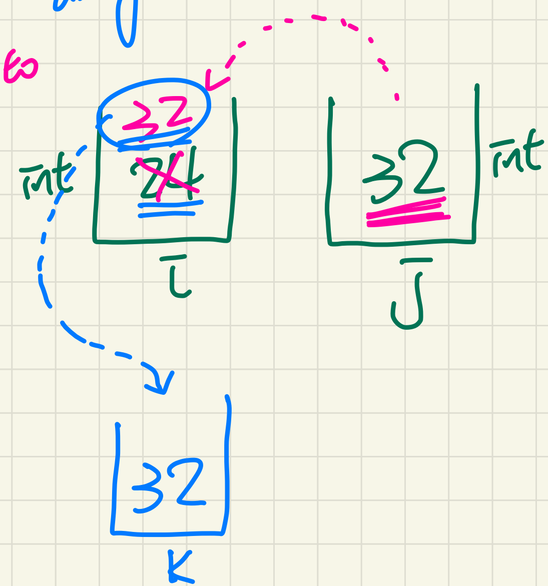
assignment

```
1  int i = 24;
2  int j = 32;
3  i = j;
4  int k = i;
```

RHS source of assignment

re-assign value of i

copy and store into



= → assignment operator.

assignment

== → relational operator

↳ T or F

equal (value comparison)

Lecture 1

Part D

Elementary Programming -
Variable Re-Assignments
Expressions, Type Correctness

Variable: Initialization vs. Re-Assignments

VariableCanBeReassigned.java

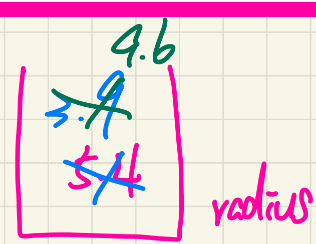
```java
public class VariableCanBeReassigned {
    public static void main(String[] args) {
        /* A variable can be initialized. */
        double radius = 5.4;
        System.out.println("Radius is: " + radius);

        /* 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);
    }
}
```

5.4

3.9

3.9

9.6



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"

Pi is 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);
```

Counter value is 1

~~1~~ 2

counter

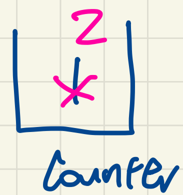
Counter value is 2

Common Mistake: Declaring the Same Variable More Than Once

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

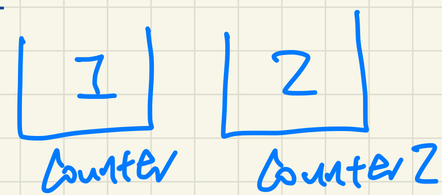
Fix 1: Only Keep the 1st Declaration

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



Fix 2: Declare a New Variable

```
int counter = 1;
int counter2 = 2;
```



Common Mistake: Using a Variable Before Declaring It

declaration

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

x

x

①

("YORK " + "University") * (46 % 4)

concat.

string.

②

number

Overall not type correct even though sub-expressions ① ② are type-correct.

Expressions (1)

	(1 + 2) * (23 % 5)	"Hello " + "world"
Type Correct?	YES. 9	YES "Hello World"
	"Hello " * "world"	"46" % "4"
Type Correct?	No. concat.	No.

	"Hello " + 3 + 2	"Hello " + (3 + 2)
Type Correct?	YES. "Hello32"	YES. "Hello 5"
	"Hello " + 3 + 2 * 2	"Hello " + "3" * 2
Type Correct?	YES "Hello34"	"Hello 3" * 2 No.

Expressions (2)

T.C. 2.

$\boxed{\text{"LaLa " + "land"}}$ * $\boxed{(46 \% 4)}$

"LaLa land"

No. not t.c.

T.C. 2

$\boxed{\text{"LaLa " + "land"}}$ + $\boxed{(46 \% 4)}$

"Lala land"

Yes. its t.c.

concat

"Lala land 2"

Lecture 1

Part E

Elementary Programming - Coercion vs. Casting

Automatic Coercion: int to double

13.5 3.0 coerced int double

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

fractional part present

int int

9

coerced

9.0

value2

However, does the following work?

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

13.5 - X not compile.

no fractional part.

coerced to 3.0

not compatible with int.

Fix

extract the integral part.

13

value1

Manual Casting: double to int

Case 1: double to int

① `int value1 = 3.1415926;` ✗

② `int value4 = (int) 3.1415926;` ✓

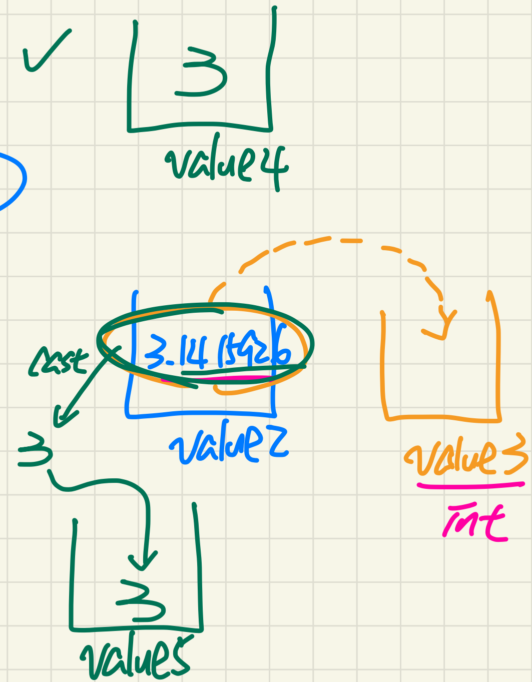
3 3

③ `double value2 = 3.1415926;`

④ `int value3 = value2;` ✗

⑤ `int value5 = (int) value2;` ✓

3 3



Manual Casting: int to double

Case 1: int to double

(double) 1 / 2) equivalent
((double) 1) / 2

1 % 2 → 1

0.5

double v1 = 1; coerced to 1.0

print(v1 / 2) → 0.5

```
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 */
```

int v2 = 1; no coercion

print(v2/2) → 0.

cast 0.0

((double) 1) → 1.0

3 * (2 + 3) higher

precedence

Exercise: Tracing Program

Consider the following Java code:

```
double d1 = 3.1415926;
System.out.println("d1 is " + d1);
double d2 = d1;
System.out.println("d2 is " + d2);
int i1 = (int) d1;
System.out.println("i1 is " + i1);
d2 = i1 * 5;
System.out.println("d2 is " + d2);
```

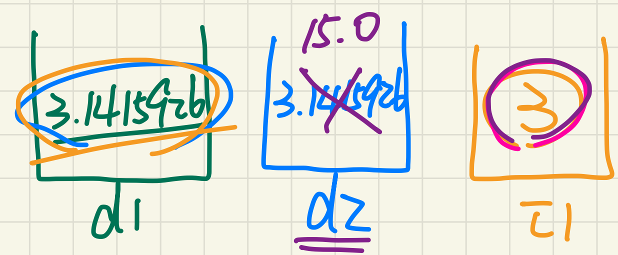
3.

15. → coerced to 15.0.

→ no coercion

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

d1 is 3.1415926
d2 is 3.1415926
i1 is 3
d2 is 15.0



Exercise: Type Correctness

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

```
double d1 = 23;
int i1 = 23.6;
String s1 = ' ';
char c1 = " ";
```

coerced to 23.0

23.0

d1

char

string

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

23

23

i1

69.0

d1

69 → coerced to 69.0

92.0

s d.

not t.c.

coerced to 23.0

"La "

s1

"La La Land"

s2

Lecture 1

Part F

Elementary Programming -
Augmented Assignments
Escape Sequences

Augmented Assignments

150
100
balance

50
deposit

- You very often want to increment or decrement the value of a variable by some amount.

```
balance = balance + deposit;
balance = balance - withdraw;
```

150

syntactic sugar.

- Java supports special operators for these:

```
balance += deposit; // balance = balance + deposit
balance -= withdraw;
```

balance *= deposit
balance /= deposit

- Java supports operators for incrementing or decrementing by 1:

```
i ++; j --;
```

① i ++

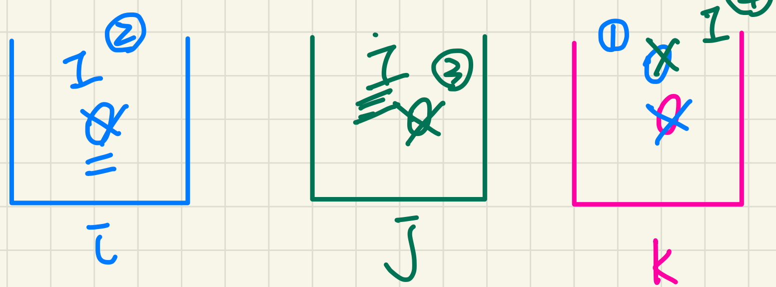
② i = i + 1

③ i += 1

i ** X

Exercise: Preceeding vs Following ++

```
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 */
```



k = i++;

↳ ① use i's value for assignment to k

→ ② perform ++

k = ++j;

→ ③ perform ++ assgn. to k.

→ ④ use j's (new) value for

parse

Escape Sequence

①?

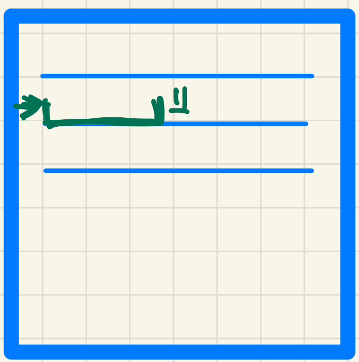
ambiguity

· ① 2nd ' denotes end of char literal

· ② 2nd ' denotes "content" of char literal

`'''`	[INVALID; need to escape ']
`'\''`	[VALID]
`'"'`	[VALID; no need to escape "]
`"""`	[INVALID; need to escape "]
`"\""`	[VALID]
`"'"`	[VALID; no need to escape ']
`"\n\t\""`	[VALID]

valid



Lecture 1

Part G

Elementary Programming - Sources for Variable Assignments

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

console apps

```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 = 3.14169; /* a named constant for π */
    double area = PI * radius * radius; /* area = πr^2 */
    /* Display result */
    System.out.println(
      "Area for circle of radius " + radius + " is " + area)
  }
}
```

With~~out~~ User Input

model.

```
double getArea(double r){
}
```

Example: Convert Seconds to Minutes

Test: 500 seconds

```
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(); 500
    int minutes = seconds / 60; /* minutes */   quotient [6]
    int remainingSeconds = seconds % 60; /* seconds */
    System.out.print(seconds + " seconds is "); [20]
    System.out.print(" minutes and ");
    System.out.println(remainingSeconds + " seconds");
  }
}
```

int : 500 — int

Exercise: Modify the program so that it will display hours if necessary.

e.g., 7945 seconds -> 2 hours, ~~X~~ 12 minutes, ~~Y~~ 25 seconds

Where Can An 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;
```

(i / j) * (i % j)

int

type of expression must match the declared type of assignment target

- An input from the user

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

int

Comparison of Values

int
double
char
boolean

primitive type

use ==

'a'
c1

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

String

reference type

s1 == "quit" X
↳ - compile
- won't crash
- won't work ↳ logical error.

use equals

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

Printing to Console

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

print(s1)
println(s2)

AB.

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

AB

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

A
B

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

A
B

Lecture 2

Part A

Selections -
Motivation of Conditionals

Why Selective Actions

Test Inputs:

radius = 3

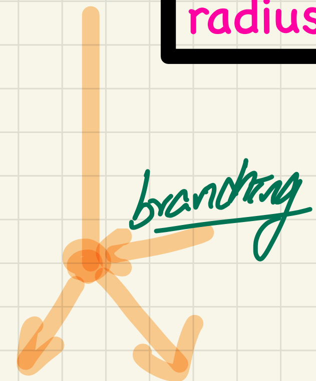
radius = -3

```java
import java.util.Scanner;
public class ComputeArea {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.println("Enter the radius of a circle:");
    double radiusFromUser = input.nextDouble();
    final double PI = 3.14;
    double area = radiusFromUser * radiusFromUser * PI;
    System.out.print("Circle with radius " + radiusFromUser);
    System.out.println(" has an area of " + area);
    input.close();
  }
}
```

3 -3

3 -3

executed despite that input radius < 0.



in this case, an alternative block of code should be executed.

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
```

Lecture 2

Part B

Selections -
Boolean Data Type

Not Equal To

```
int x = 3;

int y = 4;

int z = 4;
```

Relations between 2 numbers

$x < y$

$x = y$

$x > y$

$x <= y$ ≡ $x < y$ or $x == y$ ≡ not the case $x > y$ ≡ $!(x > y)$

false

$! \equiv$ not

$x == y$ ≡ not $!(x != y)$

x != y	≡	!(x == y)
y != z		!(y == z)

true

false not the case

⟹ true is the case

Lecture 2

Part C

Selections -
If-Statement: Syntax and Semantics

① Smallest if-statement

```
if ( ________ ) {
    [            ]
}
```

boolean expression

body of if-statement for a particular branch

③

```
if ( ... ) { — }
else { — }
```

default action

④ longest

```
if ( ~ ) { ... }
else if ( ~ ) { ... }
else if ( ~ ) { ... }
else { ... }
```

✓ ② larger if-statement

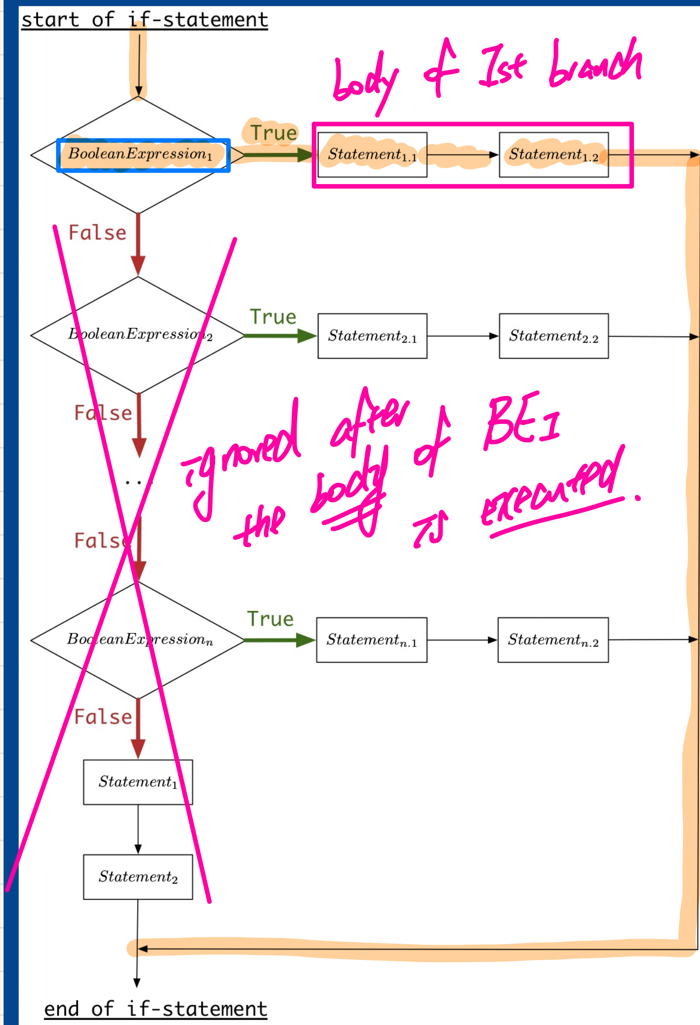
```
if ( ~ ) { ... }
else if ( ~ ) { ... }
else if ( ~ ) { ... }
```

A Single If-Statement

Semantics/ Meaning

Syntax

```
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 are
  Statement1; Statement2;
}
```



body of 1st branch

ignored after the body of BE1 is executed.

Case 1

BooleanExpression1 evaluares to **true**

If-Statement Case 1: Example

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");
}
```

-4

-4 < 0 . T

ignored/bypassed.

Console

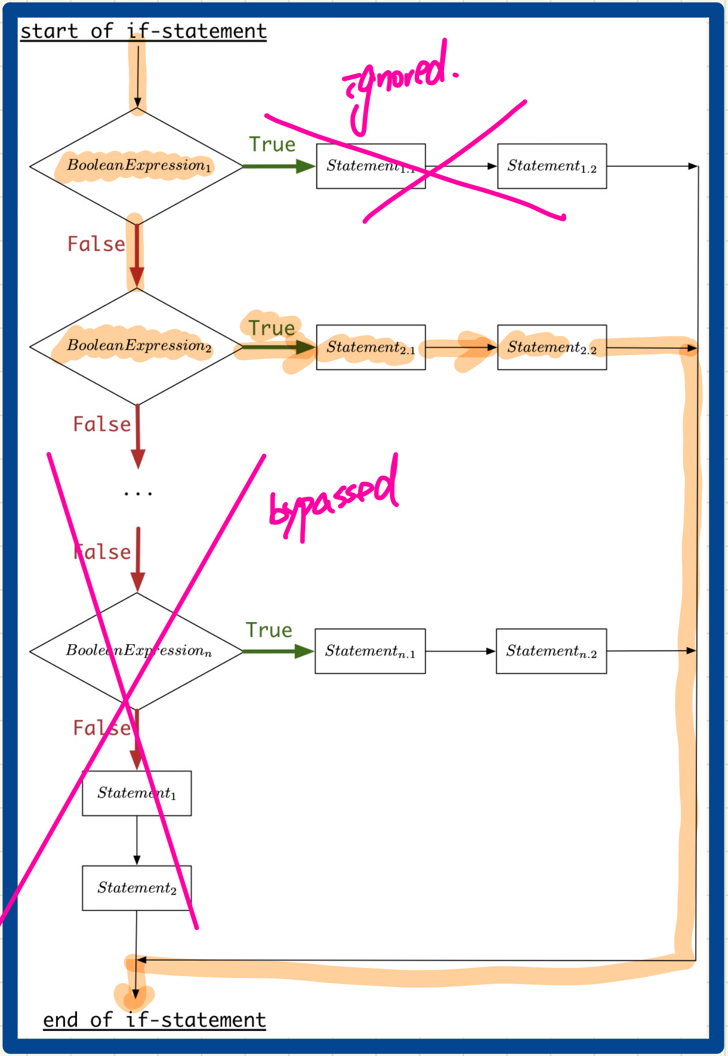
i is negative

A Single If-Statement

Semantics/Meaning

Syntax

```
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 are
  Statement1; Statement2;
}
```



Case 2

BooleanExpression1 evaluares to **false**

BooleanExpression2 evaluares to **true**

If-Statement Case 2: Example

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 10");
}
else if(i == 10) {
  System.out.println("i is equal to 10");
}
else {
  System.out.println("i is greater than 10");
}
```

5<0 F

5<10 T.

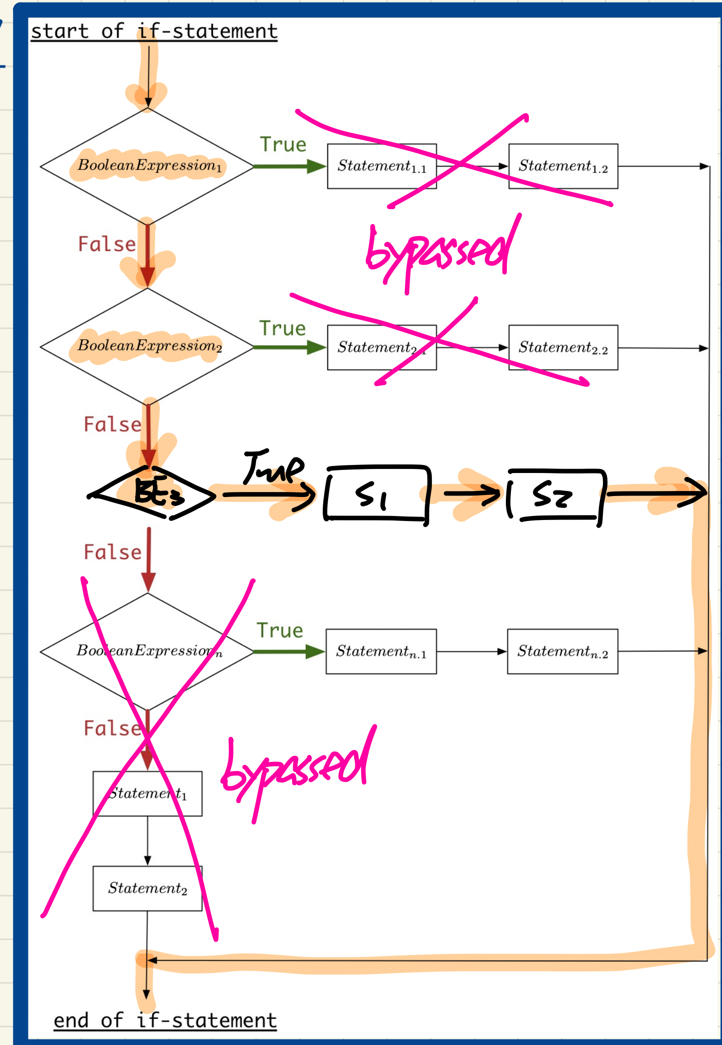
bypassed.

Console

i is less than 10

A Single If-Statement

Semantics/ Meaning



Syntax

```
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 are
  Statement1; Statement2;
}
```

Case 3

$BooleanExpression_1$ evaluares to **false**

$BooleanExpression_2$ evaluares to **false**

$BooleanExpression_3$ evaluares to **true**

If-Statement Case 3: Example

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

```
int i = 10;
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");
}
```

10<0 F

10<10 F

10==10 T

bypassed.

Exercise: Run debugger on Eclipse for Case 3.

Console

I is equal to 10

A Single If-Statement

Semantics/ Meaning

Syntax

```
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 are
  Statement1; Statement2;
}
```

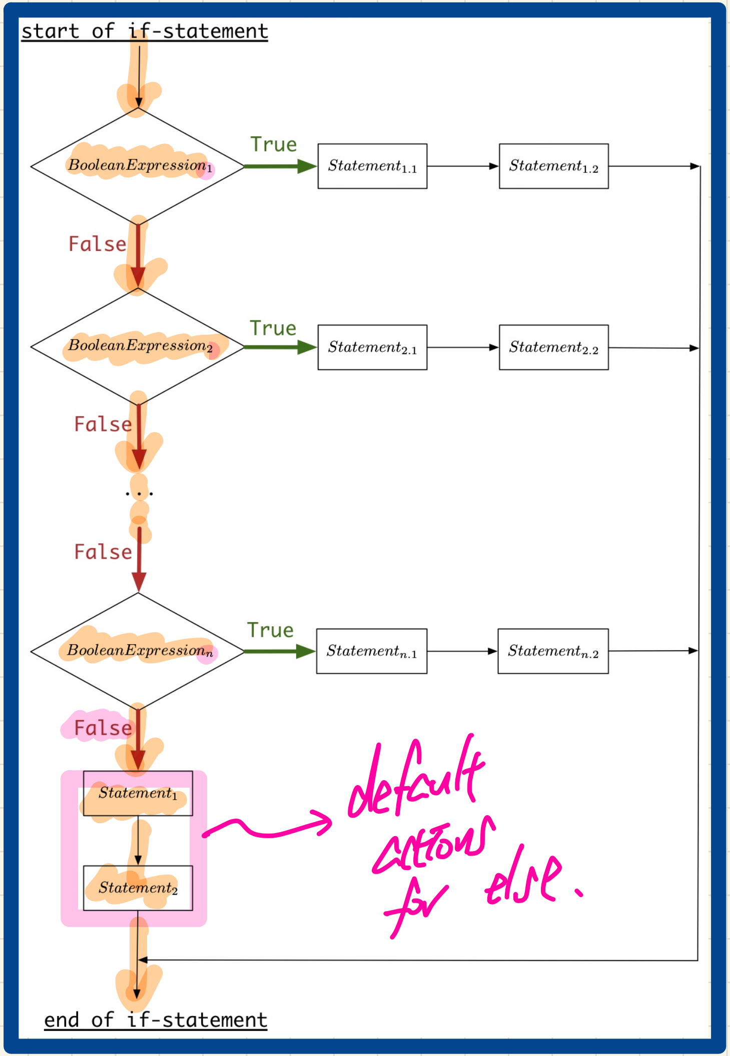
Case 4 An **else** statement is present

*BooleanExpression*1 evaluares to **false**

*BooleanExpression*2 evaluares to **false**

...

*BooleanExpression*n evaluares to **false**



If-Statement Case 4: Example

No satisfying branches, and an `else` part is present, then the *default action* is executed.

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

Console

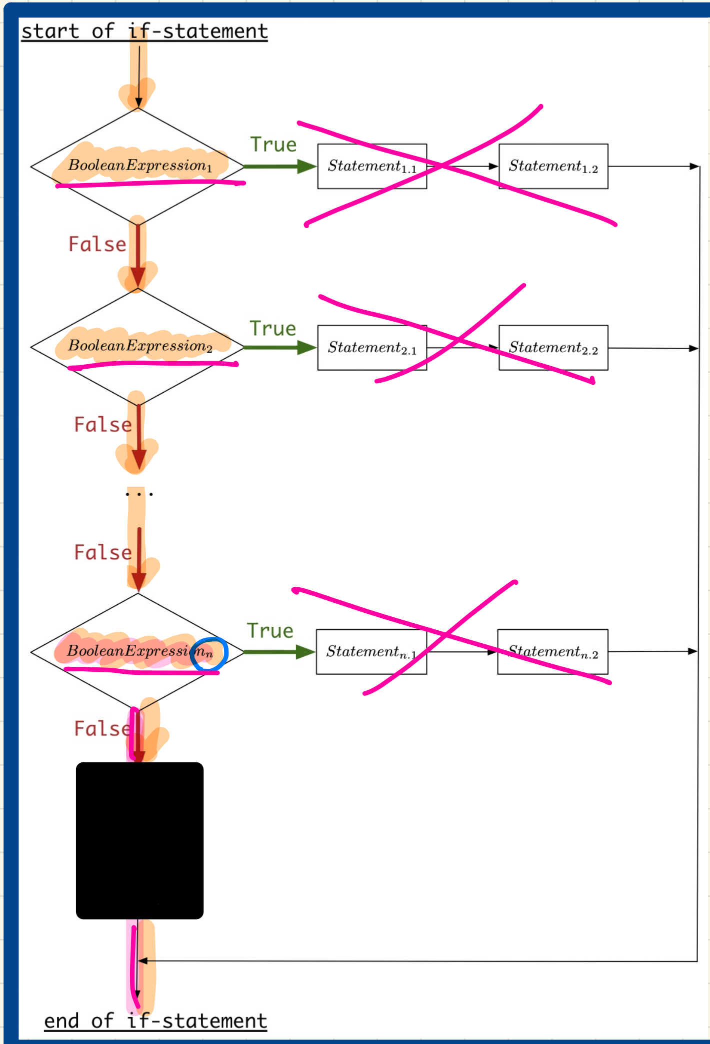
I is greter than 10.

A Single If-Statement

Semantics/ Meaning

Syntax

```
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 are
  Statement1; Statement2;
}
```



Case 5 An **else** statement is **absent**

$BooleanExpression_1$ evaluares to **false**

$BooleanExpression_2$ evaluares to **false**

...

$BooleanExpression_n$ evaluares to **false**

If-Statement Case 5: Example

No satisfying branches, and an `else` part is absent, 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");
}
```

12 < 0 F

12 < 10 F

12 == 10 F.

Console

Lecture 2

Part D

Selections -
Logical Operators

Defining Logical Operators: Truth Tables

¬ ∧ ∨

Negation (¬, not) Conjunction (∧, and) Disjunction (∨, or)

P	! P
false	
true	

P	Q	P && Q
false	false	
false	true	
true	false	
true	true	

P	Q	P || Q
false	false	
false	true	
true	false	
true	true	

L logical OP R

T, F T, F

Example of Logical Operation: Negation

!(radius > 0)
radius <= 0

0

radius > 0

Exercise:
Run in Debugger.

Test Inputs:

radius = 5
radius = 0
radius = -3

The result is the "negated" value of its operand.

Operand op	! op
true	false
false	true

!F → T

data type

isPositive.

! isPositive → false

!(! isPositive) → True

```
double radius = input.nextDouble();
final double PI = 3.14;
boolean isPositive = radius > 0;
if (!isPositive) { /* not the case that isPositive is true */
  System.out.println("Error: radius value must be positive.");
}
else { /* ! isPositive is false   isPositive is True */
  System.out.println("Area is " + radius * radius * PI);
}
```

5 = 0 = -3

T . F F

relational expression → evaluates to a Boolean value (T, F)

F T T

Example of Logical Operation: Conjunction

Test Inputs:

age = 30

age = 50

age = 70

If one of the operands is *false*, their conjunction is *false*.

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

```java
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: Conjunction

Test Inputs:

age = 30

age = 50

age = 70

Exercise: Try 30, 70 on Debugger.

If one of the operands is *false*, their conjunction is *false*.

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

!F T   !T F   70

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

T && T T

Example of Logical Operation: Disjunction

isSenior
age >= 65

isChild
age < 18

Test Inputs:

age = 70

age = 15

age = 40

If one of the operands is *true*, their disjunction is *true*.

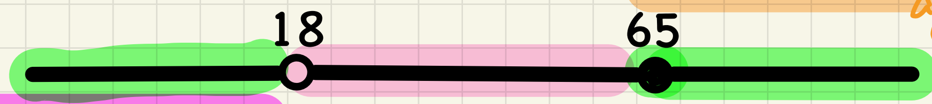
Left Operand `op1`	Right Operand `op2`	`op1 || op2`
false	*false*	*false*
true	*false*	*true*
false	*true*	*true*
true	*true*	*true*

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

Example of Logical Operation: Disjunction

isSenior
age >= 65

isChild
age < 18



Test Inputs:

- age = 70
- age = 15
- age = 40

If one of the operands is *true*, their disjunction is *true*.

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

Exercise: Try all values in Debugger.

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

F || F F

T || F T

F || T T

<u>Lecture 2</u>

<u>Part E</u>

Selections -
Laws of Logical Operators,
Precedence of Logical Operators

Logical Law: Negation of Relational 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 Inputs:

i = 17, j = 3

i = -4, j = 13

$17 \leq 3$

$!(i <= j) \equiv i > j$

$!(i > j) \equiv i <= j$

$i > j$

T

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

equivalent to

F

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

$i <= j$

$i > j$

$i <= j$

$i > j$

Two-Way If-Stmt: Handling Errors

Test Inputs:

radius = 9

radius = -5

Trace on both sides

```
public class ComputeArea {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.println("Enter a radius value:");
    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);
    }
    input.close();
  }
}
```

-5 >= 0 (F)

```
public class ComputeArea2 {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.println("Enter a radius value:");
    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!");
    }
    input.close();
  }
}
```

!(radius < 0)

≡ radius >= 0

!(radius >= 0)

||

radius < 0

Logical Laws: DeMorgan

T

B_1	B_2	$!(B_1 \&\& B_2)$	$!B_1 \mid\mid !B_2$
true	true	false	false
true	false	true	true
false	true	true	true
false	false	true	true

DeMorgan for Conjunction

!T F

B_1	B_2	$!(B_1 \mid\mid B_2)$	$!B_1 \&\& !B_2$
true	true	false	false
true	false	false	false
false	true	false	false
false	false	true	true

DeMorgan for Disjunction

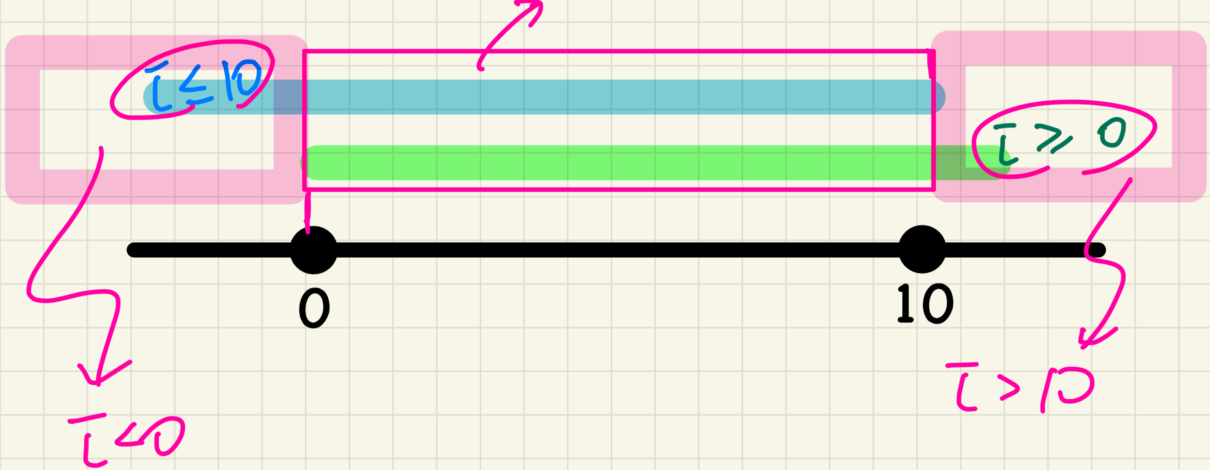
DeMorgan Law of Conjunction: Example (1)

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

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

$!(0 <= i \ \&\& \ i <= 10) \equiv \ !(0 <= i) \ || \ !(i <= 10) \equiv \boxed{0 > i \ || \ i > 10}$

$i \leq 10 \ \&\& \ i \geq 0$



DeMorgan Law of Conjunction: Example (2)

F.

never executed

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

T

always executed.

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

$$!(i < 0 \;\&\&\; false)$$

$$\equiv \;\; !(i<0) \;||\; !(false)$$

$$\equiv \;\; i \geq 0 \;||\; T$$

$$\equiv \;\; T.$$

$$i < 0 \;\&\&\; false$$

$$\leadsto F.$$

DeMorgan Law of Conjunction: Example (3)

never executed.

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

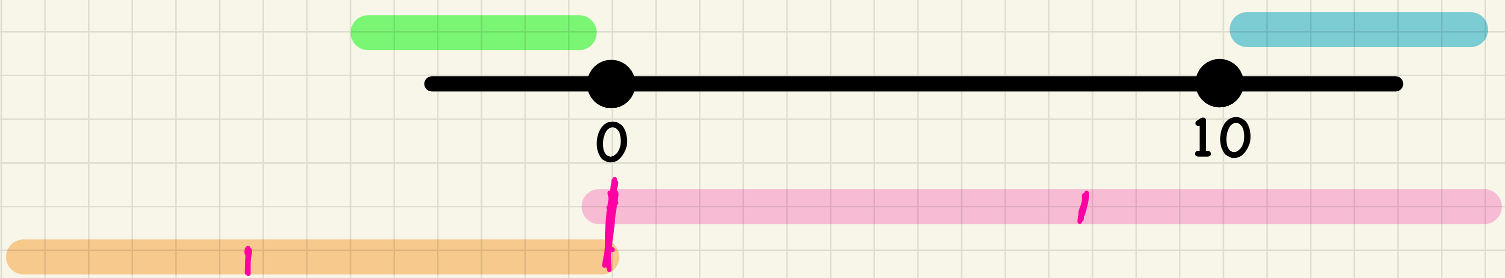
always executed.

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

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

$$\equiv \ !(i<0) \ || \ !(i>10) \ \equiv \ \boxed{i>=0 \ || \ i<=0}$$

T.



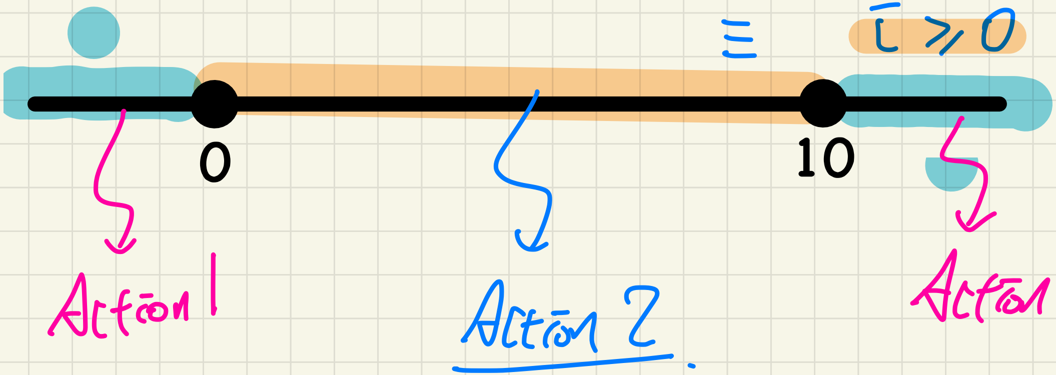
DeMorgan Law of Disjunction: Example (1)

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

- **When is** *Action 2* **executed?** `0 <= i && i <= 10`

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

$$\equiv i \geq 0 \;\&\&\; i \leq 10$$



DeMorgan Law of Disjunction: Example (2)

T — always executed

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

F — never executed

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

$!(i < 0 \mid\mid true)$

DeMorgan

?

Exercise

F

$\boxed{i < 0 \mid\mid true}$ T

DeMorgan Law of Disjunction: Example (3)

T

always executed

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

never executed

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

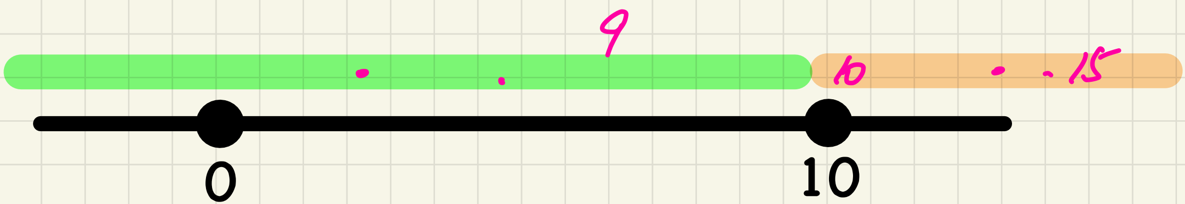
DeMorgan.

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

||| Exercise

?

F



Precedence of Logical Operators

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

!
&&
||

&& has higher precedence than ||

evaluated first

p || (q && r)

T || (T && F)

T

F

T

(p || q) && r

(T || T) && F

T

F

p || q && r

p || q && r

③ !(p || q) && r)

① = ② ≠ ③

Exercise

Find p, q, r showing ② and ③ may evaluate to different results.

① !p || q && r = ② (!p) || (q && r)

Lecture 2

Part F

Selections -
Two-Way vs. Multi-Ways If-Statements,
Nested If-Statements

Two-Way If-Statement without else Part

-23 10 T F

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

Test Inputs:

radius = 10

radius = -23

Console

Area for circle is . –

Console

-23 10 T

```
if (radius >= 0) {
  area = radius * radius * PI;
  System.out.println("Area for the circle of is " + area);
}
else {
  /* Do nothing. */
}
```

→ !(radius >= 0).

Console

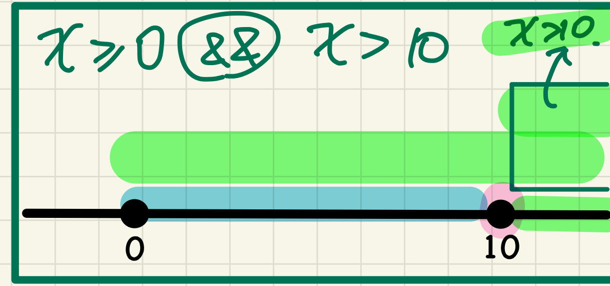
Area for circle is - -

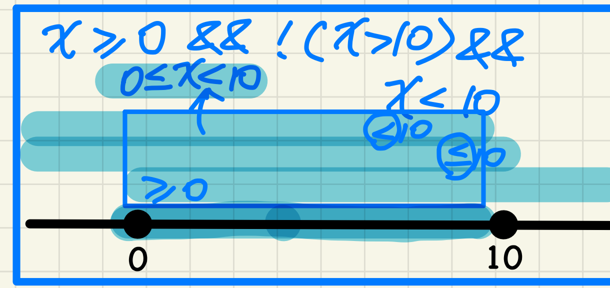
Console

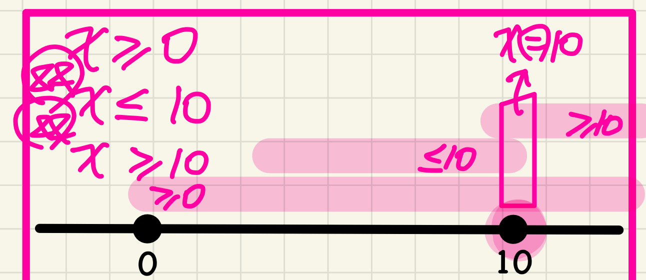
Compound If-Statement: Implicit Conditions

```java
int x = input.nextInt();
int y = 0;
if (x >= 0) {
  System.out.println("x is positive");
  if (x > 10) { y = x * 2; }
  else if (x < 10) { y = x % 2; }
  else { y = x * x; }
}
else { /* x < 0 */
  System.out.println("x is negative");
  if(x < -5) { y = -x; }
}
```

single if-statement







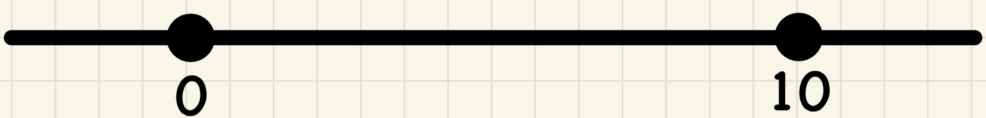
Compound If-Statement: Tracing

```
int x = input.nextInt();
int y = 0;
if (x >= 0) {
  System.out.println("x is positive");
  if (x > 10) { y = x * 2; }
  else if (x < 10) { y = x % 2; }
  else { y = x * x; }
}
else { /* x < 0 */
  System.out.println("x is negative");
  if(x < -5) { y = -x; }
}
```

Test Inputs:

- x = 5
- x = 10
- x = -2

Exercise:
Trace on paper and Debugger.



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");
}
```

≡

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

Test Inputs:

score = 83

score = 71

score = 59

71 F

T

x

71 T

Multi-Way If-Statement without else Part

~~score~~ letterGrade | ~~score~~ letterGrade | ~~score~~ letterGrade "F"

"F"

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

F F F

≡

```java
String letterGrade = "F";
if (score >= 80.0) {
  letterGrade = "A";
}
else {
  if (score >= 70.0) {
    letterGrade = "B";
  }
  else {
    if (score >= 60.0) {
      letterGrade = "C";
    }
    else {
      /* do nothing */
    }
  }
}
```

F F

Test Inputs:

score = 83

score = 71

score = 59

Lecture 2

Part G

Selections -
Overlapping vs. Disjoint Conditions,
Single If-Stmt vs. Multiple If-Stmts

disjoint.

Overlapping vs. Non-Overlapping Intervals

$i >= 3$

$i <= 8$

overlapping

$i <= 3$

$i >= 8$

non-overlapping.

Single If-Stmt vs. Multiple If-Stmts: Overlapping Conditions

- Single vs. multiple
- overlapping

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

Console

i is >= 3

independent if-stmts.

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

Console

i is >= 3
i is <= 8

Single If-Stmt vs. Multiple If-Stmts: Non-Overlapping Conditions

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

Console

i is <= 3

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

Console

i is <= 3

Common Error: Multiple If-Statements with Overlapping Conditions

84

```
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");
}
```

T T T

3 if-statements.

incorrect

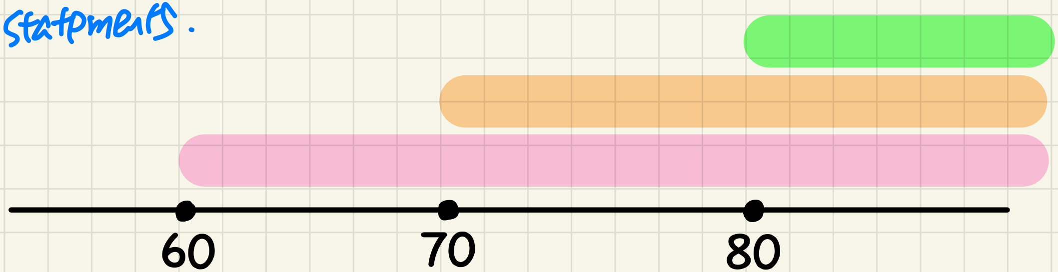
A
B
C

```
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");
}
```

correct

A

single if-statement.



Test Inputs:

marks = 84

Overlapping Conditions: Exercise (1)

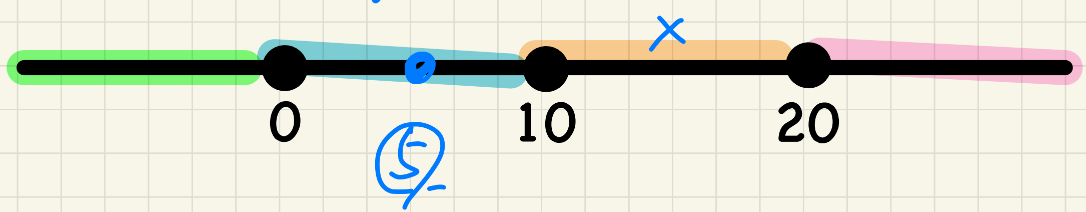
Does this program always print exactly one line?

```
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"); }
```

disjoint.

no value can satisfy more than one of them

⇒ only one if-stmt's body of code is executed.



Overlapping Conditions: Exercises (2, 3)

≥ 10

0

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 statement ⇒ exactly one branch is executed

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"); }
```

$!(x<0)$ && $x < 10$

$\equiv x \geq 0$ && $x < 10$

$!(x<0)$ && $!(x<10)$ && $x<20$

$\equiv \boxed{x \geq 0 \text{ \&\& } x \geq 10}$ && $x<20$

$x \geq 10$

Lecture 2

Part H

Selections -
Scope of Variables

Scope of Variables: Method

```java
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);
}
```

Scope of Variables: Branches

```java
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: Use of Variables from Other 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 k = i * -3; /* a new variable also called j */
    if (j < k) { ... }      ×
  }
}
```

Scope of Variables: Use of Variables Outside If-Stmt

```
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) { ... }
  }
  System.out.println("i * j is " + (i * j));   ×
}
```

outside scopes of [ ] and [ ]

Scope of Variables: Method Parameters & Return Values

```
public class SumApp {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    int i = input.nextInt();
    int j = input.nextInt();
    int k = Utilities.getSum(i, j);
    System.out.println(k);
} }
```

conceptually:

int k = result;

what Java run time does

```
public class Utilities {
  public static int getSum(int x, int y) {
    int result = x + y;
    return result;
} }
```

i j

but you can not write this

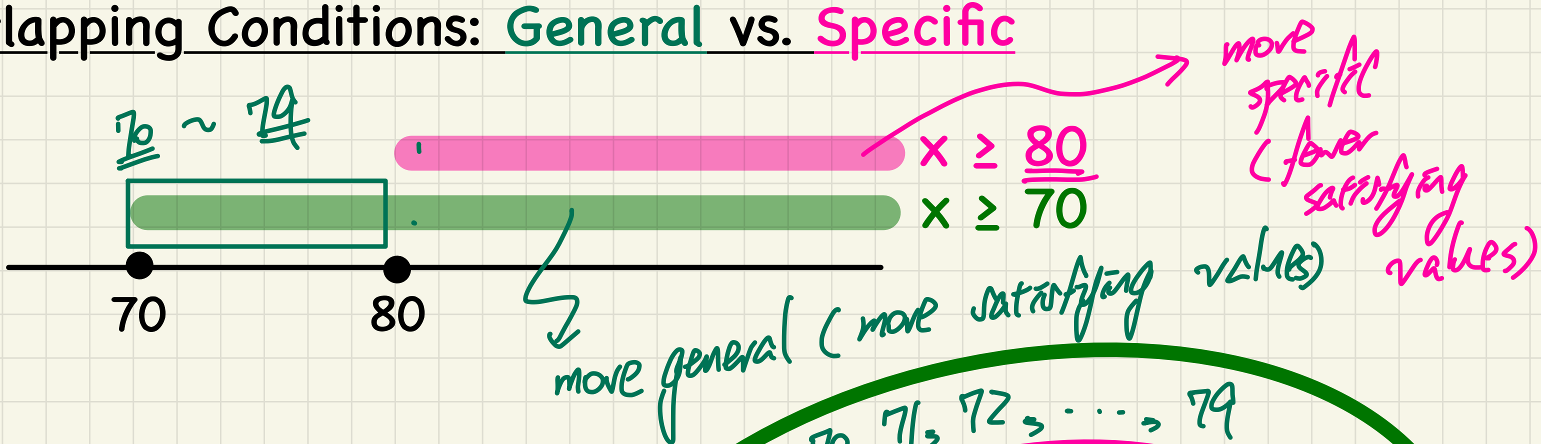
i j

<u>Lecture 2</u>

<u>Part I</u>

Selections -
Single If-Stmts
Conditions: General vs Specific

Overlapping Conditions: General vs. Specific

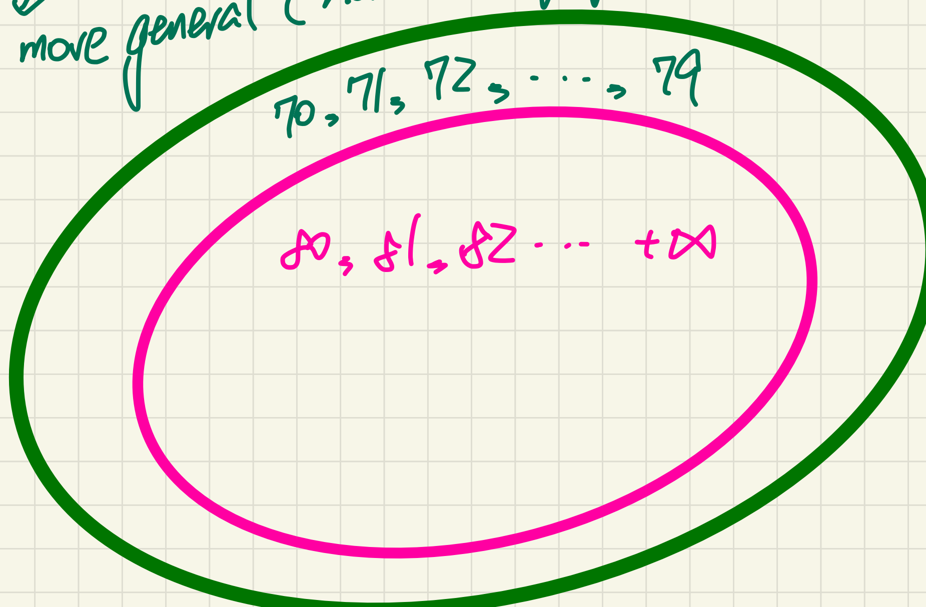


x ≥ 70 is more general

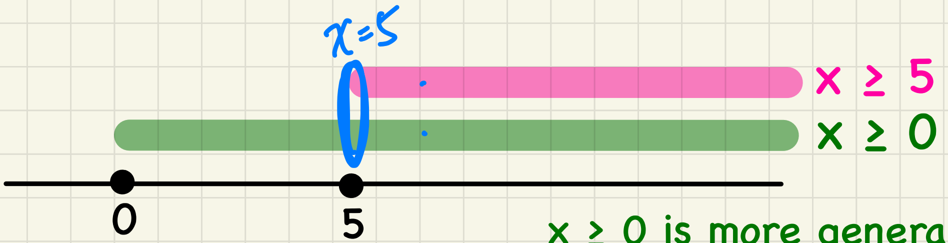
x ≥ 80 is more specific

Boolean condition

↳ set of satisfying values

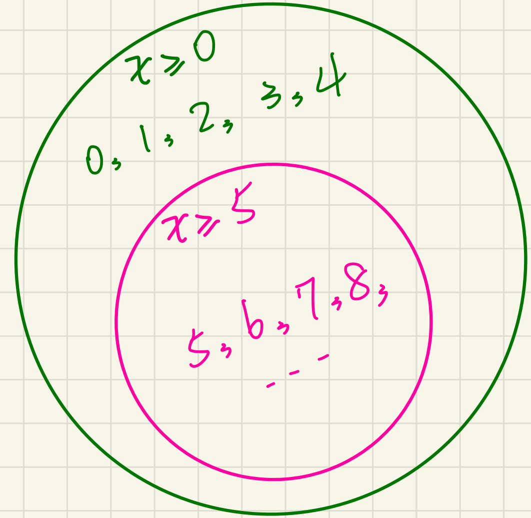


Overlapping Conditions in a Single If-Statement



x ≥ 0 is more general

x ≥ 5 is more specific



Test Inputs:

x = 5

more specific

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

v1

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

T

Ex

x>=5

is different from having this order

more general.

v2

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

T

Ex

x>=0

Single If-Stmt with General to Specific Branching Conditions

```
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" ;
}
```

T

Pass

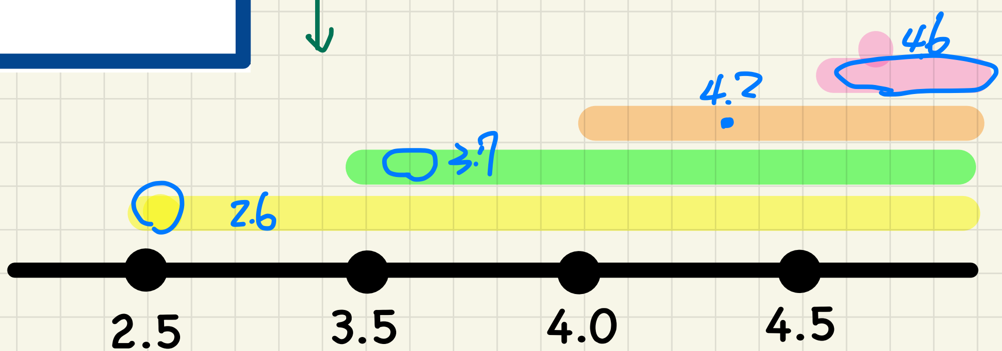
correct but inaccurate.

single if-stmt.

branching conditions sorted from most general to most specific

Test Inputs:

gpa = 4.8



Lecture 2

Part J

Selections -
Short-Circuit Effect of && and ||

b1 && b2

F means no need to evaluate b2.

T.   T   T

        F   F

↳ as long as one operand is false, result is F

b1 || b2

T means no need to evaluate b2.

F.   T : T

        F : F

↳ as long as one operand is true, result is T.

Short-Circuit Evaluation: &&

Q.* y/x >2 && x != 0

10/0 Crash!

SCE would not help.

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

Test Inputs:

x = 0, y = 10

x = 5, y = 10

```
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");
  }
}
```

(guarding construct)

protects the division y/x

y/x > 2

y/x ≤ 2

F

0 != 0 && 10/0 > 2

F

unnecessary to evaluate.

5 != 0 && 10/5 > 2

T F

Q.* y/x>2 || x==0

Exercise Justify this version using || is equivalent to the previous version using &&.

Short-Circuit Evaluation: ||

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

Test Inputs:

x = 0 y = 10

x = 5 y = 10

```
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");
}
```

0

10

guarding constraint

y/x >2

y/x ≤2

0 == 0 || 10/0 > 2

T

not necessary to evaluate

F.

5 == 0 || 10/5 > 2

F F

Short-Circuit Evaluation: Common Errors

Test Inputs:
x = 0 y = 10

division to protect/guard.

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

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

crash.

meant to be guarding constraint.

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

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

10/0 ⤳ crash.

Lecture 2

Part K

Selections -
More Common Errors and Pitfalls

Common Errors: 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.

Fix

```
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); }
```

-3

T

0.

Fix

interpretation by Java compiler

if { } were missing.

Test Inputs:

radius = -3

Common Errors: Misplaced Semicolon

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

not part of the if-stmt.

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

F

-4 * -4 * π

Test Inputs:

radius = -4

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

≡

```
if (radius >= 0) {
  // do nothing.
}
```

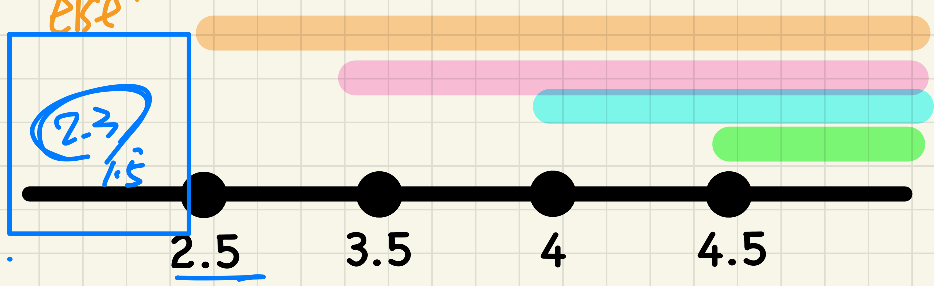
Common Errors: Variable Not Properly Re-Assigned

```
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 Inputs:

gpa = 1.5

single if-stmt without an "else"



Common Errors: Ambiguous "else"

"dangling" else.

v1

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

Test Inputs:

x = 20

v2

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

Test Inputs:

x = 20

x is negative.

Common Pitfall: Simplifiable Boolean Expressions

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

boolean isEven = number % 2 == 0;

isEven	isE == False	!isEven
T	T==F F	F
F	F==F T	T

!isEven

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

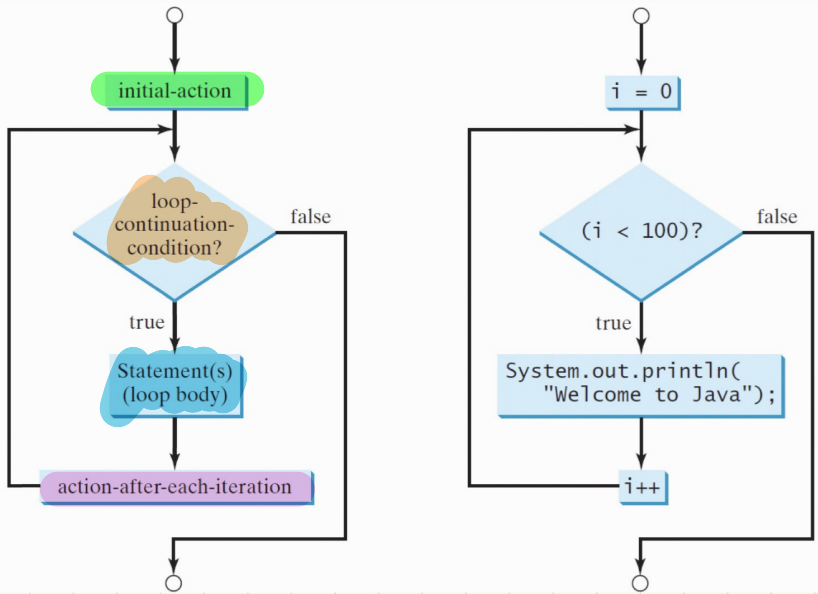
Lecture 3

Part A

Loops -
for-Loop vs. while-Loop
Syntax and Semantics

for-Loop: Syntax and Semantics

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



Q. How many times is the stsy condition (i < 100) checked?

Q. How many times is the loop body (println) executed?

[1, 3) ⇝ 1, 2 [1, 3] 1, 2, 3

for-Loop: Tracing

i < 100

(99 − 0) + 1 ⇝ 100

0 : 99

T 100 F

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

inclusive [n, m] size? m − n + 1

lower upper

[23, 24] ⇝ 24 − 23 + 1 ⇝ 2.

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*	–	–

100 ⇝ iterations

⇝ 1

⇝ no infinite loop.

Q. How many times is the stsy condition (i < 100) checked? 101

Q. How many times is the loop body (println) executed? 100

for-Loop: Alternative Syntax

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

println(i); ✗

- 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++;
}
println(i); ✓
```

for-Loop: Exercises (1)

v1

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

x 100

v2

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

x 100

Q. Are the outputs same or different?

count = 2i − 1

count	count < 100	Iteration
0	T	1
1	T	2
⋮		
99	T	100
100	F	

[0, 99]

100

count	count < 201	Iteration (i)
1	T	1
3	T	2
5	T	3
7	T	4
⋮		
199	T	100
201	F	

199 = 2i − 1

i = 100

for-Loop: Exercises (2)

[0, 99] → 100

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

0

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

[1, 100] → 100

1

Q. Are the outputs same or different?

for-Loop: Exercises (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

2 3 4 5 6

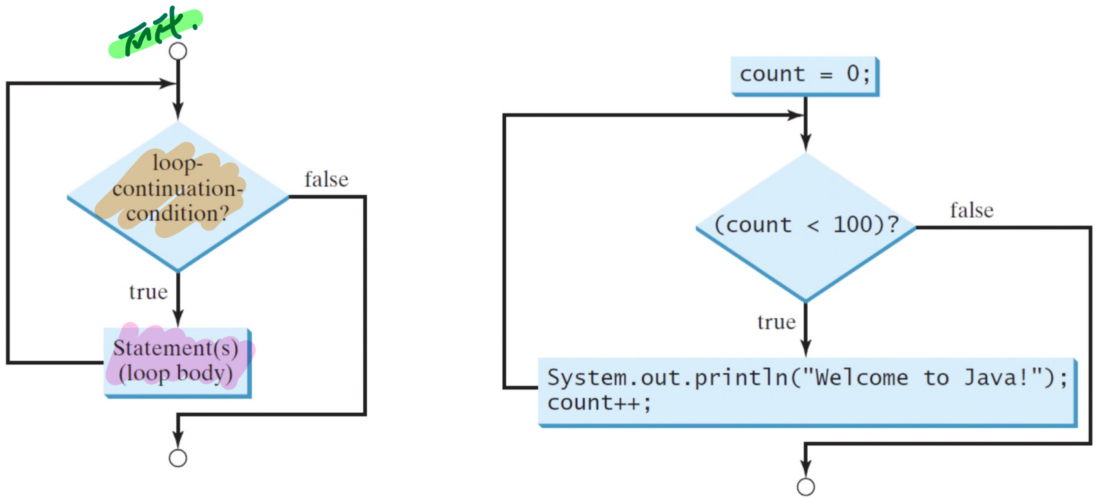
```
int i = 1;
for ( ; i <= 5 ; ) {
  i ++;
  System.out.print(i); }
```

i	i<=5	i+	i++
1	T	1	2
2	T	2	3
3	T	3	4
4	T	4	5
5	T	5	6
	F		

Output: 23456

while-Loop: Syntax and Semantics

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



Q. How many times is the stsy condition (i < 100) checked?

Q. How many times is the loop body (println) executed?

while-Loop: Tracing

$j = i + 2$

$102 = i + 2 \Rightarrow i = 100$

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

j	$j < 103$	Enter/Stay Loop?	Iteration	Actions
3	$3 < 103$	*True*	1 i	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*	–	–

Q. How many times is the stsy condition (i < 100) checked? 101

Q. How many times is the loop body (println) executed? 100

while-Loop: Exercises (1)

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

[0, 99] → 100

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

[1, 100] → 100

Q. Are the outputs same or different?

count	count < 100	Iteration

count	count <= 100	Iteration

while-Loop: Exercises (2)

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

[0, 99]

0

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

[1, 100]

1

Q. Are the outputs same or different?

<u>**Lecture 3**</u>

<u>**Part B**</u>

Loops -
Compound Loops,
for-Loops vs. and while-Loops

Compound Loop: Exercises (1)

```
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 Inputs:

radius = -3

Test Inputs:

radius = 2

radius = -3

Test Inputs:

radius = 2

radius = 3

reaching this line, we already exit from loop.

↳ $!(radius >= 0)$

$\equiv$ $radius < 0$

Compound Loop: Exercises (2.1)

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

Test Inputs:

radius = -3

Test Inputs:

radius = 2

radius = -3

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

!T = F

Test Inputs:

radius = 2

radius = 3

Compound Loop: Exercises (2.2)

Test Inputs:

radius = 2

radius = -3

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    isPositive = radius >= 0; }
10 System.out.println("Error: negative radius value.");
```

Console

?

try this on Eclipse

for-Loop vs. 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 */
}
```

where B is any valid Boolean expression.

expressive power equivalent

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 ++;
}
```

where B is any valid Boolean expression.

Lecture 3

Part C

Loops -
Stay Condition vs. Exit Condition

Stay Condition vs. Exit Condition

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

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

↳ repeat Action 1 as long as p && q evaluates true

↳ exit from loop: $!(p \&\& q) \equiv !p \,||\, !q$

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

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

↳ repeat Action 2 as long as p || q evaluates true

↳ exit from loop: $!(p \,||\, q) \equiv !p \,\&\&\, !q$

Stay Condition vs. Exit Condition: Exercise

infinite loop

Consider the following loop:

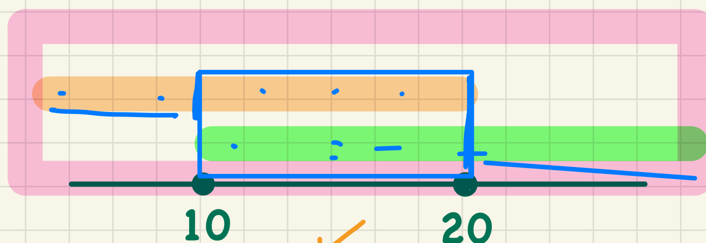
```
int x = input.nextInt();
while(10 <= x || x <= 20) {
  /* body of while loop */
}
```

True

always evaluates to true ⇒ never exit

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

Stay Condition



True

$10 > x$ && $x > 20$ = False

never exit

Exit Condition

$!(10 <= x \ || \ x <= 20)$

Lecture 3

Part D

Loops -
Arrays: Declaration and Initialization

Initializing an Array of Integers (1)

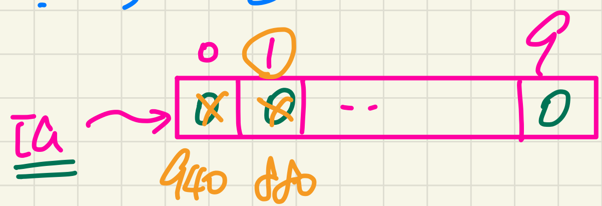
ia

0	1	2	3	4	5	6	7	8	9
940	880	830	790	750	660	650	590	510	440

Approach 1: Initializer

```
int[] ia = {940, 880, 830, ..., 440};
```

Approach 2: Discrete Assignments




```
int[] ia = new int[10];
ia[0] = 940;    ia[1] = 880; ...
```

Initializing an Array of Integers (2)

ia

0	1	2	3	4	5	6	7	8	9
7	10	13	16	19	22	25	28	31	34

10

ArrayIndexOutOfBoundsException

Invalid index

Approach 3: Patternizing Stored Values

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

seq

0	1	2		9
~~0~~	~~0~~	0	...	0
7	10	13		

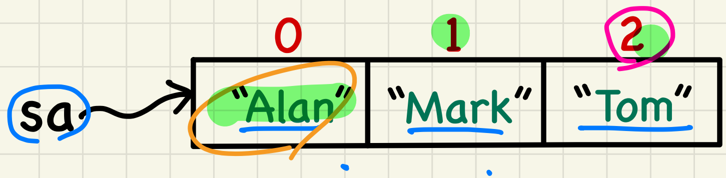
i	i < seq.length	i - 1	seq[i - 1]
0		-1	seq[-1]
1		0	seq[0] 7
2		1	seq[1] 10
3	True.		
4			
5			
6			
7			
8			
9			
10	F		

1st it.

~~10th~~ 7th.

4th.

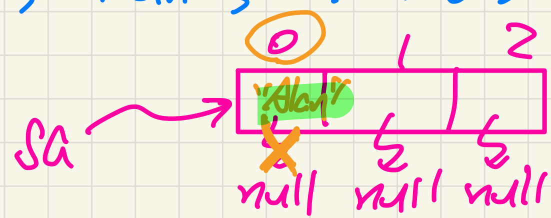
Initializing an Array of Strings



Approach 1: Initializer

```
String[] sa = {"Alan", "Mark", "Tom"};
```

Approach 2: Discrete Assignments




```
String[] sa = new String[3];
sa[0] = "Alan";
```

for-Loops vs. while-Loops: Iterating through Arrays

```java
int[] a = new int[100];
for(int i = 0; i < a.length; i ++) {
  /* Actions to repeat. */
}
```

min index of array

Stay condition

Exit: !(i < a.length)

'''

i >= a.length

first invalid index

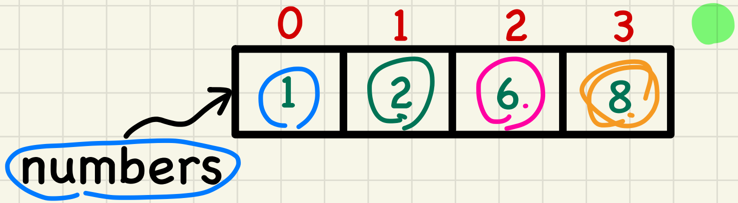
```java
int[] a = new int[100];
int i = 0;
while(i < a.length) {
   /* Actions to repeat. */
   i ++;
}
```

Lecture 3

Part E

Loops and Arrays - Computational Problems

Computational Problem: Average



Test Inputs:

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

4.25

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);
```

0 < 0 F 4

0.0/0 → division by zero exception.

4

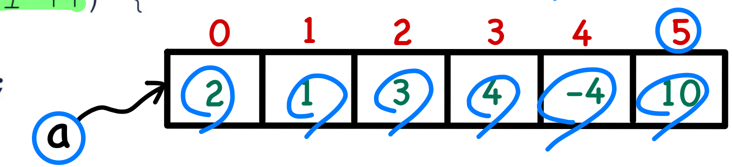
i	sum
0	1
1	3
2	9
3	17
4	

exit.

Computational Problem: Conditional Printing

b

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



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

at the end of the 6th it.

Console

2
1
3
4
10

Computational Problem: Printing Comma-Separated Lists

```java
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(".");
```

Current index not yet reaching the largest index

largest valid index

names.length - 1

names

0	1	2
"Alan"	"Mark"	"Tom"

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

Console

Names: Alan, Mark, Tom.

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]);
}
```

i: 0 1 2 3 4
j: 4 3 2 1 0

$j = \boxed{ns.length - 1 - i}$

$5 - 1 - i$

$4 - i$

infinite loop

$i \geq 0$

names.length − 1

~~i < ns.length~~

i − −

j

$j = 3 - i$

$\boxed{i < 0}$

exit

$\boxed{ns.length - 1}$

i	i < ns.length
3	T
2	T
1	T
0	T
−1	T

numbers: 0 → 1, 1 → 2, 2 → 3, 3 → 4

0 ↔ 3
1 ↔ 2
2 ↔ 1
3 ↔ 0

numbers

Computational Problem: Finding Maximum

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

current element > max so far.

a

0	1	2	3	4	5
2	1	3	4	-4	10

scanned

i	*a*[*i*]	*a*[*i*] > *max*	update *max*?	*max*
0	–	–	–	2
0	2	2>2 *false*	N	2
1	1	1>2 *false*	N	2
2	3	3>2 *true*	Y	3
3	4	4>3 *true*	Y	4
4	-4	-4>4 *false*	N	4
5	10	10>4 *true*	Y	10

6

Console

Max is 10

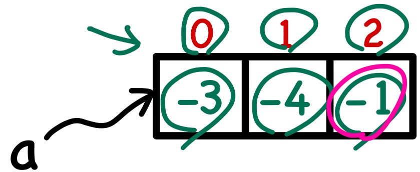
Computational Problem: Finding Maximum

Exercise 1

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

```
1  int max = a[0]; 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);
```

× × ×



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

Console

Max is 0

Computational Problem: Finding Maximum

$i > i \equiv F$

Exercise 2

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);
```

23

1

a

0	1	2
23	-2	46

max

i	i < a.length	a[i]	a[i] > max
~~0~~		~~23~~	~~a[0] > a[0]~~
1	True		
2			
3	F		

⇕ F

1st iteration

⇕

always compare a[0] with itself ⇒ F

Console

Computational Problem: Checking a Universal Property

generalization of &&

> 0

F

0	1	2	3	4
2	3	-1	4	5

ns

witness of violation

boolean *allPositive*

$$
\begin{array}{ll}
 & ns[0] > 0 \\
\&\& & ns[1] > 0 \\
\&\& & ns[2] > 0 \\
\&\& & ns[3] > 0 \\
\&\& & ns[4] > 0
\end{array}
$$

F

F

→ Zero of &&

False && b ≡ False

→ Identity of &&

True && $\boxed{b}$ ≡ b.

Computational Problem: Checking an Existential Property

>0

generalization of ||

	0	1	2	3	4
ns	-2	-3	1	-4	-5

>0

witness of satisfaction

boolean *atLeastOnePositive*

$$
\begin{array}{l}
ns[0] > 0 \\
\|\; ns[1] > 0 \\
\|\; ns[2] > 0 \\
\|\; ns[3] > 0 \\
\|\; ns[4] > 0
\end{array}
$$

T

T

Zero of ||

$$\text{True} \;\|\; b \equiv \text{True}$$

Identity of ||

$$\text{False} \;\|\; b \equiv b$$

Computational Problem: Are All Numbers Positive?

what if F.

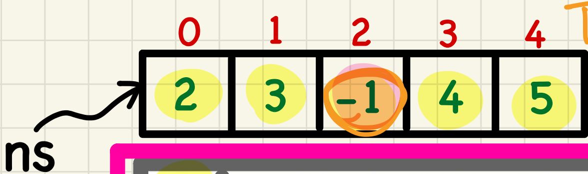
```
int[] ns = {2, 3, -1, 4, 5};
boolean soFarOnlyPosNums = true;
int i = 0;
while (i < ns.length) {
  soFarOnlyPosNums = soFarOnlyPosNums && (ns[i] > 0);
  i = i + 1;
}
```

Identity of &&

Version 1

ns[i] > 0 check is meaningless (zero of &&).

F && ___ ≡ F



i	soFarOnlyPosNums	i < ns.length	stay?	ns[i]	ns[i] > 0
0	true (F)	true	YES	2	true
1	true	true	YES	3	true
2	true	true	YES	-1	false
3	false	true	YES	4	true
4	false	true	YES	5	true
5	false	false	NO	–	–

Computational Problem: At Least One Number Positive?

```java
int[] ns = {-2, -3, 1, -4, -5};
boolean seenSomePosNum = false;
int i = 0;
while (i < ns.length) {
  seenSomePosNum = seenSomePosNum || (ns[i] > 0);
  i = i + 1;
}
```

Identity of ||

Version 1

F || F ≡ F

F || F ≡ F

F || T ≡ T

T || F

||

T ?

True || ___ ≡ True

ns

0	1	2	3	4
-2	-3	1	-4	-5

i	seenSomePosNum	i < ns.length	stay?	ns[i]	ns[i] > 0
0	false	true	YES	-2	false
1	false	true	YES	-3	false
2	false	true	YES	1	true
3	true	true	YES	-4	false
4	true	true	YES	-5	false
5	true	false	NO	–	–

T

Computational Problem: Are All Numbers Positive?

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

Version 2

the final value of corresponds to the last check

ns

0	1	2	3	4
2	3	-1	4	5

witness

expected: univ. property: F

i	soFarOnlyPosNums	i < ns.length	stay?	ns[i]	ns[i] > 0
0	true	true	YES	2	true
1	true	true	YES	3	true
2	true	true	YES	-1	false
3	false	true	YES	4	true
4	true	true	YES	5	true
5	true	false	NO	–	–

X

Computational Problem: At Least One Number Positive?

```
int[] ns = {-2, -3, 1, -4, -5};
boolean seenSomePosNum = false;
int i = 0;
while (i < ns.length) {
  seenSomePosNum = ns[i] > 0; /* wrong */
  i = i + 1;
}
```

Version 2

final result corresponds to ns[4] > 0.

ns

0	1	2	3	4
-2	-3	1	-4	-5

witness ⇒ expected exist. check: T

i	seenSomePosNum	i < ns.length	stay?	ns[i]	ns[i] > 0
0	false	true	YES	-2	false
1	false	true	YES	-3	false
2	false	true	YES	1	true
3	true	true	YES	-4	false
4	false	true	YES	-5	false
5	false	false	NO	–	–

x

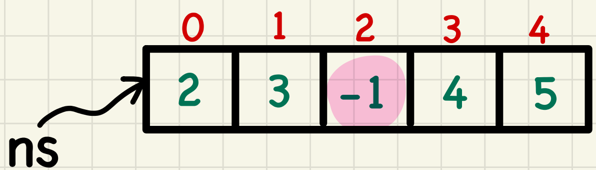
Computational Problem: Are All Numbers Positive?

Version 3

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

✓ F F - exit

unnecessary



exit: !(stopn && i<ns.len.)

T && F ≡ F ≡ !stopn || i >= ns.len

have just seen a number ≤ 0

i	soFarOnlyPosNums	i < ns.length	stay?	ns[i]	ns[i] > 0
0	true	true	YES	2	true
1	true	true	YES	3	true
2	true	true	YES	-1	false
3	false	true	NO	–	–

Computational Problem: At Least One Number Positive?

Version 3

```
int[] ns = {-2, -3, 1, -4, -5};
boolean seenSomePosNum = false;
int i = 0;
while (!seenSomePosNum && i < ns.length) {
  seenSomePosNum = seenSomePosNum || ns[i] > 0;
  i = i + 1;
}
```

!T && 3<5 ≡ F && T ≡ F

F || T ≡ T

ns

0	1	2	3	4
-2	-3	1	-4	-5

~~unnecessary~~

loop already exit.

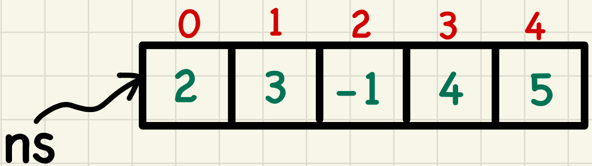
exit: $!(!sspn \ \&\& \ i < ns.len) \equiv sspn \ || \ i >= ns.len$

i	*seenSomePosNum*	*i* < *ns.length*	stay?	*ns*[*i*]	*ns*[*i*] > 0
0	*false*	*true*	YES	-2	*false*
1	*false*	*true*	YES	-3	*false*
2	*false*	*true*	YES	1	*true*
3	*true*	*true*	NO	–	–

Computational Problem: Are All Numbers Positive?

Version 4

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



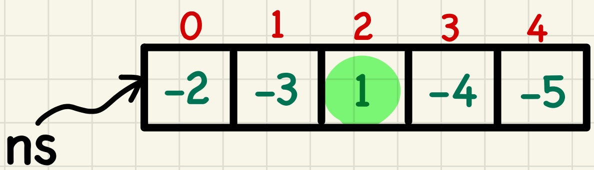
i	*soFarOnlyPosNums*	*i* < *ns.length*	stay?	*ns*[*i*]	*ns*[*i*] > *0*
0	*true*	*true*	YES	2	*true*
1	*true*	*true*	YES	3	*true*
2	*true*	*true*	YES	-1	*false*
3	*false*	*true*	NO	–	–

Computational Problem: At Least One Number Positive?

Version 4

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

!T = F



i	seenSomePosNum	i < ns.length	stay?	ns[i]	ns[i] > 0
0	false	true	YES	-2	false
1	false	true	YES	-3	false
2	false	true	YES	1	true
3	true	true	NO	–	–

Computational Problem: Are All Numbers Positive?

Four possible solutions (`soFarOnlyPosNums` initialized *true*): summary

1. Scan the entire array and accumulate the result.

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

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

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

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

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

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

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

Computational Problem: At Least One Number Positive?

summary

Four possible solutions (`seenSomePosNum` initialized *false*):

1. Scan the entire array and accumulate the result.

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

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

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

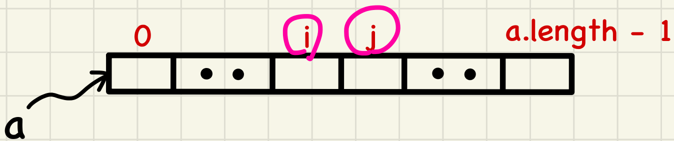
3. The result is accumulative until the early exit point.

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

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

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

Sorting Orders of Arrays



decreasing $a[i] > a[j]$

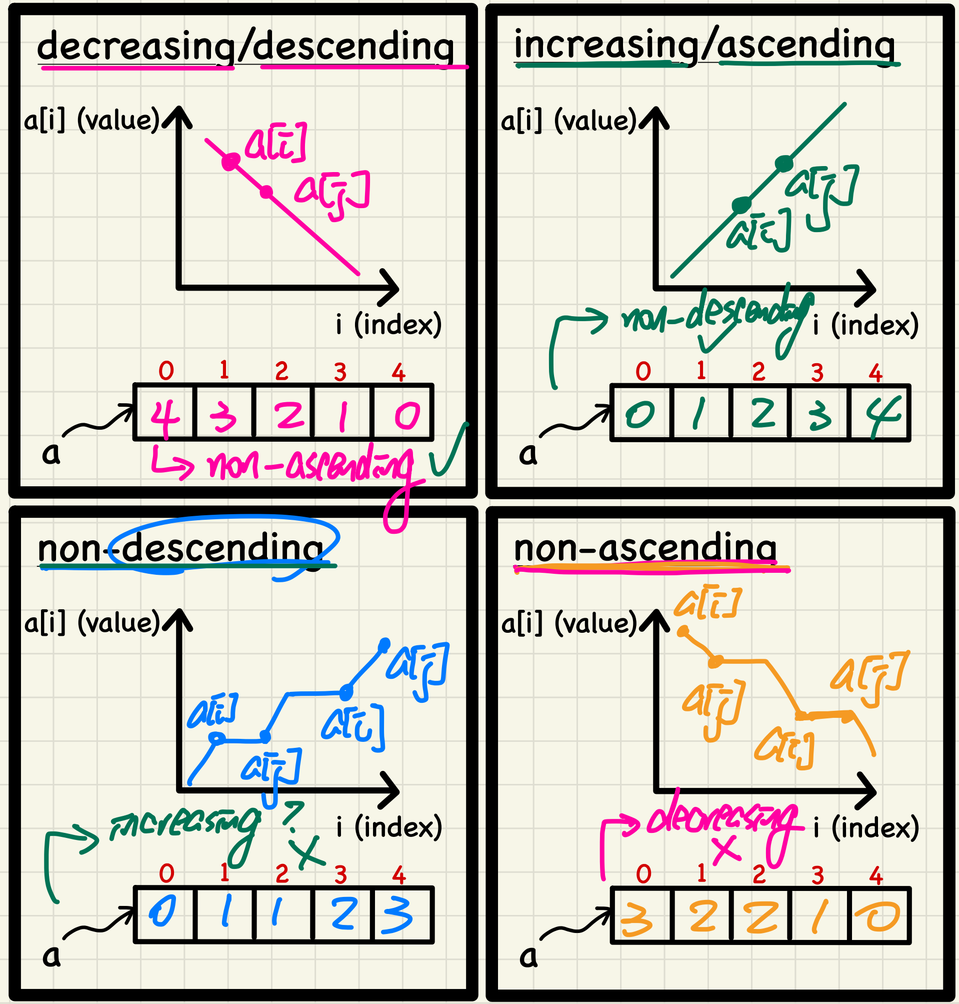
ascending $a[i] < a[j]$

non-descending $!(a[i] > a[j])$

$\equiv a[i] \leq a[j]$

non-ascending $!(a[i] < a[j])$

$\equiv a[i] \geq a[j]$



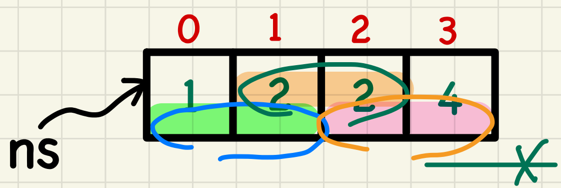
Computational Problem: Is an Array Sorted?

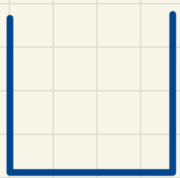
```java
boolean isSorted = true;
for(int i = 0; isSorted && i < a.length - 1; i ++) {
  isSorted = a[i] <= a[i + 1];
}
```

3

Test Case 1

4





isSorted

i	i < a.length	a[i] <= a[i + 1]	
0		$1 \leq 2$	T
1		$2 \leq 2$	T
2		$2 \leq 4$	T
3	T		

Computational Problem: Is an Array Sorted?

```java
boolean isSorted = true;
for(int i = 0; isSorted && i < a.length - 1; i ++) {
  isSorted = a[i] <= a[i + 1];
}
```

F F F

Test Case 2

4

ns

0	1	2	3
2	4	3	3

isSorted

i	i < a.length	a[i] <= a[i + 1]
0		$2 \leq 4$ T
1		$4 \leq 3$ F
~~2~~		

exit

Lecture 3

Part F

Loops and Arrays -
Short Circuit Evaluation and Indexing

Unguarded Array Indexing

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

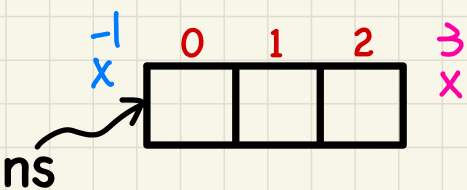
AIOBE.

AIOBE.

Test Inputs:

i = -1

i = 3



resolution to AIOBE

SCE.

&&
||
guard array indexing.

ns[i]

Use of Conjunction (&&)

Guarding Array Indexing using Short Circuit

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

Test Inputs:

$i = -1$

$i = 3$

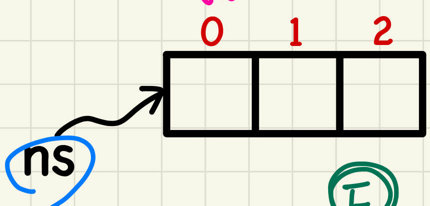
valid

-1 3

indexing to guard

not evaluated.

Exercise.



$0 <= 3$ && $3 < 3$ $\boxed{\&\&\ ns[3] \% == 0.}$

T F

bypassed

will not be evaluated.

bypassed.

F

$0 <= -1$ $\boxed{\&\&\ -1 < 3\ \&\&\ ns[-1] \% 2 == 0}$

Use of Disjunction (||)

Guarding Array Indexing using Short Circuit

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

Test Inputs:

i = -1

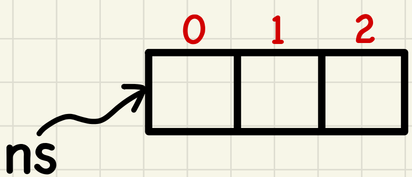
i = 3

invalid

3

to guard.

bypassed



$\underset{F}{\underline{3 < 0}} \;||\; \underset{T}{\underline{3 >= 3}} \;||\; \boxed{ns[3] \% 2 == 1}$

not evaluated.

invalid

bypassed

not evaluated.

$\underset{T}{\underline{-1 < 0}} \;||\; \boxed{-1 >= 3 \;||\; ns[-1] \% 2 == 1}$

Guarding Array Indexing using Short Circuit

```
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 Inputs:

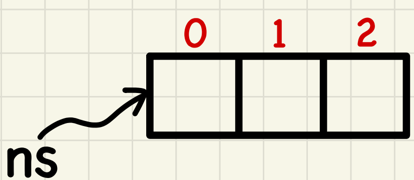
i = -1

i = 3

① work

② crash?

Q. L10: 0 <= i && ns[i] % 2 == 0 && i < ns.length?



Use of Conjunction (&&)

Exercise

Guarding Array Indexing using Short Circuit

```
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."); }
```

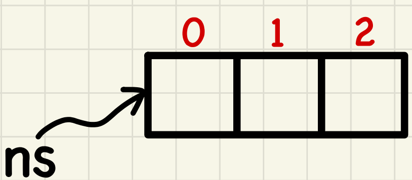
Test Inputs:

i = -1

i = 3

① crash ?
② work ?

Q. L10: i < 0 || ns[i] % 2 == 0 || i >= ns.length?



Use of Disjunction (||) Exercise

Lecture 3

Part G

Loops and Arrays - Common Errors

Common Errors: Improper Initialization of Loop Counter

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

false

X

nothing will be executed

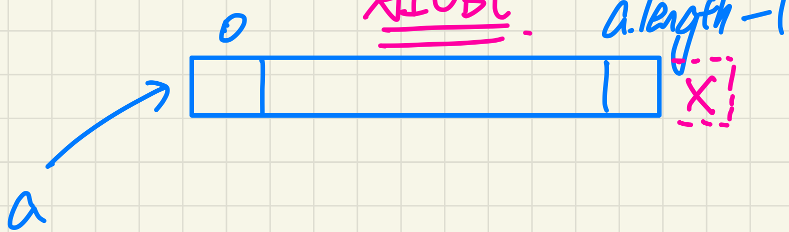
fix: boolean userWantsToContinue = true;

Common Errors: Improper Stay Condition

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

i ↳ a.length

AIOBE



i == a.length

↳ iteration.

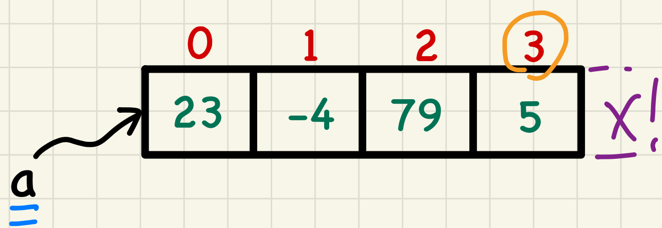
Common Errors: Improper Update to Loop Counter

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

4

fix

i++;



i	a[i]
0	a[1]
1	a[2]
2	a[3]
3	a[4] AIOOBE X

Common Errors: Improper Update to Stay Condition

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

userWantsToContinue = answer.equals("Y");

Logical Error: infinite loop if 1st iteration allowed

∵ userWantsToContinue not updated.

Common Errors: Improper Initial 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 ++; }
```

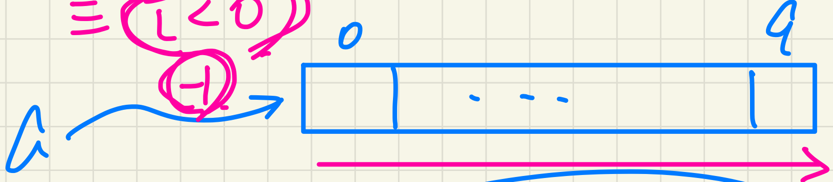
T

a[-1] ⇒ AIOBE

exit: !(i >= 0)

≡ i < 0

-1



a.length == 10

Common Errors: Misplaced Semicolon

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

entire loop

not body of the loop.

Console

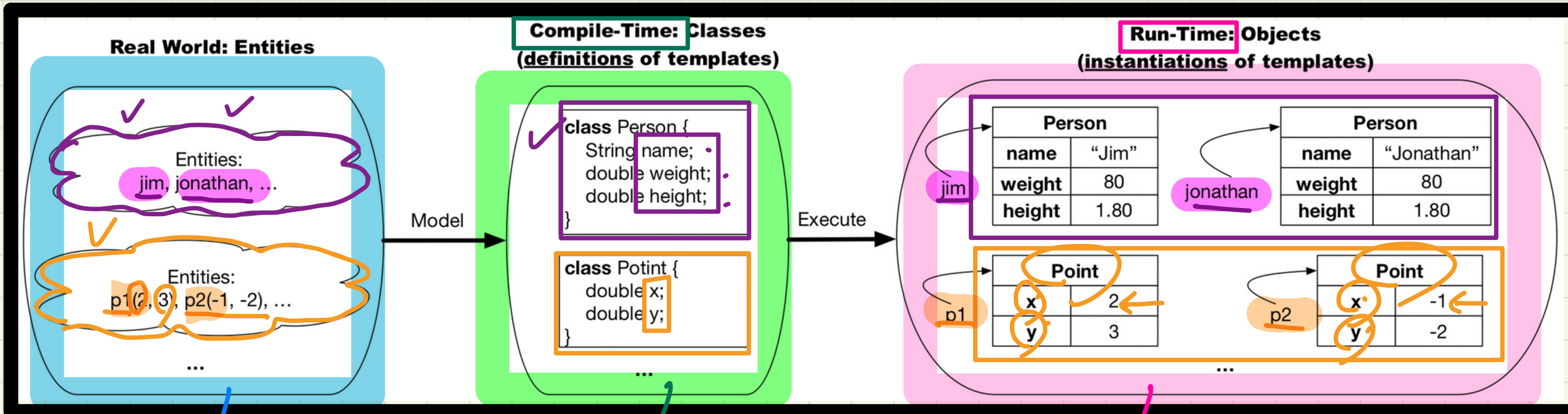
Hello.

Lecture 4

Part A

Classes and Objects - Object Orientation

Observe-Model-Execute Process



observe

model

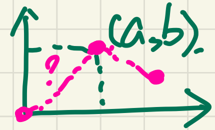
execute

Modelling: from Entities to Classes

Identify Critical Nouns & Verbs

classes attributes

accessors mutators



Example 1

class Point

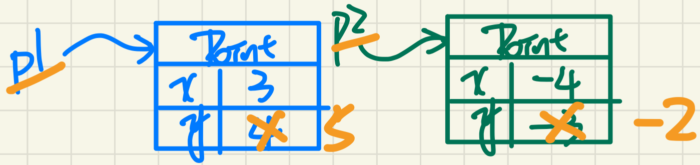
Points on a two-dimensional plane are identified by their signed distances from the X- and Y-axises. A point may move arbitrarily towards any direction on the plane. Given two points, we are often interested in knowing the distance between them.

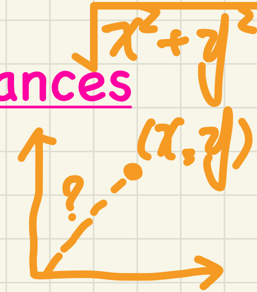
attribute (x, y)

Example 2

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.

OO Thinking: Templates vs. Instances






```java
public class Point {
  private double x;
  private double y;
}
```

- A *template* (e.g., class `Point`) defines what's **shared** by a set of related entities (i.e., 2-D points).
  - Common *attributes* (`x`, `y`)
  - Common *behaviour* (move left, move up)
- Each template may be *instantiated* as multiple instances, each with *instance-specific* values for attributes `x` and `y`:
  - `Point` instance `p1` is located at $(3, 4)$
  - `Point` instance `p2` is located at $(-4, -3)$
- Instances of the same template may exhibit *distinct behaviour*.
  - When `p1` moves up for 1 unit, it will end up being at $(3, 5)$
  - When `p2` moves up for 1 unit, it will end up being at $(-4, -2)$
  - Then, `p1`'s distance from origin: $[\sqrt{3^2 + 5^2}]$
  - Then, `p2`'s distance from origin: $[\sqrt{(-4)^2 + (-2)^2}]$

What Is a Method?

Header (def.).

$RT\ m(T_1\ p_1, T_2\ p_2, \dots, T_n\ p_n)\ \{ \dots \}$

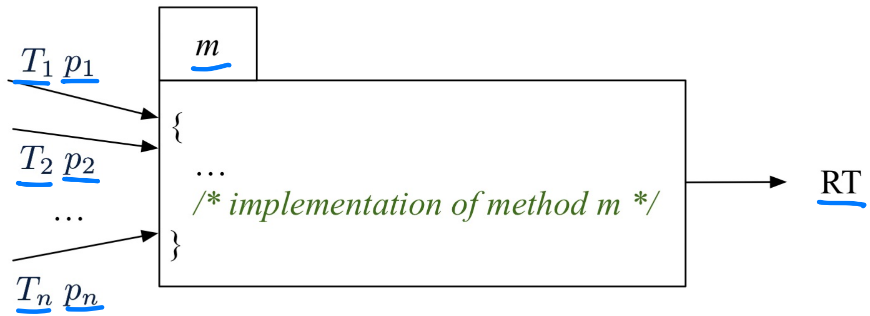
parameters.

Usage

$m(a_1, a_2, \dots a_n)$

arguments

- 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* [$p_1, p_2, \dots, p_n$]
  - The corresponding *parameter types* [$T_1, T_2, \dots, T_n$]
- A call to method *m* has the form: $m(a_1, a_2, \dots, a_n)$

  Types of *argument values* $a_1, a_2, \dots, a_n$ must match the the corresponding parameter types $T_1, T_2, \dots, T_n$.

Parameters vs. Arguments

parameters.

Template Definition

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

  double getDistanceFrom(Point other) {...}

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

① Method declared in the context object's type? ✓

Method Usages

② Arguments compatible with param. types?

p1.getDistanceFrom(p2)

context object

argument

```
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

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.

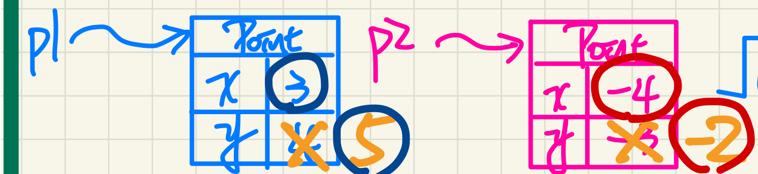
     e.g., `double bmi = jim.getBMI();`

     e.g., `println(p1.getDistanceFromOrigin());`

OOP: Creating and Manipulating Objects

$p1.x = 3$
$p1.y = 4$

$p2.x = -4$
$p2.y = -3$



$$\sqrt{(p1.x)^2+(p1.y)^2}$$

3 3 4 5

-4 -4 -3 -2

$$\sqrt{(p2.x)^2+(p2.y)^2}$$

```java
public class Point {
    private double x;
    private double y;

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

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

    public double getX() {
        return this.x;
    }

    public double getY() {
        return this.y;
    }

    public double getDistanceFromOrigin() {
        double dist =
            Math.sqrt(this.x * this.x
                    + this.y * this.y);
        return dist;
    }
}
```

3 -4 4 -3

p2.y += 1

p1.y += 1

p1 p2

1 1

p1 p2

p1.x

p2.x

p1 p2

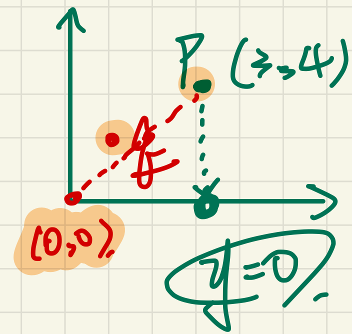
p1 p2

```java
public class PointTester {
    public static void main(String[] args) {
        Point p1 = new Point(3, 4);
        Point p2 = new Point(-4, -3);

        System.out.println("p1 " + "(" + p1.getX() + ", " + p1.getY() + ")");
        System.out.println("p2 " + "(" + p2.getX() + ", " + p2.getY() + ")");
        System.out.println(p1.getDistanceFromOrigin());
        System.out.println(p2.getDistanceFromOrigin());

        p1.moveUp(1);
        p2.moveUp(1);

        System.out.println("p1 " + "(" + p1.getX() + ", " + p1.getY() + ")");
        System.out.println("p2 " + "(" + p2.getX() + ", " + p2.getY() + ")");
        System.out.println(p1.getDistanceFromOrigin());
        System.out.println(p2.getDistanceFromOrigin());
    }
}
```

3

-4

Use of Accessors vs. Mutators



p.moveToXaxis().

p.getDistFrom(q).

p.getDistFromOr()

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

- Calls to *mutator methods* *cannot* be used as values. → void
  - e.g., `System.out.println(jim.setWeight(78.5));` ×
  - e.g., `double w = jim.setWeight(78.5);` ×
  - e.g., `jim.setWeight(78.5);` ✓ stands alone without being used → void
- Calls to *accessor methods* *should* be used as values.
  - e.g., `jim.getBMI();` × return value not used → compiles but not useful
  - 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)`

Lecture 4

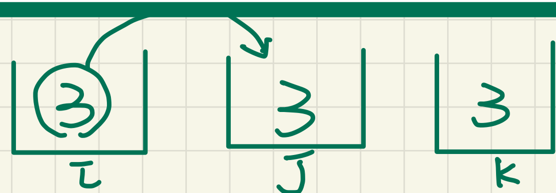
Part B

Classes and Objects - Reference Aliasing

Copying Primitive vs. Reference Values

Primitive

```
int i = 3;
int j = i;    System.out.println(i == j); /*true*/
int k = 3;    System.out.println(k == i && k == j); /*true*/
```

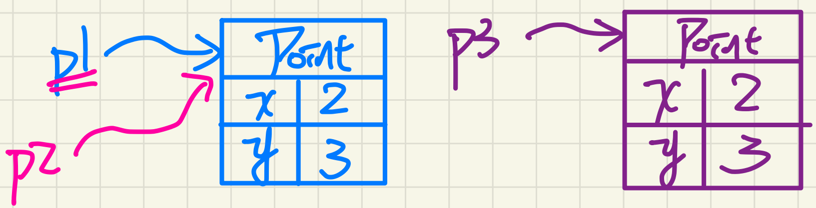


values of primitives

values of addresses

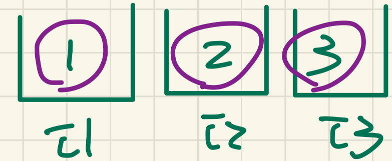
Reference

```
Point p1 = new Point(2, 3);
Point p2 = p1;  System.out.println(p1 == p2); /*true*/
Point p3 = new Point(2, 3);
Systme.out.println(p3 == p1 || p3 == p2); /*false*/
Systme.out.println(p3.x == p1.x && p3.y == p1.y); /*true*/
Systme.out.println(p3.x == p2.x && p3.y == p2.y); /*true*/
```



Copying Primitive Values

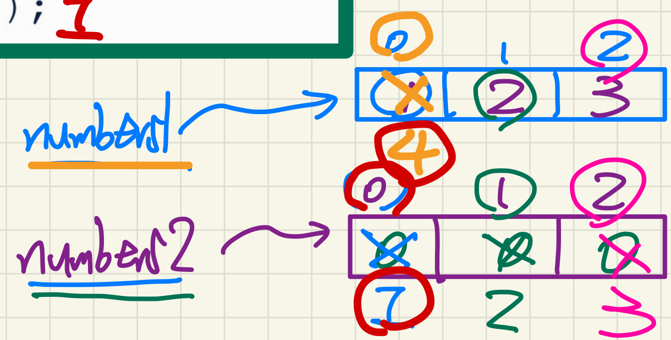
```
int i1 = 1;
int i2 = 2;
int i3 = 3;
int[] numbers1 = {i1, i2, i3};
int[] numbers2 = new int[numbers1.length];
for(int i = 0; i < numbers1.length; i ++) {
  numbers2[i] = numbers1[i];
}
numbers1[0] = 4;
System.out.println(numbers1[0]); 4
System.out.println(numbers2[0]); 1
```



1st: nums2[0] = nums1[0];

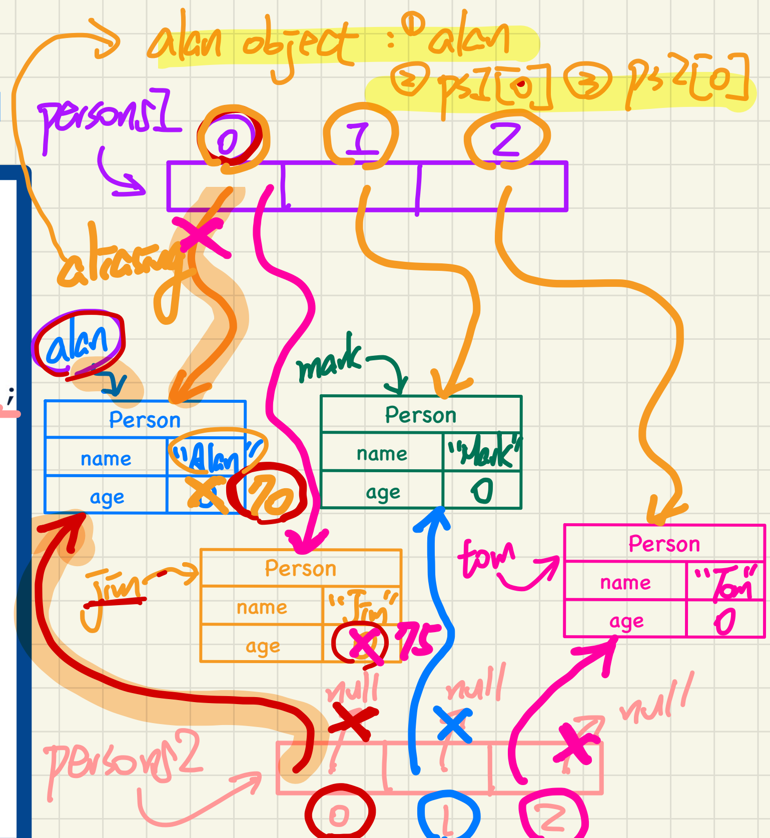
2nd: nums2[1] = nums1[1];

3rd: nums2[2] = nums1[2];



Copying Reference Values: Aliasing

```
Person alan = new Person("Alan");
Person mark = new Person("Mark");
Person tom = new Person("Tom");
Person jim = new Person("Jim");
Person[] persons1 = {alan, mark, tom}; *
Person[] persons2 = new Person[persons1.length];
for(int i = 0; i < persons1.length; i ++) {
  persons2[i] = persons1[i]; }
persons1[0].setAge(70);
System.out.println(jim.getAge()); 0
System.out.println(alan.getAge()); 70
System.out.println(persons2[0].getAge()); 70
persons1[0] = jim;
persons1[0].setAge(75);
System.out.println(jim.getAge()); 75
System.out.println(alan.getAge()); 70
System.out.println(persons2[0].getAge()); 70
```



* persons1 is an array of size 3 where each index stores the address of some Person object

persons1[0] = alan;
persons1[1] = mark;
persons1[2] = tom;

1st: ps2[0] = ps1[0]
2nd: ps2[1] = ps1[1]
3rd: ps2[2] = ps1[2]

<u>**Lecture 4**</u>

<u>**Part C**</u>

Classes and Objects -
Java Data Types,
Anonymous Objects

Variable Declaration

Type

boolean
double
char

name of var

primitive type ①

int — x

allowable value stored at runtime depends on the declared type

?

x

String — s

from Java library

Member — alan

reference type.

② from customised Java project

null

alan.setTrainer(...);

NullPointerException

x. — X

?

alan

allowable **address value** of an object instantiated from the declared type.

Reference-Typed Return Values

accessor (not modifying context object)

mutator (modifying c.o.)

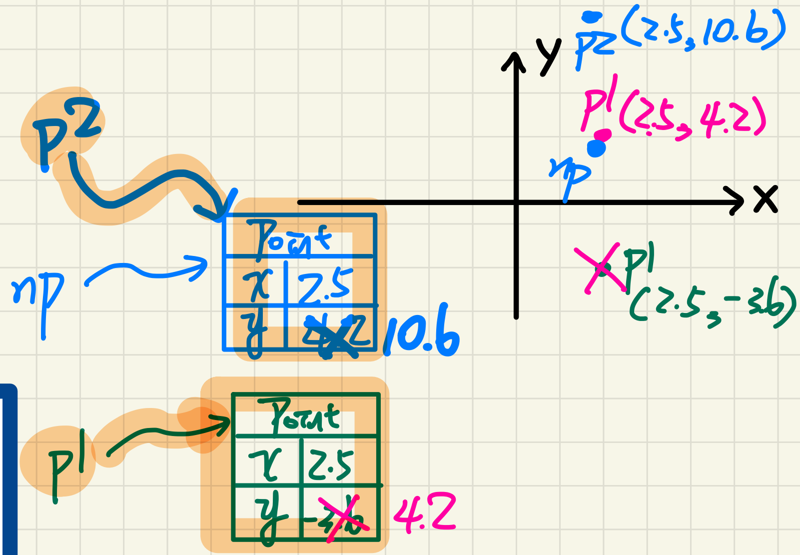
```java
public class Point {
  public void moveUpBy(int i) { this.y = this.y + i; }
  Point movedUpBy(int i) {
    Point np = new Point(x, y);
    np.moveUp(i);
    return np;
  }
}
```

$np.x = p1.x;$

$np.y = p1.y;$

```java
public class PointTester {
  public 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);
  }
}
```

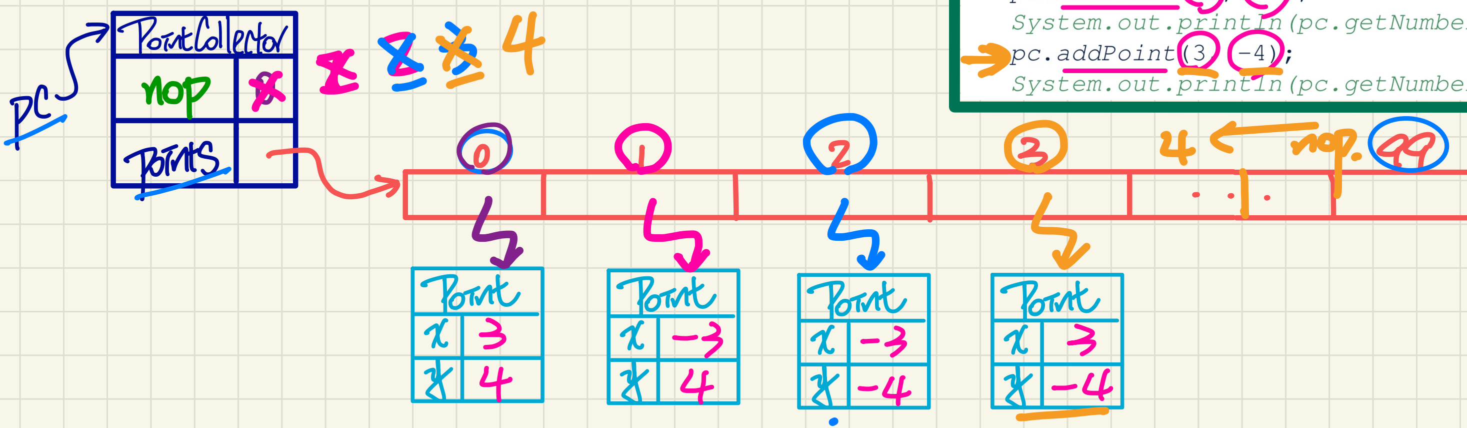
false.



Programming Pattern: Mutator

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

```java
public class PointCollectorTester {
  public static void main(String[] args) {
    PointCollector pc = new PointCollector();
    System.out.println(pc.getNumberOfPoints());
    pc.addPoint(3, 4);
    System.out.println(pc.getNumberOfPoints());
    pc.addPoint(-3, 4);
    System.out.println(pc.getNumberOfPoints());
    pc.addPoint(-3, -4);
    System.out.println(pc.getNumberOfPoints());
    pc.addPoint(3, -4);
    System.out.println(pc.getNumberOfPoints());
```



Programming Pattern: Accessor

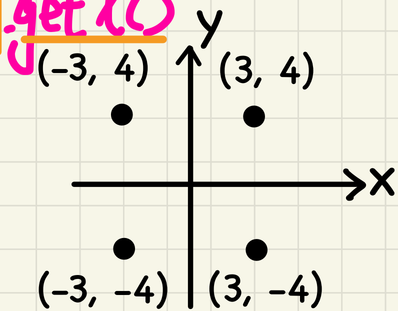
```
public Point[] getPointsInQuadrantI() {
  Point[] ps = new Point[this.nop];
  int count = 0; /* number of points in Quadrant I */
  for(int i = 0; i < this.nop; i++) {
    Point p = this.points[i];
    if(p.x > 0 && p.y > 0) { ps[count] = p; count++; } }
  Point[] q1Points = new Point[count];
  /*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

```
Point[] ps = pc.getPointsInQuadrantI();
System.out.println(ps.length); /* 1 */
System.out.println("(" +
  ps[0].getX() + ", " + ps[0].getY() + ")");
```

q1Points[0] = ps[0];

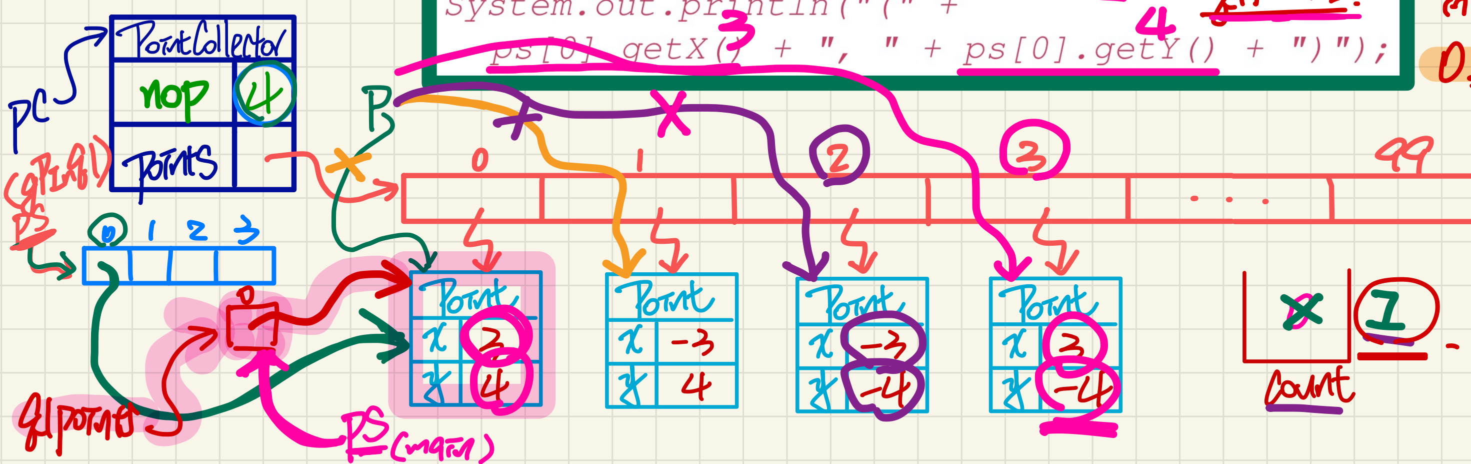
ps[0].getX()

C.O.



① there are "count" points in q1.

② these points are stored in indices: 0, ..., count-1



Lecture 4

Part D

Classes and Objects -
More Advanced Use of **this**

Example: Reference to this

jim.spouse != null || elsa.spouse != null

F F F

```java
public class Person {
  private String name;
  private Person spouse;
  public Person(String name) {
    this.name = name;
  }
  public void marry(Person other) {
    if(this.spouse != null || other.spouse != null) {
      /* Error: both must be single */
    }
    else { this.spouse = other; other.spouse = this; }
  }
}
```

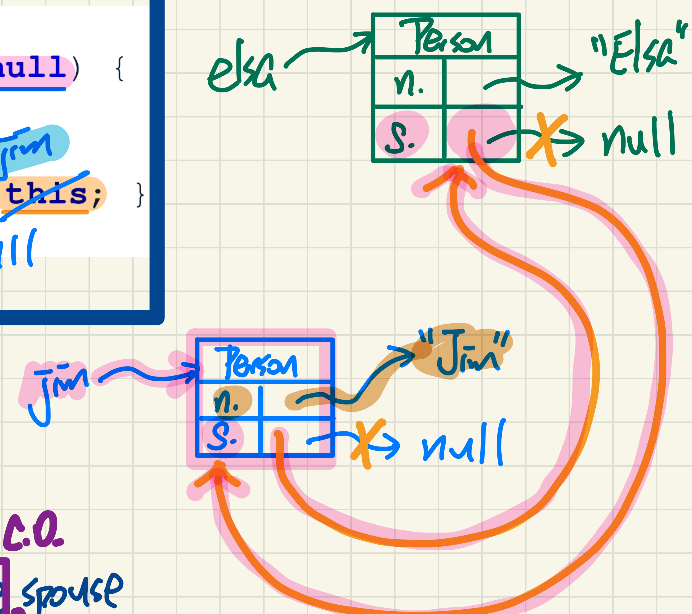
jim.spouse() spouse() name()

jim, elsa

this.spouse == null && other.spouse == null

jim.spouse = elsa;

elsa.spouse = jim;




```java
Person jim = new Person("Jim");
Person elsa = new Person("Elsa");
jim.marry(elsa);
```

C.O.

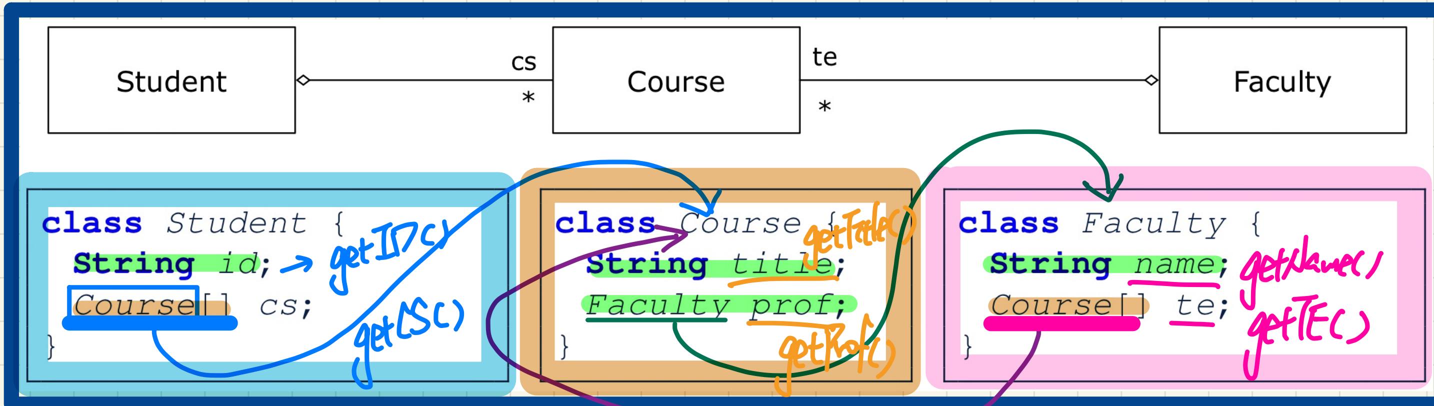
* jim.spouse.spouse

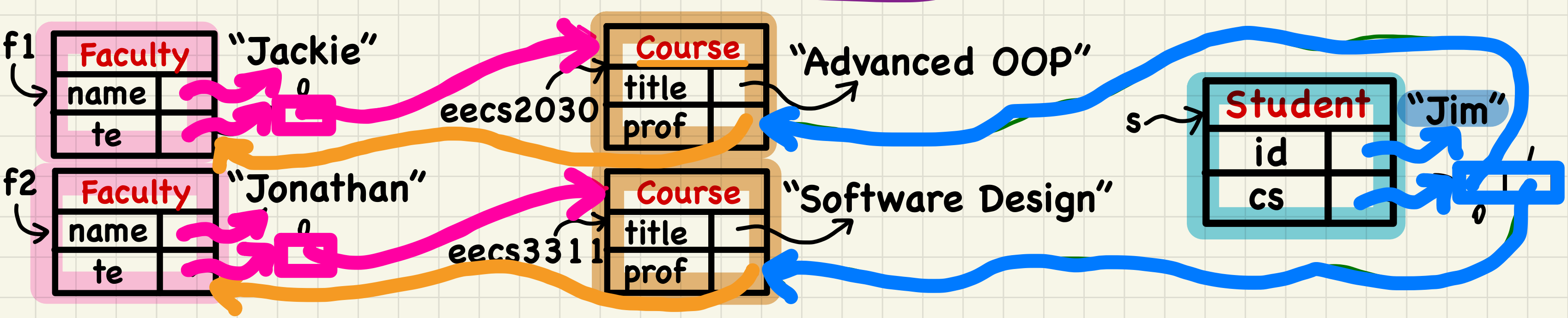
Lecture 4

Part E

Classes and Objects -
Navigating Classes via the Dot Notation

Object Structure: Student, Course, Faculty





Dot Notation for Navigating Classes (1)


```
class Student {
  String id;
  Course[] cs;
}
```

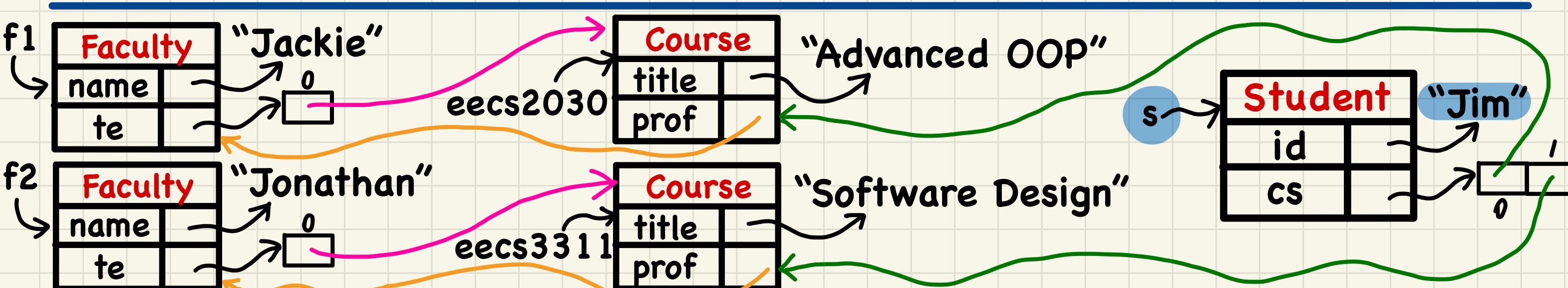
```
class Course {
  String title;
  Faculty prof;
}
```

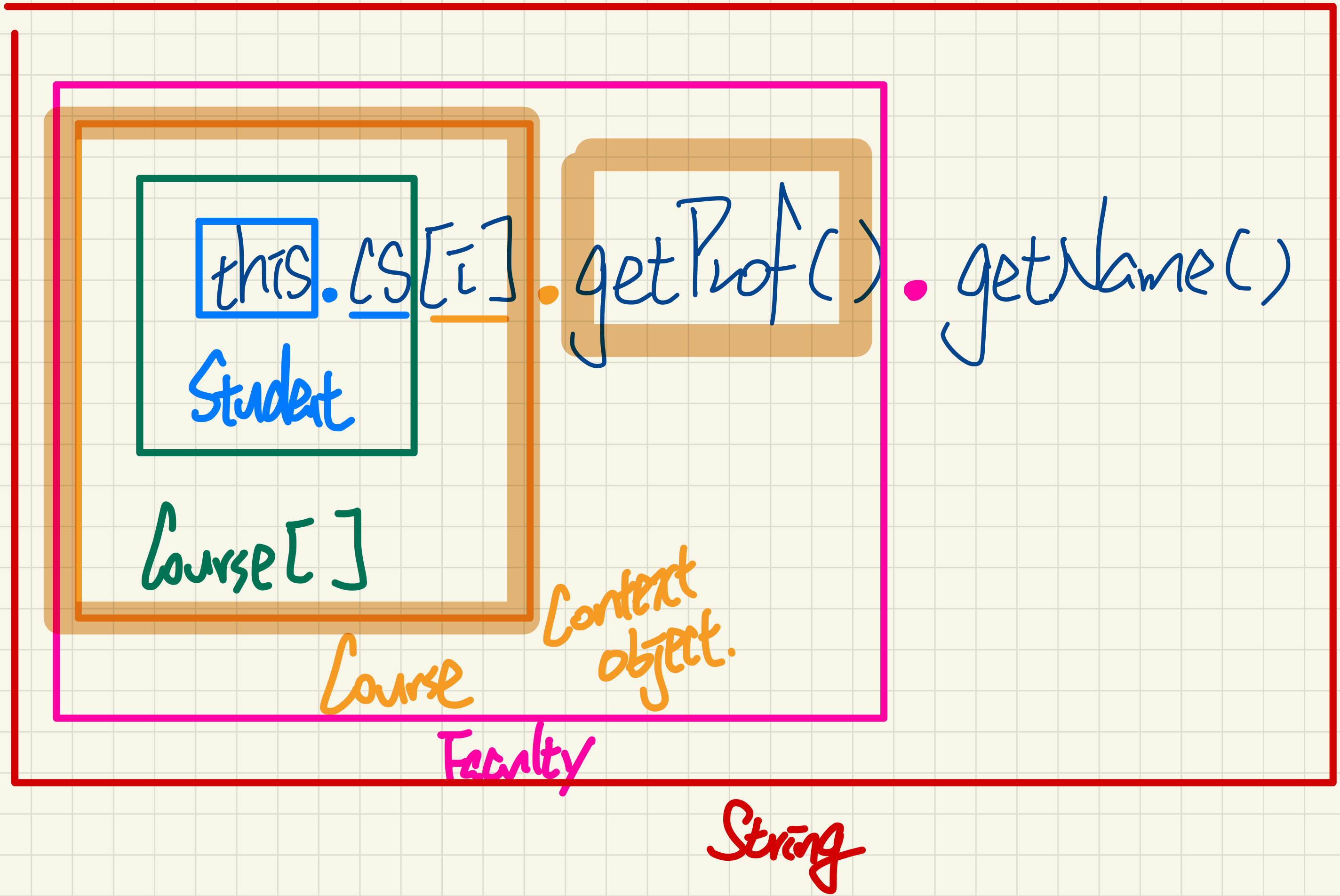
```
class Faculty {
  String name;
  Course[] te;
}
```

```
/* Get the student's id.
 */
String getID() {
  return this.id;
}
```

```
/* Title of ith course
 */
String getTitle(int i) {
  return this.cs[i].getTitle();
}
```

```
/* Name of
 * ith course's instructor
 */
String getName(int i) {
  return this.cs[i].getProf().getName()
}
```





this.cs[i].getProf().getName()
Student
Course[]
Course
Context object.
Faculty
String

Dot Notation for Navigating Classes (2)


```
class Student {
  String id;
  Course[] cs;
}
```

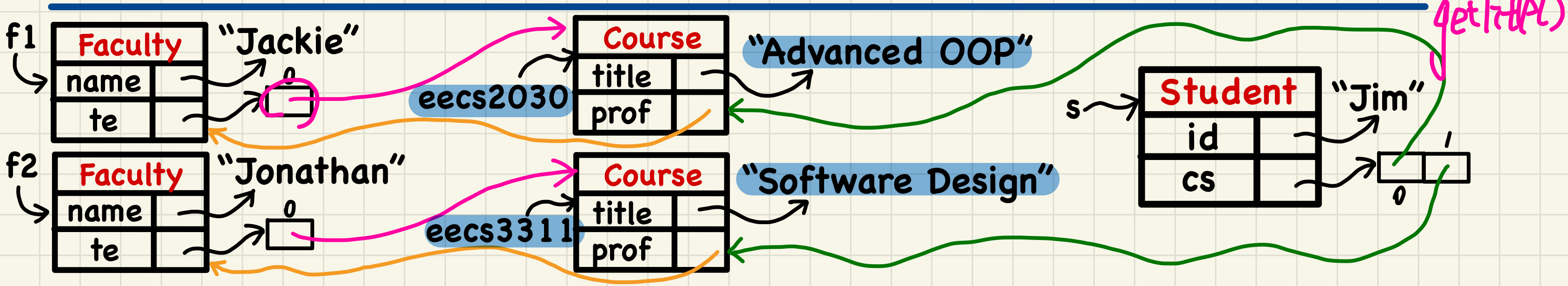
```
class Course {
  String title;
  Faculty prof;
}
```

```
class Faculty {
  String name;
  Course[] te;
}
```

```
/* Get course's title.
 */
String getTitle() {
  return this.getTitle();
}
```

```
/* Name of instructor
 */
String getName() {
  return this.prof.getName()
}
```

```
/* Title of instructor's
 * ith teaching course
 */
String getTitle(int i) {
  return this.getProf().getTE()[i].getTitle()
}
```



Dot Notation for Navigating Classes (3)

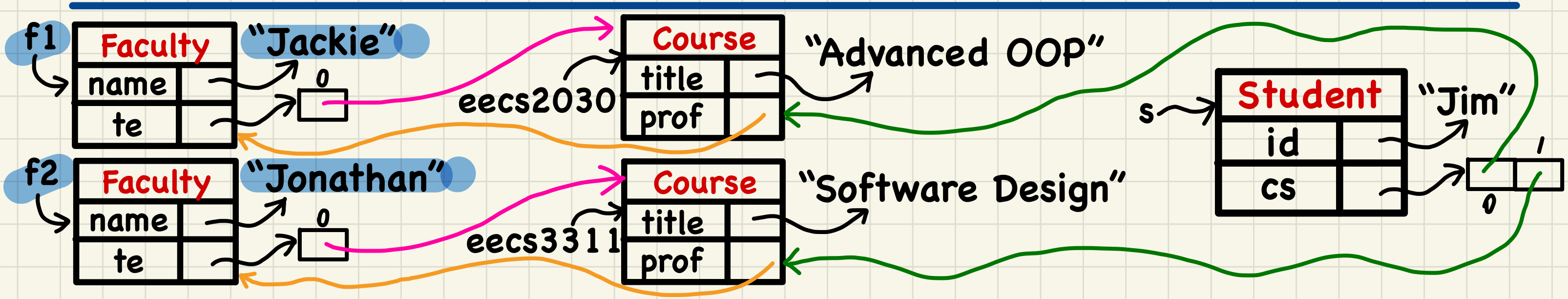

```
class Student {
  String id;
  Course[] cs;
}
```

```
class Course {
  String title;
  Faculty prof;
}
```

```
class Faculty {
  String name;
  Course[] te;
}
```

```
/* Name of instructor
 */
String getName() {
  return this.name;
}
```

```
/* Title of instructor's
 * ith teaching course
 */
String getTitle(int i) {
  return this.te[i].getTitle();
}
```



Lecture 4

Part F

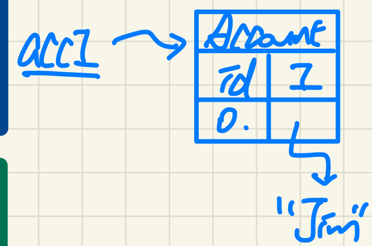
Classes and Objects - Static Variables

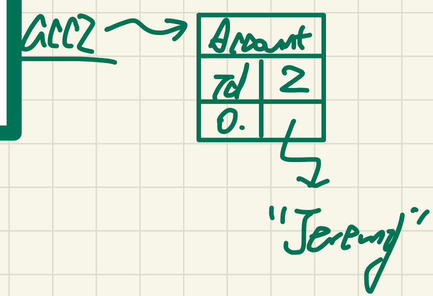
Managing Account IDs: Manual

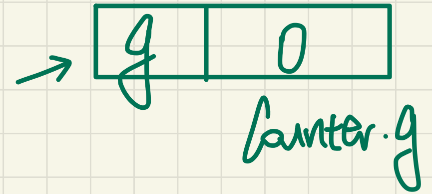
```java
public class Account {
  private int id;
  private String owner;
  public int getID() { return this.id; }
  public Account(int id, String owner) {
    this.id = id;
    this.owner = owner;
  }
}
```

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

1 2

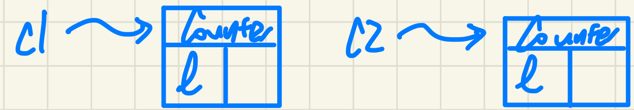






non-static variables

int l;



- attribute
- instance-specific: ① each object of the class has its own copy.
  ② To access it, a context object is necessary. e.g. c1.l.
- initialized in constructors.

static variables

static int g = 0;

- instance-independent: ① all objects of the class share the same copy.
  ② To access it, class name suffices.
- initialized upon declaration.

Declaring Global Variables among Objects

```java
public class Counter {
    private int l;
    static int g = 0;

    public Counter() {
        this.l = 0;
    }

    public int getLocal() {
        return this.l;
    }

    public void incrementLocal() {
        this.l ++;
    }

    public void incrementGlobal() {
        g ++;
    }
}
```

initialization.

init. of non-static variable.

c1, c2 — l specific to context obj.

c1 — g shared by all Counter instances.

static g already available

```java
public class CounterTester {
    public static void main(String[] args) {
        Counter c1 = new Counter();
        Counter c2 = new Counter();

        System.out.println("c1's local: " + c1.getLocal());
        System.out.println("c2's local: " + c2.getLocal());
        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 ++;
    }
}
```

access to static variables does not require a C.O.

use of a context object to access a static var is unnecessary.

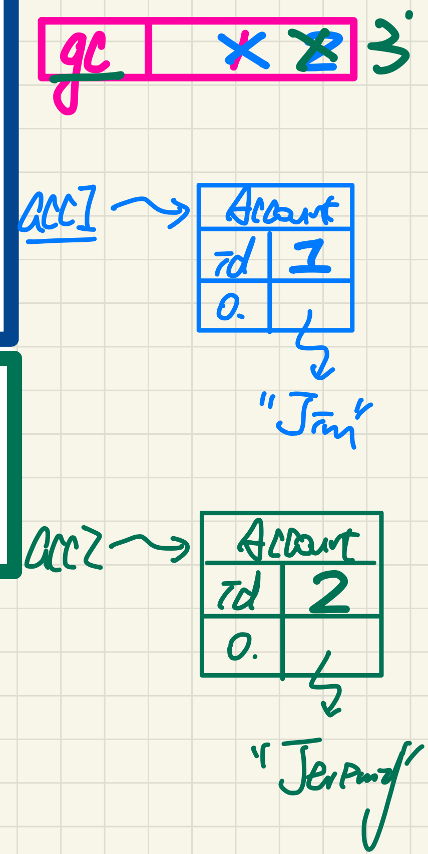
Managing Account IDs: Automatic

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

"Jim"

"Jeremy"

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



steve.accounts[steve.noa] = acc2;

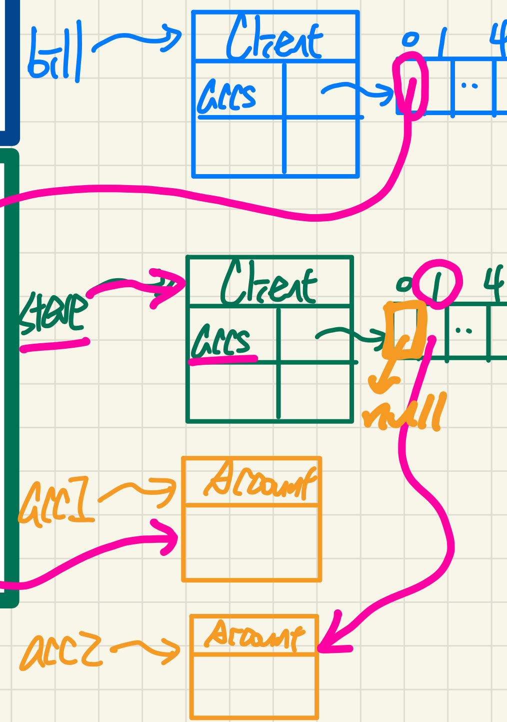
1

Misuse of Static Variables

noa | ~~0~~ ~~1~~ 2

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

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



bill.accounts[bill.noa] = acc1;

0

Use of Static Variables: Common Error

work but poor design.

static (good solution?).

```
public class Bank {
    private string branchName;
    public String getBrachName() { return this.branchName; }
    private static int nextAccountNumber = 0;
    public static String getInfo() {
        nextAccountNumber++;
        return this.branchName + nextAccountNumber;
    }
}
```

requires a context object

Design

static ① All bank objects share the same branch name?

non-static ② Each bank object has its own instance-specific branch name.

Cannot use non-static variable from a static context.

To access: Bank.getInfo()

cannot serve as a context object.

contradictory!

Lecture 5

Part A

Two-Dimensional Arrays - Nested Loops

Nested Loops: Semantics and Tracing

3

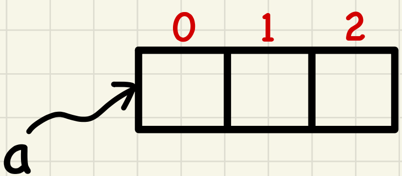
```
for(int i = 0; i < a.length; i ++) {
  for(int j = 0; j < a.length; j ++) {
    System.out.println("(" + i + ", " + j + ")");
} }
```

outer loop

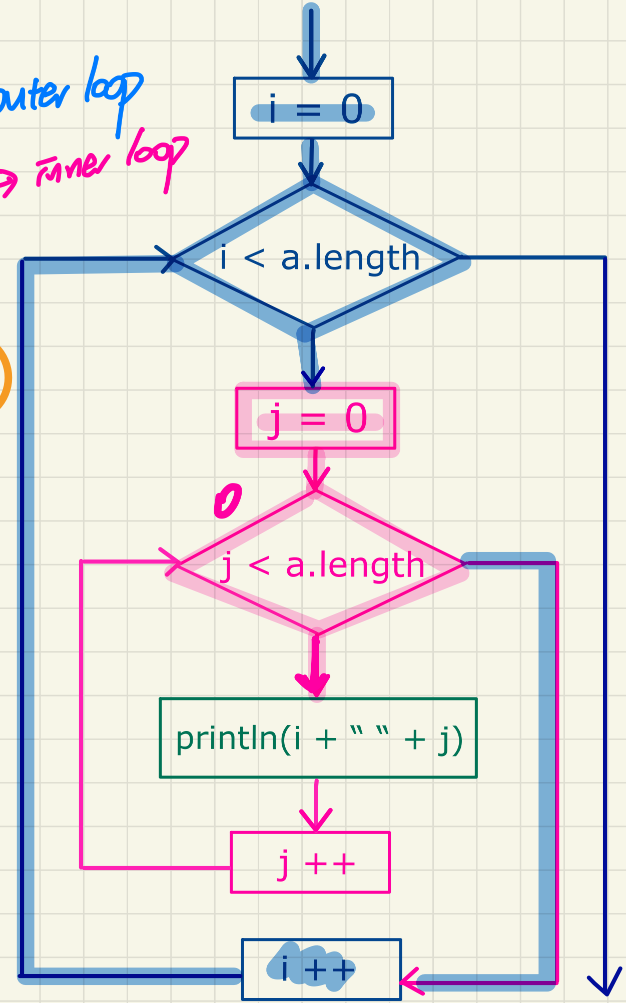
inner loop

3

$3 * 3 = 9$



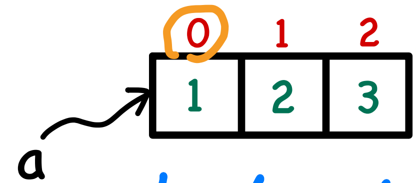
i	j
0	0
0	1
0	2
1	0
1	1
1	2
2	0
2	1
2	2



Computational Problem: Finding Duplicates

No Duplicates,
Redundant Scan

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



a.length == 100

$\boxed{100^2}$

redundant

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

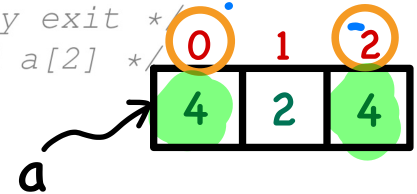
i==j

↳ unnecessary to check

Computational Problem: Finding Duplicates

Redundant Scan,
No Early Exit

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

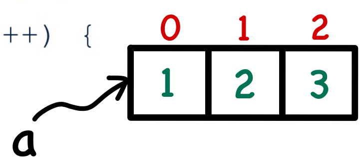
redundant to check further

we could have stopped here

Computational Problem: Finding Duplicates

No Duplicates,
Redundant Scan

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



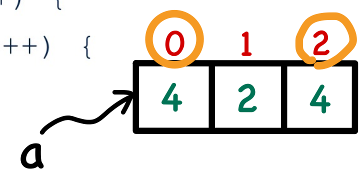
i	j	i != j	a[i]	a[j]	a[i] == a[j]	hasDup
0	0	*false*	1	1	*true*	*false*
0	1	*true*	1	2	*false*	*false*
0	2	*true*	1	3	*false*	*false*
1	0	*true*	2	1	*false*	*false*
1	1	*false*	2	2	*true*	*false*
1	2	*true*	2	3	*false*	*false*
2	0	*true*	3	1	*false*	*false*
2	1	*true*	3	2	*false*	*false*
2	2	*false*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

Computational Problem: Finding Duplicates


```java
/* Version 2 with redundant scan and early exit */
int[] a = {4, 2, 4}; /* duplicates: a[0] and a[2] */
boolean hasDup = false;
for(int i = 0; i < a.length && !hasDup; i ++) {
  for(int j = 0; j < a.length && !hasDup; j ++) {
    hasDup = i != j && a[i] == a[j];
  } /* end inner for */ } /* end outer for */
System.out.println(hasDup);
```

F

F



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

cause early exit as soon as a satisfaction witness is found.

Computational Problem: Finding Duplicates

No Duplicates, Non-Redundant Scan

got rid of : ① 0, 0 ② 0, 1 1, 0 ×

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

a

0	1	2	3
1	2	3	4

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*

n

v1, v2

i	j
0	0 ... n-1 (n)
1	0 ... n-1 (n)

v1, v2 $\boxed{n * n}$

v3

i	j
0	$\boxed{1..n-1}$ (n-1)
1	$\boxed{2..n-1}$ (n-2)
⋮	
n-1	n ×

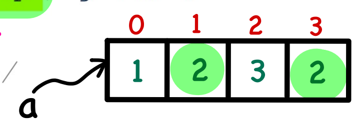
$\boxed{(n-1) + (n-2) + \cdots + 1}$ v3

Computational Problem: Finding Duplicates

Duplicates,
Non-Redundant Scan,
Early Exit

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

F
F



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

→ exit from both loops.

Lecture 6

Part A

API of Java Library -
Method Headers,
Static vs. Non-Static Methods

```
class A {
    public static int m1 (int i, String s){
        ....
    }
    public int m2(String s, int i) {
        ...
    }
}
```

parameters.

arguments.

Tester.

```
int j = A.m1(23, "Alan");
A obj = new A();
int k = obj.m2("Tom", 46);
```

Lecture 6

Part B

API of Java Library - Case Study: Math Class

Java API: Math

modifier

Math.abs(...)

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

↳ return types

method overloading
- same method names
- distinct types of parameter types.

method header:

```
public class Math {
  public static int abs(int a) {...}
}
```

omitted

`static double`	`random()` Returns a `double` value with a positive sign, greater than or equal to `0.0` and less than `1.0`.

inclusive

exclusive

Math.random() → [0.0, 1.0)
↳ [0.0, 100.0)

e.g. 0.01 * 100 → 1.00

0.01234 ↳ * 100 → 1.234

0.123 * 100 → 12.3 → 12

Lecture 6

Part C

***API of Java Library -
Case Study: ArrayList Class***

API: ArrayList<E>

declaration

generic parameter type elements.

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

instantiation of E by Person

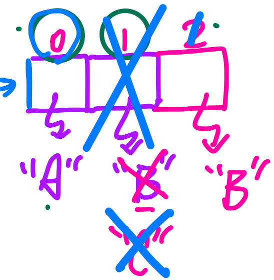
non-static

int	**size**() Returns the number of elements in this list.
boolean	**add**(E e) Person Appends the specified element to the end of this list.
void	**add**(int index, E element) Person 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 Person	**remove**(int index) 2 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 Person	**get**(int index) → a[index]. Returns the element at the specified position in this list.

overloaded methods.

Use of ArrayList<String>

Instantiating generic parameter E by String. e.g. void add(~~E~~ e)
String




```
import java.util.ArrayList;
public class ArrayListTester {
  public static void main(String[] args) {
    ArrayList<String> list = new ArrayList<String>();
    println(list.size());   0
    println(list.contains("A"));   false
    println(list.indexOf("A"));   -1
    list.add("A");
    list.add("B");
    println(list.contains("A")); println(list.contains("B")); println(list.contains("C"));
           True                        True                        False
    println(list.indexOf("A")); println(list.indexOf("B")); println(list.indexOf("C"));
           0                           1                           -1
    list.add(1, "C");
    println(list.contains("A")); println(list.contains("B")); println(list.contains("C"));
           T                           T                           T
    println(list.indexOf("A")); println(list.indexOf("B")); println(list.indexOf("C"));
           0                           2                           1
    list.remove("C");
    println(list.contains("A")); println(list.contains("B")); println(list.contains("C"));
           T                           T                           F
    println(list.indexOf("A")); println(list.indexOf("B")); println(list.indexOf("C"));
           0                           1                           -1

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

a[i]

Lecture 6

Part D

API of Java Library -
Case Study: Hashtable Class

Hash Table

- 2-column table
- Column of keys contain no duplicates.
- Column of values may contain duplicates.
- Each key uniquely identifies an entry (k, v)

- key
- value

grades.

keys	values
"Alan"	"A"
"Mark"	"B+"
"Tom"	"C"

no duplicates.

API: HashTable<K, V>

generic parameters.

K: type of keys

V: type of values

Hashtable<String, Person> t = new 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~~ Person	**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~~ String 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.

t.get(...) ↳ Person

t.put("Key1", new Person(...));

Object key: used to uniquely identify an entry.

Use of HashTable<String, String>

K V.

grades →

keys	values
~~Alan~~	~~A~~
Mark	~~B+~~ F
Tom	C

```
import java.util.Hashtable;
public class HashTableTester {
  public static void main(String[] args) {
    Hashtable<String, String> grades = new Hashtable<String, String>();      < >
    System.out.println("Size of table: " + grades.size());   0
    System.out.println("Key Alan exists: " + grades.containsKey("Alan"));   F
    System.out.println("Value B+ exists: " + grades.containsValue("B+"));   F
    grades.put("Alan", "A");
    grades.put("Mark", "B+");
    grades.put("Tom", "C");
    System.out.println("Size of table: " + grades.size());   3
    System.out.println("Key Alan exists: " + grades.containsKey("Alan"));   T
    System.out.println("Key Mark exists: " + grades.containsKey("Mark"));   T
    System.out.println("Key Tom exists: " + grades.containsKey("Tom"));   T
    System.out.println("Key Simon exists: " + grades.containsKey("Simon"));   F
    System.out.println("Value A exists: " + grades.containsValue("A"));   T
    System.out.println("Value B+ exists: " + grades.containsValue("B+"));   T
    System.out.println("Value C exists: " + grades.containsValue("C"));   T
    System.out.println("Value A+ exists: " + grades.containsValue("A+"));   F
    System.out.println("Value of existing key Alan: " + grades.get("Alan"));   A
    System.out.println("Value of existing key Mark: " + grades.get("Mark"));   B+
    System.out.println("Value of existing key Tom: " + grades.get("Tom"));   C
    System.out.println("Value of non-existing key Simon: " + grades.get("Simon"));   null
    grades.put("Mark", "F");   → existing key.
    System.out.println("Value of existing key Mark: " + grades.get("Mark"));   F
    grades.remove("Alan");
    System.out.println("Key Alan exists: " + grades.containsKey("Alan"));   F
    System.out.println("Value of non-existing key Alan: " + grades.get("Alan"));   null
```

I hope you enjoyed the journey.

All the Best !

JACKIE