Elementary Programming

EECS1022 Sections M & N:
Programming for Mobile Computing
Winter 2021

CHEN-WEI WANG

Learning Objectives

- Learn *ingredients* of programming:
  - development process: compile time (syntax, types) vs. runtime
  - literal values
  - data types [numbers, characters, strings]
  - named constants vs. variables
  - variable (re-)assignments
  - operations, operators, operands [arithmetic, relational]
  - expressions, type correctness
  - coercion vs. casting
  - input and output
  - escape sequences
- Given a problem:
  - First, plan how you would solve it mathematically.
  - Then, *Implement* your solution by writing a Java program.

Extra Practice?

- Java project archive:
  `Lecture_01_Elementary_Programming.zip`
  contains some of the lecture examples.
  Expanded it: *remaining examples* and *your own examples*!
- *Optional (but recommended)*: **Videos 1 – 7** from W19 Java tutorial:
  `https://www.eecs.yorku.ca/~jackie/teaching/tutorials/index.html#java_from_scratch`

Entry Point of Execution: the "main" Method

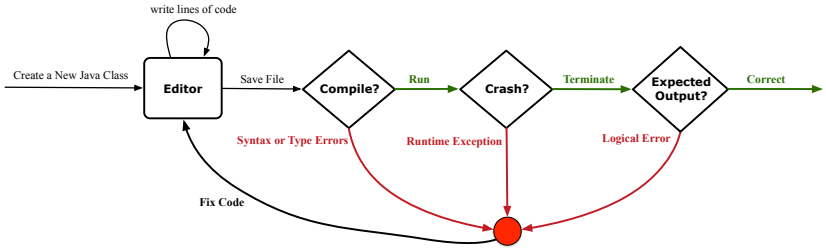
Most of your lecture examples or exercises will be defined within the context of the *main* method (for some *console application*).

```
public class MyClass {
  public static void main(String[] args) {
    /* Your programming solution is defined here. */
  }
}
```

The *main* method is treated by Java as the *starting point* of executing your program.

- *Sequential* Execution:
  - The execution starts with the first line in the *main* method, proceed line by line, from top to bottom, until there are no more lines to execute, then it *terminates*.
  - The *main* method may call methods from other classes (e.g., `model` package).

Development Process



Compile Time Errors vs. Run Time Errors

At the end of a programming test, if your submitted program:

- Cannot compile
  - ⇒ Your program cannot even be run
  - ⇒ ***Zero***!

  What you should do:
  Practice writing as many programs as possible.

- Compiles, but run with exceptions or unexpected outputs.
  - ⇒ Not necessarily zero, but likely ***low marks***!

  What you should do:
  Truly understand the logic/rationale beyond programs.

Compile Time vs. Run Time

- These terms refer to two stages of developing your program.
- *Compile time*: when editing programs in Eclipse.
  - There are two kinds of *compile-time errors*:
  - *Syntax errors*: your program does not conform to Java's grammar.
    - e.g., missing the semicolon, curly braces, or round parentheses
    - Java syntax is defined in the Java language specification.
  - *Type errors*: your program manipulates data in an inconsistent way
    e.g., `"York" * 23` [$\cdot$ multiplication is only for numbers]
- *Run time* is when executing/running the *main* method.
  - *Exceptions*: your program crashes and terminates *abnormally*
    - e.g., `ArithmeticException` (e.g., `10 / 0`), `ArrayIndexOutOfBoundException`, `NullPointerException`.
  - *Logical errors*: your program terminates *normally* but does not behave as expected
    e.g., calculating area of a circle with radius r using $2 \cdot \pi \cdot r$

Always Document Your Code

- Each important design or implementation *decision* should be carefully *documented* at the right place of 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.
 */
```

- Comments **do not affect** the runtime behaviour of programs.
- Comments are **only interpreted by** human developers.
  ⇒ Useful for ***revision*** and ***extension***.

Literals (1)

A *literal* is a *constant value* that appears directly in a program.

1. *Character* Literals
   - A single character enclosed within a pair of single quotes
   - e.g., `'a'`, `'1'`, `'*'`, `'('`, `' '`
   - It is invalid to write an empty character: `''`
2. *String* Literals
   - A (possibly empty) sequence of characters enclosed within a pair of double quotes
   - e.g., `""`, `"a"`, `"York"`, `"*#@$"`, `" "`
3. *Integer* Literals
   - A non-empty sequence of numerical digits
   - e.g., 0, -123, 123, 23943
4. *Floating-Point* Literals
   - Specified using a combination of an integral part and a fractional part, separated by a decimal point, or using the scientific notation
   - e.g., 0.3334, 12.0, 34.298, 1.23456E+2 (for 1.23456×10^2), 1.23456E-2 (for 1.23456×10^{-2})

Operations

An *operation* refers to the process of applying an *operator* to its *operand*(s).

1. *Numerical* Operations [results are numbers]
   - e.g., `1.1 + 0.34`
   - e.g., `13 / 4` [quotient: `3`]
   - e.g., `13.0 / 4` [precision: `3.25`]
   - e.g., `13 % 4` [remainder: `1`]
   - e.g., `-45`
   - e.g., `-1 * 45`
2. *Relational* Operations [results are true or false]
   - e.g., `3 <= 4` [*true*]
   - e.g., `5 < 3` [*false*]
   - e.g., `56 == 34` [*false*]
3. *String* Concatenations [results are strings]
   - e.g., `"York" + " " + "University"` is equivalent to `"York University"`

Java Data Types

A (data) type denotes a set of related *runtime values*.

1. *Integer* Type

`byte`	8 bits	$[-128, \ldots, -1, 0, 1, \ldots, 2^7 - 1]$
`short`	16 bits	$[-2^{15}, 2^{15} - 1]$
`int`	32 bits	$[-2^{31}, 2^{31} - 1]$
`long`	64 bits	$[-2^{63}, 2^{63} - 1]$

2. *Floating-Point Number* Type

`float`	32 bits
`double`	64 bits

3. *Character* Type
   `char`: the set of single characters
4. *String* Type
   `String`: the set of all possible character sequences

*Declaring a variable `v` to be of type `T` constrains `v` to store **only** those values defined in `T`.*

Named Constants vs. Variables

A *named constant* or a *variable*:

- Is an identifier that refers to a *placeholder*
- Must be declared with its *type* (of stored value) before use:

```
final double PI = 3.14159; /* a named constant */
double radius; /* an uninitialized variable */
```

- Can only store a value that is *compatible with its declared type*

However, a *named constant* and a *variable* are different in that:

- A named constant must be *initialized*, and cannot change its stored value.
- A variable may change its stored value as needed.

Assignments

An *assignment* designates a value for a variable, or initializes a *named constant*.

That is, an assignment replaces the *old value* stored in a placeholder with a *new value*.

An *assignment* is done using the assignment operator (=).

An *assignment operator* has two operands:

- The *left* operand is called the *assignment target*
  which must be a variable name
- The *right* operand is called the *assignment source*
  which must be an expression whose type is *compatible* with the declared type of *assignment target*

e.g., This is a *valid* assignment:

```
String name1 = ``Heeyeon'';
```

e.g., This is an *invalid* assignment:

```
String name1 = (1 + 2) * (23 % 5);
```


Multiple Executions of Same Print Statement

Executing *the same print statement* multiple times *may or may not* output different messages to the console.

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);
```

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);
```


Identifiers & Naming Conventions

- Identifiers are *names* for identifying Java elements: *classes*, *methods*, *constants*, and *variables*.
- An identifier:
  - Is an arbitrarily long sequence of characters: letters, digits, underscores (_), and dollar signs ($).
  - Must start with a letter, an underscore, or a dollar sign.
  - Must not start with a digit.
  - Cannot clash with reserved words (e.g., `class`, `if`, `for`, `int`).
- *Valid* ids: `$2`, `Welcome`, `name`, `_name`, `YORK_University`
- *Invalid* ids: `2name`, `+YORK`, `Toronto@Canada`
- More conventions:
  - Class names are compound words, all capitalized:
    e.g., `Tester`, `HelloWorld`, `TicTacToe`, `MagicCardGame`
  - Variable and method names are like class names, except 1st word is all lower cases: e.g, `main`, `firstName`, `averageOfClass`
  - Constant names are underscore-separated upper cases:
    e.g., `PI`, `USD_IN_WON`

Variables: Common Mistakes (1)

Mistake: The same variable is declared more than once.

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

Fix 1: Assign the new value to the same variable.

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

Fix 2: Declare a new variable (with a different name).

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

Which fix to adopt depends on what you need!

Variables: Common Mistakes (2)

Mistake: A variable is used before it is declared.

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

Fix: Move a variable's declaration before its very first usage.

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

Remember, Java programs are always executed, line by line, *from top to bottom*.

Numerical Type Conversion: Coercion

- *Implicit* and *automatic* type conversion
- Java *automatically* converts an integer value to a real number when necessary (which adds a fractional part).

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

- However, does the following work?

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

  - RHS evaluates to `13.5` due to coercion.
  - LHS declares a variable for storing integers (with no fractional parts).

∴ Not compatible [***compile-time error***]

⇒ Need a way to "truncate" the fractional part!

Expressions (1)

An *expression* is a composition of *operations*.
An expression may be:

- *Type Correct*: for each constituent operation, types of the *operands* are compatible with the corresponding *operator*.

  e.g., `(1 + 2) * (23 % 5)`

  e.g., ```Hello '' + ``world''`

- *Not Type Correct*

  e.g., ```46'' % ``4''`

  e.g., `(``YORK '' + ``University'') * (46 % 4)`

  - `` ``YORK'' `` and `` ``University'' `` are both strings
    ∴ LHS of `*` is *type correct* and is of type `String`
  - `46` and `4` are both integers
    ∴ RHS of `%` is *type correct* and is of type `int`
  - Types of LHS and RHS of `*` are not compatible
    ∴ Overall the expression (i.e., a multiplication) is *not type correct*

Numerical Type Conversion: Casting

- *Explicit* and *manual* type conversion
- **Usage 1**: To assign a real number to an integer variable, you need to use explicit *casting* (which throws off the fractional part).

```
int value3 = (int) 3.1415926;
```

- **Usage 2:** You may also use explicit *casting* to force precision.
  - ```
    System.out.println(1 / 2);  /* 0 */
    ```
    ∵ When both operands are integers, division evaluates to quotient.
  - ```
    System.out.println( ((double) 1) / 2 ); /* 0.5 */
    System.out.println( 1 / ((double) 2) ); /* 0.5 */
    System.out.println( ((double) 1) / ((double) 2) ); /* 0.5 */
    ```
    ∵ Either or both of the integers operands are cast to `double` type
  - ```
    System.out.println((double) 1 / 2);  /* 0.5 */
    ```
    ∵ Casting has *higher precedence* than arithmetic operation.
  - ```
    System.out.println((double) (1 / 2));  /* 0.0 */
    ```
    ∵ Order of evaluating division is forced, via parentheses, to occur first.

Numerical Type Conversion: Exercise

Consider the following Java code:

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

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

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


Expressions (2.1)

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

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

- **L1**: YES [coercion]
- **L2**: NO [cast assignment source, i.e., `(int) 23.6`]
- **L3**: NO [cannot assign char to string]
- **L4**: NO [cannot assign string to char]

Expressions (2.2)

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

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

- **L1**: YES [proper cast]
- **L2**: YES [coercion]
- **L3**: YES [string literal assigned to string var.]
- **L4**: YES [type-correct string concat. assigned to string var.]
- **L5**: NO [string × number is undefined]

Augmented Assignments

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

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

- Java supports special operators for these:

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

- Java supports operators for incrementing or decrementing by 1:

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

- *Confusingly*, these increment/decrement assignment operators can be used in assignments:

```
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 */
```

Escape Sequences

An *escape sequence* denotes a single character.

- Specified as a *backslash* (\) followed by a *single character*
  - e.g., \t, \n, \', \", \\
- *Does not mean literally*, but means specially to Java compiler
  - \t means a tab
  - \n means a new line
  - \\ means a back slash
  - \' means a single quote
  - \" means a double quote
- May use an *escape sequence* in a character or string literal:
  - `''` [INVALID; need to escape ']
  - `'\''` [VALID]
  - `'"'` [VALID; no need to escape "]
  - `""` [INVALID; need to escape "]
  - `"\""` [VALID]
  - `"' "` [VALID; no need to escape ']
  - `"\n\t\""` [VALID]

Case Study 1: Compute the Area of a Circle

Problem: declare two variables `radius` and `area`, initialize `radius` as 20, compute the value of `area` accordingly, and print out the value of `area`.

```
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);
  }
}
```

It would be more flexible if we can let the user specify the inputs via keyboard!

Input and Output

Reading input from the console enables *user interaction*.

```
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);
  }
}
```


Useful Methods for Scanner

- *nextInt()* which reads an integer value from the keyboard
- *nextDouble()* which reads a double value from the keyboard
- *nextLine()* which reads a string value from the keyboard

Case Study 2: Display Time

Problem: prompt the user for an integer value of seconds, divide that value into minutes and remaining seconds, and print the results. For example, given an input 200, output "200 seconds is 3 minutes and 20 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();
    int minutes = seconds / 60; /* minutes */
    int remainingSeconds = seconds % 60; /* seconds */
    System.out.print(seconds + " seconds is ");
    System.out.print(" minutes and ");
    System.out.println(remainingSeconds + " seconds");
  }
}
```


Where May Assignment Sources Come From?

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

- A literal

```
int i = 23;
```

- A variable

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

- An expression involving literals and variables

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

- An input from the user

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


Literals (2)

Q. Outputs of `System.out.println('a')` versus `System.out.println(''a'')`? [SAME]

Q. Result of comparison `''a'' == 'a'`? [TYPE ERROR]

- Literal `''a''` is a string (i.e., *character sequence*) that consists of a single character.
- Literal `'a'` is a single *character*.

∴ You cannot compare a character sequence with a character.

`print` vs. `println`

Executing `System.out.println(someString)` is the same as executing `System.out.print(someString + "\n")`.

- e.g.,

```
System.out.print("Hello");
System.out.print("World");
```

```
HelloWorld
```

- e.g.,

```
System.out.println("Hello");
System.out.println("World");
```

```
Hello
World
```

Beyond this lecture. . .

- Create a *console tester* in Eclipse.
  Try out the examples given in the slides.
- *Optional (but recommended)*: **Videos 1 – 7** from W19 Java tutorial:
  `https://www.eecs.yorku.ca/~jackie/teaching/tutorials/index.html#java_from_scratch`
- See `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html` for more information about data types in Java.

Index (1)

Learning Objectives
Extra Practice?
Entry Point of Execution: the “main” Method
Development Process
Compile Time vs. Run Time
Compile Time Errors vs. Run Time Errors
Always Document Your Code
Literals (1)
Operations
Java Data Types
Named Constants vs. Variables

Index (2)

Assignments
Multiple Executions of Same Print Statement
Identifiers and Naming Conventions in Java
Variables: Common Mistakes (1)
Variables: Common Mistakes (2)
Expressions (1)
Numerical Type Conversion: Coercion
Numerical Type Conversion: Casting
Numerical Type Conversion: Exercise
Expressions (2.1)
Expressions (2.2)

Index (3)

Augmented Assignments
Escape Sequence
Case Study 1: Compute the Area of a Circle
Input and Output
Useful Methods for Scanner
Case Study 2: Display Time
Where May Assignment Sources Come From?
Literals (2)
`print` vs. `println`
Beyond this lecture. . .

Selections

EECS1022 Sections M & N:
Programming for Mobile Computing
Winter 2021

CHEN-WEI WANG

Learning Outcomes

- The Boolean Data Type
- `if` Statement
- Compound vs. Primitive Statement
- Logical Operations
- Common Errors and Pitfalls

Extra Practice?

- Java project archive: `Lecture_02_Selections.zip`
  contains some of the lecture examples.
  Expanded it: *remaining examples* and *your own examples*!
- *Optional (but recommended)*: **Videos 10 – 17** from W19 Java tutorial:
  `https://www.eecs.yorku.ca/~jackie/teaching/tutorials/index.html#java_from_scratch`

Motivating Examples (1.1)

```
1  import java.util.Scanner;
2  public class ComputeArea {
3    public static void main(String[] args) {
4      Scanner input = new Scanner(System.in);
5      System.out.println("Enter the radius of a circle:");
6      double radiusFromUser = input.nextDouble();
7      final double PI = 3.14;
8      double area = radiusFromUser * radiusFromUser * PI;
9      System.out.print("Circle with radius " + radiusFromUser);
10     System.out.println(" has an area of " + area);
11     input.close();
12   }
13 }
```

- When the above Java class is run as a Java Application, **Line 4** is executed first, followed by executing **Line 5**, ..., and ended with executing **Line 11**.
- In **Line 6**, the radius value comes from the user. Any problems?

Motivating Examples (1.2)

- 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
```

  In this case, the area should *not* have been calculated!
- We need a mechanism to take *selective actions*:
  Act differently in response to *valid* and *invalid* input values.

Motivating Examples (2.1)

Problem: Take an integer value from the user, then output a message indicating if the number is negative, zero, or positive.

- Here is an example run of the program:

```
Enter a number:
5
You just entered a positive number.
```

- Here is another example run of the program:

```
Enter a number:
-5
You just entered a negative number.
```

- Your solution program must accommodate *all* possibilities!

Motivating Examples (2.2)

- So far, you only learned about writing programs that are executed top to bottom, line by line, without ever ***branching***.
- In general, we need a mechanism to allow the program to:
  - Check a list of *conditions*; and
  - *Branch* its execution accordingly.
- e.g., To solve the above problem, we have 3 possible branches:

1. *If* the user input is negative, then we execute the first branch that prints `You just entered a negative number.`
2. *If* the user input is zero, then we execute the second branch that prints `You just entered zero.`
3. *If* the user input is positive, then we execute the third branch that prints `You just entered a positive number.`

The `boolean` Data Type

- A (data) type denotes a set of related *runtime values*.
- We need a *data type* whose values suggest either a condition *holds*, or it *does not hold*, so that we can take selective actions.
- The Java *boolean* type consists of 2 **literal values**: *true*, *false*
- All *relational expressions* have the boolean type.

Math Symbol	Java Operator	Example (*r* is 5)	Result
≤	`<=`	`r <= 5`	*true*
≥	`>=`	`r >= 5`	*true*
=	`==`	`r == 5`	*true*
<	`<`	`r < 5`	*false*
>	`>`	`r > 5`	*false*
≠	`!=`	`r != 5`	*false*

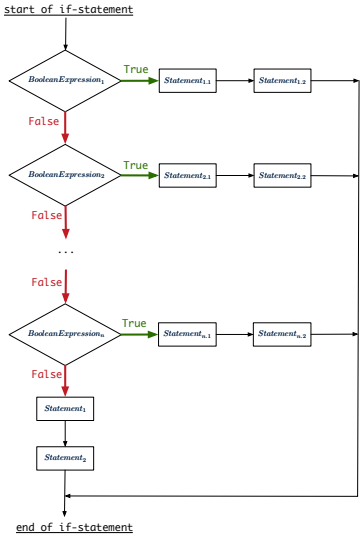
Note. You may do the following rewritings:

- `x <= y` `x > y` `x != y` `x == y`
- `!(x > y)` `!(x <= y)` `!(x == y)` `!(x != y)`

Syntax of `if` Statement

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


Semantics of `if` Statement (1.1)



Semantics of `if` Statement (1.2)

Consider a *single `if` statement* as consisting of:

- An `if` branch
- A (possibly empty) list of `else if` branches
- An optional `else` branch

At *runtime*:

- Branches of the `if` statement are *executed* from top to bottom.
- We only evaluate the **condition** of a branch if those conditions of its **preceding branches** evaluate to *false*.
- The **first** branch whose **condition** evaluates to *true* gets its body (i.e., code wrapped within { and }) *executed*.
  - After this execution, all *later* branches are *ignored*.

Semantics of `if` Statement: Case 1

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

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

```
i is negative
```

Semantics of `if` Statement: Case 2

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

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

```
i is less than 10
```


Semantics of `if` Statement: Case 3

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");
}
```

```
i is equal to 10
```


Semantics of `if` Statement: Case 4

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

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

```
i is greater than 10
```


Semantics of `if` Statement: Case 5

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");
}
```

Logical Operators

- *Logical* operators are used to create *compound* Boolean expressions.
  - Similar to *arithmetic* operators for creating compound number expressions.
  - *Logical* operators can combine Boolean expressions that are built using the *relational* operators.
    e.g., `1 <= x && x <= 10`
    e.g., `x < 1 || x > 10`
- We consider three logical operators:

Java Operator	Description	Meaning
!	logical negation	not
&&	logical conjunction	and
||	logical disjunction	or

Logical Negation

- Logical *negation* is a *unary* operator (i.e., one operand being a Boolean expression).
- The result is the "negated" value of its operand.

Operand `op`	`!op`
true	*false*
false	*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 {
  System.out.println("Area is " + radius * radius * PI);
}
```


Logical Conjunction

- Logical *conjunction* is a *binary* operator (i.e., two operands, each being a Boolean expression).
- The conjunction is *true* only when both operands are *true*.
- 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*

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


Logical Disjunction

- Logical *disjunction* is a *binary* operator (i.e., two operands, each being a Boolean expression).
- The disjunction is *false* only when both operands are *false*.
- 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 */ }
```

Logical Laws: Negation

- The *negation* of a strict inequality is a non-strict inequality.

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`

- e.g.,

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

equivalent to

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

  - Action 1 is executed when $i > j$
  - Action 2 is executed when $i <= j$.

Case Study: Error Handling of Input Radius (1)

Problem: Prompt user for radius value of a circle. Print an error message if input is negative; otherwise, print the calculated area.

```
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();
  }
}
```


Case Study: Error Handling of Input Radius (2)

The same problem can be solved by checking the *condition* of valid inputs first.

```
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();
  }
}
```


Logical Laws: DeMorgan for Conjunction

Say we have two Boolean expressions B_1 and B_2:

- What does `!(B1 && B2)` mean?
  It is **not** the case that both B_1 and B_2 are *true*.
- What does `!B1 || !B2` mean?
  It is either B_1 is *false*, B_2 is *false*, or both are *false*.
- Both expressions are equivalent! [proved by the truth table]

B_1	B_2	`!(B1 && B2)`	`!B1 || !B2`
true	true	false	false
true	false	true	true
false	true	true	true
false	false	true	true

Logical Laws: DeMorgan for Disjunction

Say we have two Boolean expressions B_1 and B_2:

- What does `!(B1 || B2)` mean?

  It is **not** the case that either B_1 is *true*, B_2 is *true*, or both are *true*.
- What does `!B1 && !B2` mean?

  Both B_1 and B_2 are *false*.
- Both expressions are equivalent! [proved by the truth table]

B_1	B_2	`!(B1 || B2)`	`!B1 && !B2`
true	*true*	*false*	*false*
true	*false*	*false*	*false*
false	*true*	*false*	*false*
false	*false*	*true*	*true*

Example: DeMorgan for Disjunction

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

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

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

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

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

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

Lesson: Be careful not to write branching conditions that use `||` but always evaluate to *true*.

Example: DeMorgan for Conjunction

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

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

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

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

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

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

Lesson: Be careful not to write branching conditions that use `&&` but always evaluate to *false*.

Operator Precedence

- Operators with *higher* precedence are evaluated before those with *lower* precedence.

  e.g., `2 + 3 * 5`
- For the three *logical operators*, negation (`!`) has the highest precedence, then conjunction (`&&`), then disjunction (`||`).

  e.g., `true || true && false` means
  - `true || (true && false)`, rather than
  - `(true || true) && false`
- When unsure, use *parentheses* to force the precedence.

Operator Associativity

- When operators with the *same precedence* are grouped together, we evaluate them from left to right.

  e.g., `1 + 2 - 3` means

  `((1 + 2) - 3)`

  e.g., `false || true || false` means

  `((false || true) || false)`

Two-Way `if` Statement without `else` Part

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

An `if` statement with the missing `else` part is equivalent to an `if` statement with an `else` part that does nothing.

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


Primitive Statement vs. Compound Statement

- A *statement* is a block of Java code that modifies value(s) of some variable(s).
- An assignment (=) statement is a *primitive statement*:

  It only modifies its left-hand-side (LHS) variable.
- An `if` statement is a *compound statement*:

  Each of its branches may modify more than one variables via other statements (e.g., assignments, `if` statements).

Compound `if` Statement: Example

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

Exercise: Draw a flow chart for the above compound statement.

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

Exercise: Draw the corresponding flow charts for both programs. Convince yourself that they are equivalent.

Multi-Way `if` Statement without `else` Part

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

In this case, since we already assign an initial, default value `"F"` to variable `letterGrade`, so when all the branch conditions evaluate to *false*, then the default value is kept.

Compare the above example with the example in slide 56.

One `if` Stmt vs. Multiple `if` Stmts (1)

Question: Do these two programs behave same at runtime?

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

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

Question: Do these two programs behave same at runtime?

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

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


One `if` Stmt vs. Multiple `if` Stmts (2)

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

```
i is >= 3
```

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

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

Two versions behave *differently* because the two conditions $i >= 3$ and $i <= 8$ *may* be satisfied simultaneously.

One `if` Stmt vs. Multiple `if` Stmts (3)

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

```
i is <= 3
```

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

```
i is <= 3
```

Two versions behave *the same* because the two conditions $i <= 3$ and $i >= 8$ *cannot* be satisfied simultaneously.

Common Error 1: Independent `if` Statements with Overlapping Conditions

```
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");
}
/* Consider marks = 84 */
```

```
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");
}
/* Consider marks = 84 */
```

- *Conditions* in a list of `if` statements are checked *independently*.
- In a single `if` statement, *only* the *first satisfying branch* is executed.

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"); }
```

- ***Yes***, because the branching conditions for the **four** if-statements are all ***non-overlapping***.
- That is, any two of these conditions ***cannot be satisfied simultaneously***:
  - `x < 0`
  - `0 <= x && x < 10`
  - `10 <= x && x < 20`
  - `x >= 20`

Overlapping Conditions: Exercise (2)

- 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"); }
```

- ***Yes***, because it's a **single** if-statement:
  Only ***the first satisfying branch*** is executed.
- But, can it be simplified?
  Hint: In a single if-statement, a branch is executed only if **all earlier branching conditions** fail.

Overlapping Conditions: Exercise (3)

- This simplified version is equivalent:

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

- At runtime, the 2nd condition `x < 10` at **L2** is checked only when the 1st condition at **L1** ***fails***
  (i.e., `!(x < 0)`, or equivalently, `x >= 0`).
- At runtime, the 3rd condition `x < 20` at **L3** is checked only when the 2nd condition at **L2** ***fails***
  (i.e., `!(x < 10)`, or equivalently, `x >= 10`).
- At runtime, the else (default) branch at **L4** is reached only when the 3rd condition at **L3** ***fails***
  (i.e., `!(x < 20)`, or equivalently, `x >= 20`).

Scope of Variables (1)

When you declare a variable, there is a limited *scope* where the variable can be used.

- If the variable is declared directly under the `main` method, then all lines of code (including branches of `if` statements) may either *re-assign* a new value to it or *use* its value.

```
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 (2.1)

- If the variable is declared under an `if` branch, an `else if` branch, or an `else` branch, then only lines of code appearing within that branch (i.e., its body) may either *re-assign* a new value to it or *use* its value.

```
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 (2.2)

- A variable declared under an `if` branch, an `else if` branch, or an `else` branch, cannot be *re-assigned* or *used* outside its scope.

```
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 (2.3)

- A variable declared under an `if` branch, `else if` branch, or `else` branch, cannot be *re-assigned* or *used* outside its scope.

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

  - A variable ***cannot*** be referred to outside its declared scope. [e.g., illegal use of `j` at **L11**]
  - A variable ***can*** be used:
    - within its declared scope [e.g., use of `i` at **L11**]
    - within sub-scopes of its declared scope [e.g., use of `i` at **L4**, **L8**]

Scope of Variables (2.4)

How about **input** parameters and **return** value?

```
1  public class SumApp {
2    public static void main(String[] args) {
3      Scanner input = new Scanner(System.in);
4      int i = input.nextInt();
5      int j = input.nextInt();
6      int k = Utilities.getSum(i, j);
7      System.out.println(k);
8    } }
```

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

  - Scope of `i`, `j`, `k`? [`SumApp.main`]
  - Scope of `x`, `y`, `result`? [`Utilities.getSum`]
  - **L5** is as if we wrote: `int k = result;` where `result` stores the value computed by executing `getSum`

General vs. Specific Boolean Conditions (1)

Two or more conditions *overlap* if they can evaluate to *true* simultaneously.

e.g., Say `marks` is declared as an integer variable:

- `marks >= 80` and `marks >= 70` overlap. [why?]
  - Values 80, 81, 82, ... make both conditions *true*
  - `marks >= 80` has **fewer** satisfying values than `marks >= 70`
  - We say `marks >= 80` is more *specific* than `marks >= 70`
  - Or, we say `marks >= 70` is more *general* than `marks >= 80`
- `marks <= 65` and `marks <= 75` overlap. [why?]
  - Values 65, 64, 63, ... make both conditions *true*
  - `marks <= 65` has **fewer** satisfying values than `marks <= 75`
  - We say `marks <= 65` is more *specific* than `marks <= 75`
  - Or, we say `marks <= 75` is more *general* than `marks <= 65`

General vs. Specific Boolean Conditions (2)

Say we have two overlapping conditions `x >= 5` and `x >= 0`:

- What values make both conditions *true*? [5, 6, 7, ...]
- Which condition is more *general*? [`x >= 0`]
- If we have a single if statement, then having this order

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

  is different from having this order

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

- Say *x* is 5, then we have
  - What output from the first program? [`x >= 5`]
  - What output from the second program? [`x >= 0`, not *specific* enough!]
- The cause of the "*not-specific-enough*" problem of the second program is that we did not check the more *specific* condition (`x >= 5`) before checking the more *general* condition (`x >= 0`).

Common Error 2: `if-elseif` Statement with Most General Condition First (1)

```
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" ;
}
```

The above program will:

- Not award a “High Distinction” to *gpa* == 4.8.
- Why?

Common Error 2: `if-elseif` Statement with Most General Condition First (2)

- Always *“sort”* the branching conditions s.t. the more *specific* conditions are checked before the more *general* conditions.

```
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";
}
else { graduateWith = "Fail"; }
```


Short-Circuit Evaluation (1)

- Both *Logical operators* && and || evaluate from left to right.
- Operator && continues to evaluate only when operands so far evaluate to *true*.

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

- Operator || continues to evaluate only when operands so far evaluate to *false*.

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


Short-Circuit Evaluation (2)

- Both *Logical operators* && and || evaluate from left to right.
- Short-Circuit Evaluation is not exploited: crash when `x == 0`

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

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

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

Common Error 3: Missing Braces (1)

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);
```

In the above code, it is as if we wrote:

```
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);
}
```


Common Error 3: Missing Braces (2)

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

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

This program will *mistakenly* print “Area is 0.0” when a *negative* number is input by the user, why? Fix?

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


Common Error 4: Misplaced Semicolon

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

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

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

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


Common Error 5: Variable Not Properly Re-Assigned

```
1 String graduateWith = "";
2 if (gpa >= 4.5) {
3   graduateWith = "High Distinction" ; }
4 else if (gpa >= 4) {
5   graduateWith = "Distinction"; }
6 else if (gpa >= 3.5) {
7   graduateWith = "Credit"; }
8 else if (gpa >= 2.5) {
9   graduateWith = "Pass"; }
```

The above program will award “” to *gpa* == 1.5. Why?
Possible Fix 1: Change the *initial value* in Line 1 to “Fail”.
Possible Fix 2: Add an *else* branch after Line 9:

```
else { graduateWith = "fail" }
```

Compare this example with the example in slide 34.

Common Errors 6: Ambiguous `else` (1)

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

- When *x* is 20, this program considers it as negative. Why?
  ∵ `else` clause matches the *most recent* unmatched `if` clause.
  ∴ The above is as if we wrote:

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


Common Errors 6: Ambiguous `else` (2)

- Fix?

  Use pairs of curly braces ({}) to force what you really mean to specify!

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


Common Pitfall 1: Updating Boolean Variable

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

Correct, but *simplifiable*: `boolean isEven = (number%2 == 0);`
Similarly, how would you simply the following?

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

Simplify `isEven == false` to `!isEven`

Beyond this lecture. . .

- Create a *console tester* in Eclipse.
  Try out the examples given in the slides.
- Solve the motivating example in **Slide 5**.
- *Optional (but recommended)*: **Videos 10 – 17** from W19 Java tutorial:

  https://www.eecs.yorku.ca/~jackie/teaching/tutorials/index.html#java_from_scratch

Index (1)

Learning Outcomes
Extra Practice?
Motivating Examples (1.1)
Motivating Examples (1.2)
Motivating Examples (2.1)
Motivating Examples (2.2)
The `boolean` Data Type
Syntax of `if` Statement
Semantics of `if` Statement (1.1)
Semantics of `if` Statement (1.2)
Semantics of `if` Statement: Case 1

Index (2)

Semantics of `if` Statement: Case 2
Semantics of `if` Statement: Case 3
Semantics of `if` Statement: Case 4
Semantics of `if` Statement: Case 5
Logical Operators
Logical Operators: Negation
Logical Operators: Conjunction
Logical Operators: Disjunction
Logical Laws: Negation
Case Study: Error Handing of Input Radius (1)
Case Study: Error Handing of Input Radius (2)

Index (3)

Logical Laws: DeMorgan for Conjunction
Logical Laws: DeMorgan for Disjunction
Example: DeMorgan for Conjunction
Example: DeMorgan for Disjunction
Operator Precedence
Operator Associativity
Two-Way `if` Statement without `else` Part
Primitive Statement vs. Compound Statement
Compound `if` Statement: Example
Multi-Way `if` Statement with `else` Part
Multi-Way `if` Statement without `else` Part

Index (4)

One `if` Stmt vs. Multiple `if` Stmts (1)
One `if` Stmt vs. Multiple `if` Stmts (2)
One `if` Stmt vs. Multiple `if` Stmts (3)
Common Error 1: Independent `if` Statements with Overlapping Conditions
Overlapping Conditions: Exercise (1)
Overlapping Conditions: Exercise (2)
Overlapping Conditions: Exercise (3)
Scope of Variables (1)
Scope of Variables (2.1)
Scope of Variables (2.2)

Index (5)

Scope of Variables (2.3)
Scope of Variables (2.4)
General vs. Specific Boolean Conditions (1)
General vs. Specific Boolean Conditions (2)
Common Error 2: `if-elseif` Statement with Most General Condition First (1)
Common Error 2: `if-elseif` Statement with Most General Condition First (2)
Short-Circuit Evaluation (1)
Short-Circuit Evaluation (2)
Common Error 3: Missing Braces (1)

Index (6)

Common Error 3: Missing Braces (2)
Common Error 4: Misplaced Semicolon
Common Error 5: Variable Not Properly Re-Assigned
Common Error 6: Ambiguous `else` (1)
Common Error 6: Ambiguous `else` (2)
Common Pitfall 1: Updating Boolean Variable
Beyond this lecture...

Loops

EECS1022 Sections M & N:
Programming for Mobile Computing
Winter 2021

CHEN-WEI WANG

Assumption

- It is assumed that you already completed the ***Week 4 Java Tutorials***:
  https://www.youtube.com/playlist?list=PL5dxAmCmjv4vEdqdvaQH4LppQvGstofS
  which introduces the syntax of `for`-loops and `while`-loops.

Learning Outcomes

Understand about *Loops*:

- Motivation: *Repetition* of *similar actions*
- Two common loops: `for` and `while`
- Primitive vs. Compound Statements
- *Nesting* loops within `if` statements
- *Nesting* `if` statements within loops
- Common Errors and Pitfalls

Motivation of Loops

- We may want to *repeat* the *similar action(s)* for a (bounded) number of times.

  e.g., Print the "Hello World" message for 100 times

  e.g., To find out the maximum value in a list of numbers
- We may want to *repeat* the *similar action(s)* under certain circumstances.

  e.g., Keep letting users enter new input values for calculating the BMI until they enter "quit"
- *Loops* allow us to repeat similar actions either
  - *for* a specified number of times; or
  - *while* a specified condition holds *true*.

Extra Practice?

- Java project archive: `Lecture_03_Loops.zip`

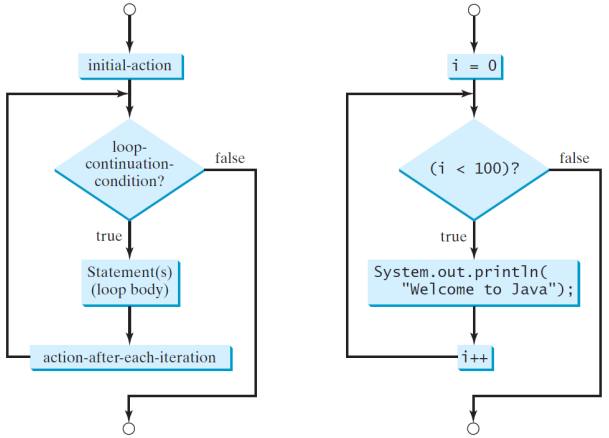
  contains some of the lecture examples.

  Expanded it: *remaining examples* and *your own examples*!
- *Optional (but recommended)*: **Videos 20 – 33** from W19 Java tutorial:

  `https://www.eecs.yorku.ca/~jackie/teaching/tutorials/index.html#java_from_scratch`

The `for` Loop (1)

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



The `for` Loop (2)

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

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*	–	–

- The number of *iterations* (i.e., 100) corresponds to the number of times the loop body is executed.
- # of times that we check the *stay condition (SC)* (i.e., 101) is # of iterations (i.e., 100) plus 1. [*True* × 100; *False* × 1]

The `for` Loop: Exercise (1)

Compare the behaviour of this program

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

and this program

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

- Are the outputs same or different?
- It is similar to asking if the two intervals

$$[0,1,2,\ldots,100) \text{ and } [1,3,5,\ldots,201)$$

  contain the same number of integers.
- *Same*, both loop bodies run exactly 100 times and do not depend on the value of *count*.

The `for` Loop (3)

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

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

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


The `for` Loop: Exercise (2)

Compare the behaviour of this program

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

and this program

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

Are the outputs same or different? *Different*, both loop body run exactly 100 times and depend on the value of *count*.

The `for` Loop: Exercise (3)

Compare the behaviour of the following three programs:

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

Output: `12345`

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

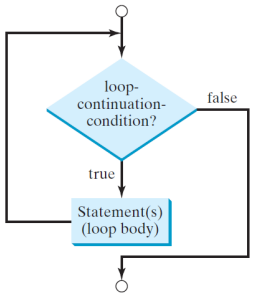
Output: `12345`

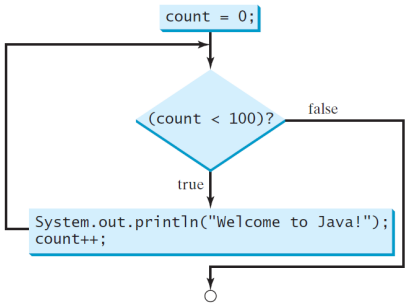
```
int i = 1;
for ( ; i <= 5 ; ) {
  i ++;
  System.out.print(i); }
```

Output: `23456`

The `while` Loop (1)

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





The `while` Loop (2)

```
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	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*	–	–

- The number of *iterations* (i.e., 100) corresponds to the number of times the loop body is executed.
- # of times that we check the *stay condition (SC)* (i.e., 101) is # of iterations (i.e., 100) plus 1. [*True* × 100; *False* × 1]

The `while` Loop: Exercise (1)

Compare the behaviour of this program

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

and this program

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

Are the outputs same or different? *Same*, both loop bodies run exactly 100 times and do not depend on the value of *count*.

The `while` Loop: Exercise (2)

Compare the behaviour of this program

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

and this program

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

Are the outputs same or different? *Different*, both loop body run exactly 100 times and depend on the value of *count*.

Primitive Statement vs. Compound Statement

- A *statement* is a block of Java code that modifies value(s) of some variable(s).
- An assignment (=) statement is a *primitive statement*: it only modifies its left-hand-side (LHS) variable.
- A `for` or `while` loop statement is a *compound statement*: the loop body may modify more than one variables via other statements (e.g., assignments, `if` statements, and `for` or `while` statements).
  - e.g., a loop statement may contain as its body `if` statements
  - e.g., a loop statement may contain as its body loop statements
  - e.g., an `if` statement may contain as its body `loop` statements

Compound Loop: Exercise (1.1)

How do you *extend* the following program

```
System.out.println("Enter a radius value:");
double radius = input.nextDouble();
double area = radius * radius * 3.14;
System.out.println("Area is " + area);
```

with the ability to *repeatedly* prompt the user for a radius value, until they explicitly enter a negative radius value to terminate the program (in which case an error message is also printed)?

```
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.");
```


Compound Loop: Exercise (1.2)

Another alternative: Use a boolean variable `isPositive`

```
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.");
```

- In **L2**: What if user enters `2`? What if user enters `-2`?
- Say in **L2** user entered `2`, then in **L8**:
  What if user enters `3`? What if user enters `-3`?
- What if `isPositive = radius >= 0` in **L9** is missing?

Compound Loop: Exercise (1.3)

Another alternative: Use a boolean variable `isNegative`

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

- In **L2**: What if user enters `2`? What if user enters `-2`?
- Say in **L2** user entered `2`, then in **L8**:
  What if user enters `3`? What if user enters `-3`?
- What if `isNegative = radius < 0` in **L9** is missing?

Converting between `for` and `while` Loops (1)

- 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.

- However, when there is not a loop counter (i.e., *i*, *count*, *etc.*) that you intend to explicitly maintain, stick to a `while` loop.

Converting between `for` and `while` Loops (2)

- 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.

- However, when there is a loop counter (i.e., *i*, *count*, *etc.*) that you intend to explicitly maintain, stick to a `for` loop.

Stay Condition (SC) vs. Exit Condition (1)

- A `for(...; SC ; ...)` loop or a `while(SC)` loop
  - *stays* to repeat its body **as long as** SC evaluates to *true*.
  - *exits* **as soon as** its SC evaluates to *false*.
- Say we have two Boolean variables:

```
boolean p, q;
```

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

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

`!(p && q)`
this is equivalent to `!p || !q`

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

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

`!(p || q)`
this is equivalent to `!p && !q`

Stay Condition (SC) vs. Exit Condition (2)

Consider the following loop:

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

- It compiles, but has a logical error. Why?
- Think about the *exit condition*:
  - `!(10 <= x || x <= 20)` [∵ *negation* of stay condition]
  - `!(10 <= x) && !(x <= 20)` [∵ law of disjunction]
  - `10 > x && x > 20` [∵ law of negation]
- `10 > x && x > 20` is equivalent to *false*, since there is no number smaller than 10 and larger than 20 at the same time.
- An exit condition being *false* means that there is no way to exit from the loop! [infinite loops are *BAD*!]

Problems, Data Structures, and Algorithms

- A *well-specified* *computational problem* precisely describes the desired *input/output relationship*.
  - **Input:** A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
  - **Output:** The maximum number max in the input array, such that $max \geq a_i$, where $1 \leq i \leq n$
  - An *instance* of the problem: $\langle 3, 1, 2, 5, 4 \rangle$
- A *data structure* is a systematic way to store and organize data in order to facilitate *access* and *modifications*.
- An *algorithm* is:
  - A solution to a well-specified *computational problem*
  - A *sequence of computational steps* that takes value(s) as *input* and produces value(s) as *output*
- Steps in an *algorithm* manipulate well-chosen *data structure(s)*.

Arrays: A Simple Data Structure

- An array is a *linear* sequence of elements.

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

- *Types* of elements in an array are the *same*.
  - an array of integers [`int[]`]
  - an array of doubles [`double[]`]
  - an array of characters [`char[]`]
  - an array of strings [`String[]`]
  - an array of booleans [`boolean[]`]
- Each element in an array is associated with an integer index.
- *Range of valid indices* of an array is constrained by its *size*.
  - The 1st element of an array has the index *0*.
  - The 2nd has index 1.
  - The i^{th} element has index $i - 1$.
  - The last element of an array has the index value that is equal to the *size of the array minus one*.

Arrays: Initialization and Indexing

- Initialize a new array object with a *fixed* size:

```
String[] names = new String[10];
```

- Alternatively, initialize a new array explicitly with its contents:

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

- Access elements in an array through indexing:

```
String first = names[0];
String last = names[names.length - 1];
```

An illegal index triggers an *ArrayInexOutOfBoundsException*.

Arrays: Iterations

- Iterate through an array using a *for-loop*:

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

- Iterate through an array using a *while-loop*:

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


Array Iterations: Translating `for` to `while` (1)

- Use either when you intend to iterate through the entire array.
-

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

  In a `for` loop, the *initialization* and *update* of the *loop counter* *i* are specified as part of the loop header.
-

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

  In a `while` loop, the *loop counter* *i*
  - Is *initialized* outside and before the loop header
  - Is *updated* at the end of the loop body

Array Iterations: Translating `for` to `while` (2)

- In both the `for` and `while` loops:
  - The stay/continuation conditions are *identical*.
  - The loop counter *i* is initialized only *once* before first entrance.
  - In each iteration, the loop counter *i* is executed *at the end* of the loop body.

Problem: Calculating Average

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

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

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

Q: What's the printout when the array is empty (e.g., `int[] numbers = {};`)?

A: Division by zero (i.e., `numbers.length` is 0). Fix?

Problem: Conditional Printing

Given an integer array:

```
int[] a = {2, 1, 3, 4, -4, 10}
```

How do you print out positive numbers only?
Hint: Use a `for` loop to *iterate* over the array. In the loop body, *conditionally* print out positive numbers only.

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

Exercise: Write the equivalent using a `while` loop.

Problem: Printing Comma-Separated Lists

Problem: Given an array `names` of strings, how do you print its contents separated by commas and ended with a period?

e.g., Given array {"*Alan*","*Mark*","*Tom*"},
print "Names: Alan, Mark, Tom."

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


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 ]);
}
```


Problem: Finding Maximum

Problem: Given a *non-empty* integer array,

e.g., `int[] a = {2, 1, 3, 4, -4, 10}`, find its maximum.

Hint: *Iterate* over the array. In the loop body, maintain the *maximum found so far* and update it when necessary.

```
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);
```

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

A: No ∵ Contents of `a` may be all smaller than this initial value (e.g., all negatives).

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

A: Yes ∵ `a[0] > a[0]` is always *false* anyway.

Problem: Finding Maximum (Tracing)

```
1  int[] a = {2, 1, 3, 4, -4, 10}
2  int max = a[0];
3  for(int i = 0; i < a.length; i ++) {
4    if (a[i] > max) {
5      max = a[i]; } }
6  System.out.println("Maximum is " + max);
```

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

Problem: All Positive?

Problem: Given an array of numbers, determine if it contains *all* positive number. [UNIVERSAL PROPERTY]

```
1  int[] numbers = {2, 3, -1, 4, 5};
2  boolean soFarOnlyPosNums = true;
3  int i = 0;
4  while (i < numbers.length) {
5    soFarOnlyPosNums = soFarOnlyPosNums && (numbers[i] > 0);
6    i = i + 1;
7  }
8  if (soFarOnlyPosNums) { /* print a msg. */ }
9  else { /* print another msg. */ }
```

- Change **Line 5** to `soFarOnlyPosNums = numbers[i] > 0;`?
- **Hints:** Run both versions on the following three arrays:
  1. `{2, 3, 1, 4, 5, 6, 8, 9, 100}` [all positive]
  2. `{2, 3, 100, 4, 5, 6, 8, 9, -1}` [negative at the end]
  3. `{2, 3, -1, 4, 5, 6, 8, 9, 100}` [negative in the middle]

Problem: All Positive? (Tracing – V1)

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

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	*false*	*true*	YES	5	*true*
5	*false*	*false*	NO	–	–

Problem: All Positive? (Tracing – V2)

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

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	–	–

Problem: All Positive? (Extended)

Problem: Given an array of numbers, determine if it contains *all* positive number. Also, *for efficiency, exit from the loop as soon as you find a negative number*.

```
1 int[] numbers = {2, 3, -1, 4, 5};
2 boolean soFarOnlyPosNums = true;
3 int i = 0;
4 while (soFarOnlyPosNums && i < numbers.length) {
5   soFarOnlyPosNums = numbers[i] > 0;
6   i = i + 1;
7 }
8 if (soFarOnlyPosNums) { /* print a msg. */ }
9 else { /* print another msg. */ }
```


Problem: All Positive? (Tracing – V3)

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

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	–	–

Problem: All Positive? (Tracing – V4)

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

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	–	–

Problem: All Positive? (Summary)

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

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; }
```

Problem: At Least One Positive?

Problem: Given an array of numbers, determine if it contains *at least one* positive number. [EXISTENTIAL PROPERTY]

```
1  int[] numbers = {-2, -3, 1, -4, -5};
2  boolean seenSomePosNum = false;
3  int i = 0;
4  while (i < numbers.length) {
5    seenSomePosNum = seenSomePosNum || (numbers[i] > 0);
6    i = i + 1;
7  }
8  if (seenSomePosNum) { /* print a msg. */ }
9  else { /* print another msg. */ }
```

- Change **Line 5** to `seenSomePosNum = numbers[i] > 0;`?
- **Hints:** Run both versions on the following three arrays:
  1. {-2, -3, -1, -4, -5, -6, -8, -9} [all negative]
  2. {-2, -3, -100, -4, -5, -6, -8, 9} [positive at the end]
  3. {-2, -3, 1, -4, -5, -6, -8, -9} [positive in the middle]

Problem: At Least One Positive? (Tracing – V1)

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

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	–	–

Problem: At Least One Positive? (Tracing – V2)

```
1  int[] ns = {-2, -3, 1, -4, -5};
2  boolean seenSomePosNum = false;
3  int i = 0;
4  while (i < ns.length) {
5    seenSomePosNum = ns[i] > 0; /* wrong */
6    i = i + 1;
7  }
```

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	–	–

Problem: At Least One Positive? (Extended)

Problem: Given an array of numbers, determine if it contains *at least one* positive number. Also, *for efficiency, exit from the loop as soon as you find a positive number*.

```
1  int[] numbers = {-2, -3, 1, -4, -5};
2  boolean seenSomePosNum = false;
3  int i = 0;
4  while (!seenSomePosNum && i < numbers.length) {
5    seenSomePosNum = numbers[i] > 0;
6    i = i + 1;
7  }
8  if (seenSomePosNum) { /* print a msg. */ }
9  else { /* print another msg. */ }
```

Problem: At Least One Positive? (Tracing – V3)

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

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	–	–

Problem: At Least One Positive? (Tracing – V4)

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

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	–	–

Problem: At Least One 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; }
```


Problem: Array Sorted?

Problem: Given an array `a` of integers, how do you determine if it is sorted in a *non-decreasing* order?

e.g., Given $\{1,2,2,4\}$, print *true*; given $\{2,4,3,3\}$ print *false*.

```
1 boolean isSorted = true;
2 for(int i = 0; i < a.length - 1; i ++) {
3   isSorted = isSorted && (a[i] <= a[i + 1]);
4 }
```

Alternatively (with early exit):

```
1 boolean isSorted = true;
2 for(int i = 0; isSorted && i < a.length - 1; i ++) {
3   isSorted = a[i] <= a[i + 1];
4 }
```

Problem: Array Sorted? (Tracing 1 – V1)

```
1 int[] a = {1, 2, 2, 4}
2 boolean isSorted = true;
3 for(int i = 0; i < a.length - 1; i ++) {
4   isSorted = isSorted && (a[i] <= a[i + 1]);
5 }
```

i	$a[i]$	$a[i+1]$	$a[i] <= a[i+1]$	*isSorted*	exit?
0	–	–	–	*true*	N
0	1	2	*true*	*true*	N
1	2	2	*true*	*true*	N
2	2	4	*true*	*true*	**Y**

Problem: Array Sorted? (Tracing 2 – V1)

```
1 int[] a = {2, 4, 3, 3}
2 boolean isSorted = true;
3 for(int i = 0; i < a.length - 1; i ++) {
4   isSorted = isSorted && (a[i] <= a[i + 1]);
5 }
```

i	$a[i]$	$a[i+1]$	$a[i] <= a[i+1]$	*isSorted*	exit?
0	–	–	–	*true*	N
0	2	4	*true*	*true*	N
1	4	3	*false*	*false*	N
2	3	3	*true*	*false*	**Y**

Problem: Array Sorted? (Tracing – V2)

```
1 int[] a = {2, 4, 3, 3}
2 boolean isSorted = true;
3 for(int i = 0; isSorted && i < a.length - 1; i ++) {
4   isSorted = a[i] <= a[i + 1];
5 }
```

i	$a[i]$	$a[i+1]$	$a[i] <= a[i+1]$	*isSorted*	exit?
0	–	–	–	*true*	N
0	2	4	*true*	*true*	N
1	4	3	*false*	*false*	**Y**

Checking Universal Properties

- Determine if *all* elements satisfy a property.
- We need to repeatedly apply the logical *conjunction*.
- **Approach 1**: Scanning the Entire Array

```
1 boolean allPos = true;
2 for(int i = 0; i < a.length; i ++) {
3   allPos = allPos && (a[i] > 0);
4 }
```

- **Approach 2**: Exiting Early If Possible
  As soon as we find an element that *does not satisfy* a property, then we exit from the loop.

```
1 boolean allPos = true;
2 for(int i = 0; allPos && i < a.length; i ++) {
3   allPos = a[i] > 0;
4 }
```

Checking Universal Properties (Tracing)

```
1 int[] a = {2, 3, -1, 4, 5, 6, 8, 9, 100};
2 boolean allPos = true;
3 for(int i = 0; allPos && i < a.length; i ++) {
4   allPos = a[i] > 0;
5 }
```

i	*a*[*i*]	*a*[*i*] > *0*	*allPos*	exit?
0	–	–	*true*	N
0	2	*true*	*true*	N
1	3	*true*	*true*	N
2	-1	*false*	*false*	**Y**

- **Question:** Why `allPos` initialized *true* in Line 2?
- **Question:** What if we change the stay condition in Line 3 to only `i < a.length`?

  Intermediate values of `allPos` will be overwritten!

Checking Existential Properties (Tracing)

```
1 int[] a = {2, 3, -1, 4, 5, 6, 8, 9, 100};
2 boolean foundNegative = false;
3 for(int i = 0; ! foundNegative && i < a.length; i ++) {
4   foundNegative = a[i] < 0;
5 }
```

i	*a*[*i*]	*a*[*i*] < *0*	*foundNegative*	!*foundNegative*	exit?
0	–	–	*false*	*true*	N
0	2	*false*	*false*	*true*	N
1	3	*false*	*false*	*true*	N
2	-1	*true*	*true*	*false*	**Y**

- **Question:** Why `foundNegative` initialized *false* in Line 2?
- **Question:** What if we change the stay condition in Line 3 to only `i < a.length`?

  Intermediate values of `foundNegative` will be overwritten!

Checking Existential Properties

- Determine if *at least one* element satisfies a property.
  e.g., Is there at lease one negative element in array `a`?
- **Approach 1**: Scanning the Entire Array

```
1 boolean foundNegative = false;
2 for(int i = 0; i < a.length; i ++) {
3   foundNegative = foundNegative || a[i] < 0;
4 }
```

- **Approach 2**: Exiting Early If Possible

  As soon as we find an element that *satisfies* a property, then we exit from the loop.

```
1 boolean foundNegative = false;
2 for(int i = 0; ! foundNegative && i < a.length; i ++) {
3   foundNegative = a[i] < 0;
4 }
```


Remarks: Checking Array Properties Universal vs. Existential

- In some cases, you *must* iterate through the *entire* array in order to obtain the result.
  e.g., max, min, total, *etc.*
- In other cases, you *exit* from the loop *as soon as* you obtain the result.
  e.g., to know if all numbers positive, it is certainly *false* *as soon as* you find the first negative number
  e.g., to know if there is at least one negative number, it is certainly *true* *as soon as* you find the first negative number

Guarding Indexing using Short Circuit (1)

Problem: Ask the user how many integers they would like to input, prompt them accordingly, then ask them for an integer index, and check if the number stored at that index is even (i.e., error if it is odd).

```
How many integers?
2
Enter an integer:
23
Enter an integer:
24
Enter an index:
1
24 at index 1 is even.
```


Guarding Indexing using Short Circuit (2)

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

- Does the above code work? [*not* always!]
  - It *works* if `0 <= i && i < ns.length`
  - It *fails* on **L10** if `i < 0 || i >= ns.length`
    [*ArrayIndexOutOfBoundException*]

Guarding Indexing using Short Circuit (3.1)

```
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] */ }
```

- Does the above code work? [*always*!]
- Short-circuit effect of *conjunction* has L-to-R evaluations:

  `ns[i] % 2 == 0` is evaluated only when the *guard* (i.e., `0 <= i && i < ns.length`) evaluates to *true*.

Guarding Indexing using Short Circuit (3.2)

```
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."); }
```

- Does the above code work? [*always*!]
- Short-circuit effect of *disjunction* has L-to-R evaluations:

  `ns[i] % 2 == 1` is evaluated only when the *guard* (i.e., `i < 0 || i >= ns.length`) evaluates to *false*.

Guarding Indexing using Short Circuit (4)

- ∵ Short-circuit evaluations go from **left** to **right**.
  ∴ **Order** in which the operands are placed matters!
- Consider the following changes to **L10**:
  - `ns[i] % 2 == 0 && 0 <= i && i < ns.length`
    What if input `i` is s.t. `i < 0`? [*crash*]
    What if input `i` is s.t. `i >= ns.length`? [*crash*]
  - `0 <= i && ns[i] % 2 == 0 && i < ns.length`
    What if input `i` is s.t. `i < 0`? [*works*]
    What if input `i` is s.t. `i >= ns.length`? [*crash*]
  - `i < ns.length && ns[i] % 2 == 0 && 0 <= i`
    What if input `i` is s.t. `i < 0`? [*crash*]
    What if input `i` is s.t. `i >= ns.length`? [*works*]
- When does each change to **L10** *work* and *crash*?
  - `ns[i] % 2 == 1 || i < 0 || i >= ns.length`
  - `i < 0 || ns[i] % 2 == 1 || i >= ns.length`
  - `i >= ns.length || ns[i] % 2 == 1 || i < 0`

Common Error (1): Improper Initialization of Loop Counter

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

The default value for an initialized boolean variable is *false*.
Fix?

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


Common Error (2): Improper Stay Condition

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

The maximum index for array `a` is `a.length - 1`
Fix?

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


Common Error (3): Improper Update to Loop Counter

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

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

The indices used to print will be: `1`, `2`, `3`, `...`, `a.length`
Fix?

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

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

Common Error (4): Improper Update of Stay Condition

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

What if the user's answer in **L1** is simply `Y`? An *infinite loop*!!

∵ **SC** never gets updated when a new answer is read. Fix?

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


Common Error (5): Improper Start Value of Loop Counter

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

The value of loop counter *i* after the first `while` loop is −1!

Fix?

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


Common Error (6): Wrong Syntax

How about this?

```
while(int i = 0; i < 10; i ++) { ... }
```

You meant:

```
for(int i = 0; i < 10; i ++) { ... }
```

How about this?

```
for(i < 10) { ... }
```

You meant:

```
while(i < 10) { ... }
```

or

```
for( ; i < 10 ; ) { ... }
```


Common Error (7): Misplaced Semicolon

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

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

Output?

```
Hello!
```

Fix?

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

Index (1)

Assumption
Learning Outcomes
Extra Practice?
Motivation of Loops
The `for` Loop (1)
The `for` Loop (2)
The `for` Loop (3)
The `for` Loop: Exercise (1)
The `for` Loop: Exercise (2)
The `for` Loop: Exercise (3)
The `while` Loop (1)

Index (2)

The `while` Loop (2)
The `while` Loop: Exercise (1)
The `while` Loop: Exercise (2)
Primitive Statement vs. Compound Statement
Compound Loop: Exercise (1.1)
Compound Loop: Exercise (1.2)
Compound Loop: Exercise (1.3)
Converting between `for` and `while` Loops (1)
Converting between `for` and `while` Loops (2)
Stay Condition (SC) vs. Exit Condition (1)
Stay Condition (SC) vs. Exit Condition (2)

Index (3)

Problems, Data Structures, and Algorithms
Arrays: A Simple Data Structure
Arrays: Initialization and Indexing
Arrays: Iterations
Array Iterations: Translating `for` to `while` (1)
Array Iterations: Translating `for` to `while` (2)
Problem: Calculating Average
Problem: Conditional Printing
Problem: Printing Comma-Separated Lists
Problem: Printing Backwards
Problem: Finding Maximum

Index (4)

Problem: Finding Maximum (Tracing)
Problem: All Positive?
Problem: All Positive? (Tracing – V1)
Problem: All Positive? (Tracing – V2)
Problem: All Positive? (Extended)
Problem: All Positive? (Tracing – V3)
Problem: All Positive? (Tracing – V4)
Problem: All Positive? (Summary)
Problem: At Least One Positive?
Problem: At Least One Positive? (Tracing – V1)
Problem: At Least One Positive? (Tracing – V2)

Index (5)

Problem: At Least One Positive? (Extended)
Problem: At Least One Positive? (Tracing – V3)
Problem: At Least One Positive? (Tracing – V4)
Problem: At Least One Positive? (Summary)
Problem: Array Sorted?
Problem: Array Sorted? (Tracing 1 – V1)
Problem: Array Sorted? (Tracing 2 – V1)
Problem: Array Sorted? (Tracing – V2)
Checking Universal Properties
Checking Universal Properties (Tracing)
Checking Existential Properties

Index (6)

Checking Existential Properties (Tracing)
Remarks: Checking Array Properties
Universal vs. Existential
Guarding Indexing using Short Circuit (1)
Guarding Indexing using Short Circuit (2)
Guarding Indexing using Short Circuit (3.1)
Guarding Indexing using Short Circuit (3.2)
Guarding Indexing using Short Circuit (4)
Common Error (1):
Improper Initialization of Loop Counter
Common Error (2):
Improper Stay Condition

Index (7)

Common Error (3):
Improper Update to Loop Counter
Common Error (4):
Improper Update of Stay Condition
Common Error (5):
Improper Start Value of Loop Counter
Common Error (6): Wrong Syntax
Common Error (7): Misplaced Semicolon

Classes and Objects

YORK UNIVERSITÉ UNIVERSITY

EECS1022 Sections M & N:
Programming for Mobile Computing
Winter 2021

CHEN-WEI WANG

Assumptions

It is assumed that you also complete:

- **Java Tutorial Videos**:
  - ***Weeks 6*** [link]
  - ***Weeks 7*** [link]
  - ***Weeks 8*** [link]
- **Written Notes**:
  - ***Inferring Classes from JUnit Tests*** [link]
  - ***Manipulating Multi-Valued, Reference-Typed Attributes*** [link]

Learning Outcomes

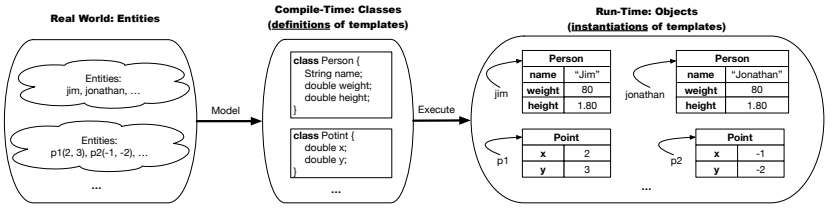
Understand:

- Object Orientation
- Classes as Templates:
  - attributes, constructors, (accessor and mutator) methods
  - use of `this`
- Objects as Instances:
  - use of `new`
  - the dot notation, method invocations
  - reference aliasing
- Reference-Typed Attributes: Single-Valued vs. Multi-Valued
- Non-Static vs. Static Variables
- Helper Methods

Where are we? Where will we go?

- We have developed the Java code within a `main` or utility method.
- In Java:
  - We may define more than one *classes*
  - Each class may contain more than one *methods*
- *object-oriented programming* in Java:
  - Use *classes* to define templates
  - Use *objects* to instantiate classes
  - At *runtime*, *create* objects and *call* methods on objects, to *simulate interactions* between real-life entities.

Object Orientation: Observe, Model, and Execute



- Study this tutorial video that walks you through the idea of *object orientation*.
- We *observe* how real-world *entities* behave.
- We *model* the common *attributes* and *behaviour* of a set of entities in a single *class*.
- We *execute* the program by creating *instances* of classes, which interact in a way analogous to that of real-world *entities*.

Object-Oriented Programming (OOP)

- In real life, lots of *entities* exist and interact with each other.
  e.g., *People* gain/lose weight, marry/divorce, or get older.
  e.g., *Cars* move from one point to another.
  e.g., *Clients* initiate transactions with banks.
- Entities:
  - Possess *attributes*;
  - Exhibit *bebaviour*; and
  - Interact with each other.
- Goals: Solve problems *programmatically* by
  - *Classifying* entities of interest
    Entities in the same class share *common* attributes and bebaviour.
  - *Manipulating* data that represent these entities
    Each entity is represented by *specific* values.

OO Thinking: Templates vs. Instances (1.1)

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.

- A template called `Point` defines the common
  - *attributes* (e.g., `x`, `y`) [≈ nouns]
  - *behaviour* (e.g., move up, get distance from) [≈ verbs]

OO Thinking: Templates vs. Instances (1.2)

- 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: $[\sqrt{(-4)^2 + (-2)^2}]$

OO Thinking: Templates vs. Instances (2.1)

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

- A template called `Person` defines the common
  - *attributes* (e.g., `age`, `weight`, `height`) [≈ nouns]
  - *behaviour* (e.g., get older, gain weight) [≈ verbs]

OO Thinking: Templates vs. Instances (2.2)

- A *template* (e.g., class `Person`) defines what's **shared** by a set of related entities (i.e., persons).
  - Common *attributes* (`age`, `weight`, `height`)
  - Common *behaviour* (get older, lose weight, grow taller)
- Each template may be *instantiated* as multiple instances, each with *instance-specific* values for attributes `age`, `weight`, and `height`.
  - `Person` instance `jim` is 50-years old, 1.8-meters tall and 80-kg heavy
  - `Person` instance `jonathan` is 65-years old, 1.73-meters tall and 90-kg heavy
- Instances of the same template may exhibit *distinct behaviour*.
  - When `jim` gets older, he becomes 51
  - When `jonathan` gets older, he becomes 66.
  - `jim`'s BMI is based on his own height and weight $[\frac{80}{1.8^2}]$
  - `jonathan`'s BMI is based on his own height and weight $[\frac{90}{1.73^2}]$

OOP: Classes ≈ Templates

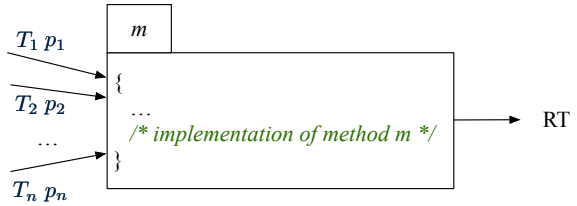
In Java, you use a *class* to define a *template* that enumerates *attributes* that are common to a set of *entities* of interest.

```
public class Person {
  private int age;
  private String nationality;
  private double weight;
  private double height;
}
```

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


OOP: Methods (1.1)

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



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

OOP: Methods (1.2)

- In the body of the method, you may
  - Declare new *local variables* (whose *scope* is within that method).
  - Use or change values of *attributes*.
  - Use values of *parameters*, if any.

```
public class Person {
  private String nationality;
  public void changeNationality(String newNationality) {
    nationality = newNationality; } }
```

- *Call* a *method*, with a *context object*, by passing *arguments*.

```
public class PersonTester {
  public static void main(String[] args) {
    Person jim = new Person(50, "British");
    Person jonathan = new Person(60, "Canadian");
    jim.changeNationality("Korean");
    jonathan.changeNationality("Korean"); } }
```

OOP: Methods (2)

- Each *class* `C` defines a list of methods.
  - A *method* `m` is a named block of code.
- We *reuse* the code of method `m` by calling it on an *object* `obj` of class `C`.
  For each *method call* `obj.m(...)`:
  - `obj` is the ***context object*** of type `C`
  - `m` is a method defined in class `C`
  - We intend to apply the *code effect of method* `m` to object `obj`.
    e.g., `jim.getOlder()` vs. `jonathan.getOlder()`
    e.g., `p1.moveUp(3)` vs. `p2.moveUp(3)`
- All objects of class `C` share *the same definition* of method `m`.
- However:
  ∵ Each object may have *distinct attribute values*.
  ∴ Applying *the same definition* of method `m` has *distinct effects*.

OOP: Methods (3)

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: Class Constructors (1.1)

- The purpose of defining a *class* is to be able to create *instances* out of it.
- To *instantiate* a class, we use one of its *constructors*.
- A constructor
  - declares input *parameters*
  - uses input parameters to *initialize* *some or all* of its *attributes*

OOP: Class Constructors (1.2)

For each *class*, you may define *one or more* *constructors*:

- *Names* of all constructors must match the class name.
- *No return types* need to be specified for constructors.
- Constructor must have *distinct* lists of *parameter types*.
  - `Person(String n), Person(String n, int age)` ✓
  - `Person(String n, int age), Person(int age, String n)` ✓
  - `Person(String fN, int age), Person(String lN, int id)` ×
- Each *parameter* that is used to initialize an attribute must have a *matching type*.
- The *body* of each constructor specifies how *some or all* *attributes* may be *initialized*.

OOP: Class Constructors (2.1)

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

  public Point(double initX, double initY) {
    x = initX;
    y = initY;
  }

  public Point(char axis, double distance) {
    if (axis == 'x') { x = distance; }
    else if (axis == 'y') { y = distance; }
    else { /* Error: invalid axis */ }
  }
}
```


OOP: Class Constructors (2.2)

```
public class Person {
  private int age;
  private String nationality;
  private double weight;
  private double height;
  public Person(int initAge, String initNat) {
    age = initAge;
    nationality = initNat;
  }
  public Person (double initW, double initH) {
    weight = initW;
    height = initH;
  }
  public Person(int initAge, String initNat,
          double initW, double initH) {
    ... /* initialize all attributes using the parameters */
  }
}
```


Visualizing Objects at Runtime (1)

- To trace a program with sophisticated manipulations of objects, it's critical for you to visualize how objects are:
  - Created using *constructors*

    ```
    Person jim = new Person(50, "British", 80, 1.8);
    ```
  - Inquired using *accessor methods*

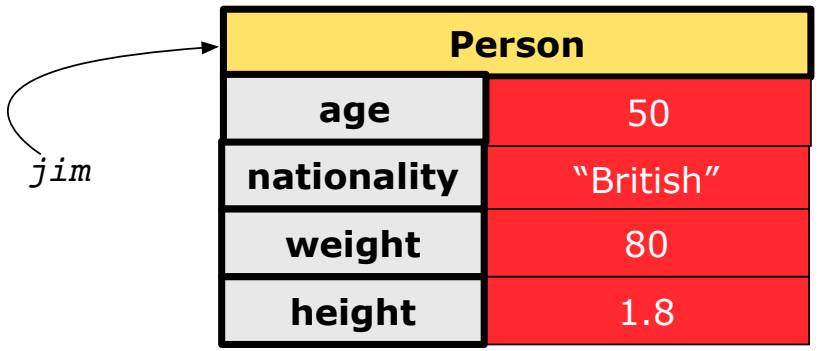
    ```
    double bmi = jim.getBMI();
    ```
  - Modified using *mutator methods*

    ```
    jim.gainWeightBy(10);
    ```
- To visualize an object:
  - Draw a rectangle box to represent *contents* of that object:
    - Title indicates the *name of class* from which the object is instantiated.
    - Left column enumerates *names of attributes* of the instantiated class.
    - Right column fills in *values* of the corresponding attributes.
  - Draw arrow(s) for *variable(s)* that store the object's *address*.

Visualizing Objects at Runtime (2.1)

After calling a *constructor* to create an object:

```
Person jim = new Person(50, "British", 80, 1.8);
```

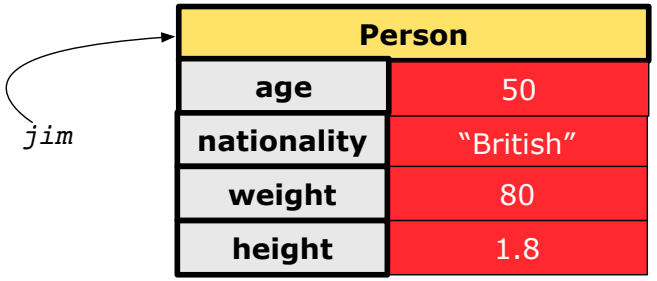


Visualizing Objects at Runtime (2.2)

After calling an *accessor* to inquire about context object `jim`:

```
double bmi = jim.getBMI();
```

- Contents of the object pointed to by `jim` remain intact.
- Retuned value $\frac{80}{(1.8)^2}$ of `jim.getBMI()` stored in variable `bmi`.

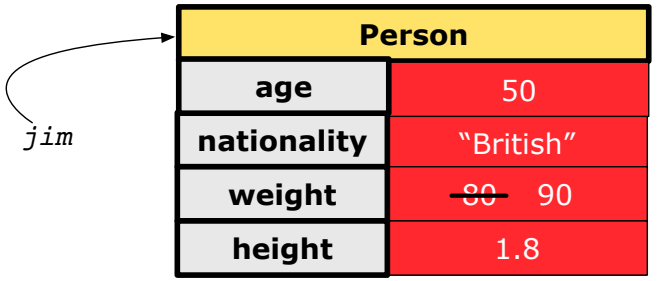


Visualizing Objects at Runtime (2.3)

After calling a *mutator* to modify the state of context object `jim`:

```
jim.gainWeightBy(10);
```

- *Contents* of the object pointed to by `jim` change.
- *Address* of the object remains unchanged.
  ⇒ `jim` points to the same object!

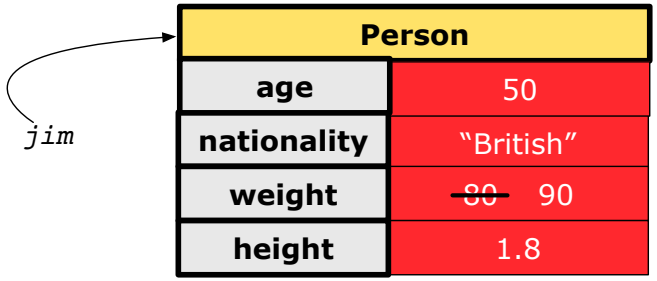


Visualizing Objects at Runtime (2.4)

After calling the same *accessor* to inquire the *modified* state of context object `jim`:

```
bmi = jim.getBMI();
```

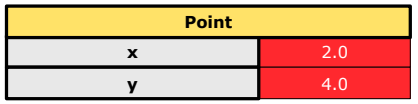
- Contents of the object pointed to by `jim` remain intact.
- Retuned value $\frac{90}{(1.8)^2}$ of `jim.getBMI()` stored in variable `bmi`.



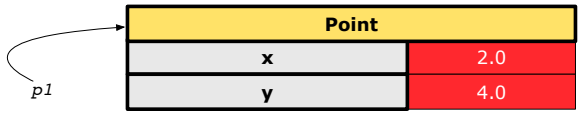
Object Creation (1.1)

```
Point p1 = new Point(2, 4);
```

1. **RHS (Source) of Assignment**: `new Point(2, 4)` creates a new *Point object* in memory.



2. **LHS (Target) of Assignment**: `Point p1` declares a *variable* that is meant to store the *address* of *some Point object*.
3. **Assignment**: Executing `=` stores new object's address in `p1`.



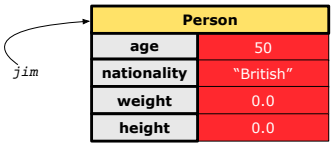
Object Creation (1.2)

```
Person jim = new Person(50, "British");
```

1. **RHS (Source) of Assignment**: `new Person(50, "British")` creates a new *Person object* in memory.

Person	
age	50
nationality	"British"
weight	0.0
height	0.0

2. **LHS (Target) of Assignment**: `Point jim` declares a *variable* that is meant to store the *address* of *some Person object*.
3. **Assignment**: Executing `=` stores new object's address in `jim`.



Object Creation (2)

```
Point p1 = new Point(2, 4);
System.out.println(p1);
```

```
Point@677327b6
```

By default, the address stored in `p1` gets printed.

Instead, print out attributes separately:

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

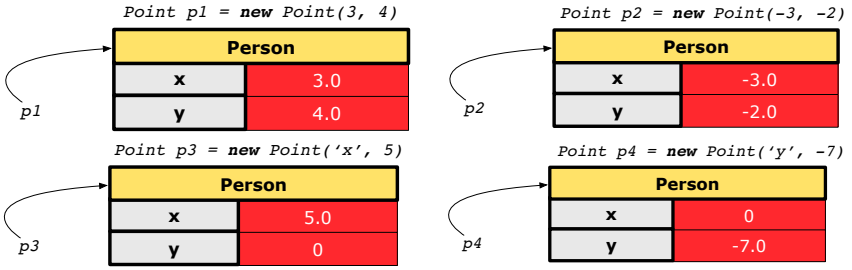
```
(2.0, 4.0)
```


OOP: Object Creation (3.1.1)

A constructor may only *initialize* some attributes and leave others *uninitialized*.

```
public class PointTester {
  public static void main(String[] args) {
    Point p1 = new Point(3, 4);
    Point p2 = new Point(-3 -2);
    Point p3 = new Point('x', 5);
    Point p4 = new Point('y', -7);
  }
}
```


OOP: Object Creation (3.1.2)

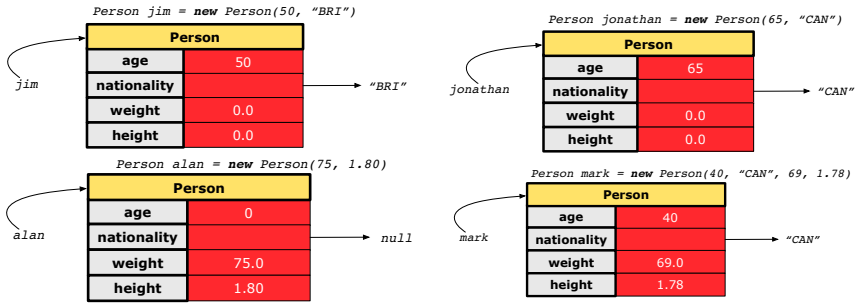


OOP: Object Creation (3.2.1)

A constructor may only *initialize* some attributes and leave others *uninitialized*.

```
public class PersonTester {
  public static void main(String[] args) {
    /* initialize age and nationality only */
    Person jim = new Person(50, "BRI");
    /* initialize age and nationality only */
    Person jonathan = new Person(65, "CAN");
    /* initialize weight and height only */
    Person alan = new Person(75, 1.80);
    /* initialize all attributes of a person */
    Person mark = new Person(40, "CAN", 69, 1.78);
  }
}
```


OOP: Object Creation (3.2.2)



OOP: Object Creation (4)

- When using the constructor, pass *valid* *argument values*:
  - The type of each argument value must match the corresponding parameter type.
  - e.g., `Person(50, "BRI")` matches `Person(int initAge, String initNationality)`
  - e.g., `Point(3, 4)` matches `Point(double initX, double initY)`
- When creating an instance, *uninitialized* attributes implicitly get assigned the *default values*.
  - Set *uninitialized* attributes properly later using **mutator** methods

```
Person jim = new Person(50, "British");
jim.setWeight(85);
jim.setHeight(1.81);
```


OOP: The Dot Notation (1)

- A binary operator:
  - **LHS** an object
  - **RHS** an attribute or a method
- Given a *variable* of some *reference type* that is **not** `null`:
  - We use a dot to retrieve any of its *attributes*.
    Analogous to 's in English
    e.g., `jim.nationality` means `jim`'s nationality
  - We use a dot to invoke any of its *mutator methods*, in order to *change* values of its attributes.
    e.g., `jim.changeNationality("CAN")` changes the `nationality` attribute of `jim`
  - We use a dot to invoke any of its *accessor methods*, in order to *use* the result of some computation on its attribute values.
    e.g., `jim.getBMI()` computes and returns the BMI calculated based on `jim`'s weight and height
  - Return value of an *accessor method* must be stored in a variable.
    e.g., `double jimBMI = jim.getBMI()`

The `this` Reference (1)

- Each *class* may be instantiated to multiple *objects* at runtime.

```
public class Point {
  private double x; private double y;
  public void moveUp(double units) { y += units; }
}
```

- Each time when we call a method of some class, using the dot notation, there is a specific *target*/*context* object.

```
1 Point p1 = new Point(2, 3);
2 Point p2 = new Point(4, 6);
3 p1.moveUp(3.5);
4 p2.moveUp(4.7);
```

  - `p1` and `p2` are called the *call targets* or *context objects*.
  - **Lines 3 and 4** apply the same definition of the `moveUp` method.
  - But how does Java distinguish the change to `p1.y` versus the change to `p2.y`?

The `this` Reference (2)

- In the *method* definition, each *attribute* has an *implicit* `this` which refers to the *context object* in a call to that method.

```
public class Point {
  private double x;
  private double y;
  public Point(double newX, double newY) {
    this.x = newX;
    this.y = newY;
  }
  public void moveUp(double units) {
    this.y = this.y + units;
  }
}
```

- Each time when the *class* definition is used to create a new `Point` *object*, the `this` reference is substituted by the name of the new object.

The `this` Reference (3)

- After we create `p1` as an instance of `Point`

```
Point p1 = new Point(2, 3);
```

- When invoking `p1.moveUp(3.5)`, a version of `moveUp` that is specific to `p1` will be used:

```
public class Point {
  private double x;
  private double y;
  public Point(double newX, double newY) {
    p1.x = newX;
    p1.y = newY;
  }
  public void moveUp(double units) {
    p1.y = p1.y + units;
  }
}
```


The `this` Reference (4)

- After we create `p2` as an instance of `Point`

```
Point p2 = new Point(4, 6);
```

- When invoking `p2.moveUp(4.7)`, a version of `moveUp` that is specific to `p2` will be used:

```
public class Point {
  private double x;
  private double y;
  public Point(double newX, double newY) {
    p2.x = newX;
    p2.y = newY;
  }
  public void moveUp(double units) {
    p2.y = p2.y + units;
  }
}
```

The `this` Reference (5)

The `this` reference can be used to *disambiguate* when the names of *input parameters* clash with the names of *class attributes*.

```
public class Point {
  private double x;
  private double y;
  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }
  public void setX(double x) {
    this.x = x;
  }
  public void setY(double y) {
    this.y = y;
  }
}
```


The `this` Reference (6.1): Common Error

The following code fragment compiles but is problematic:

```
1  public class Person {
2    private String name;
3    private int age;
4    public Person(String name, int age) {
5      name = name;
6      age = age;
7    }
8    public void setAge(int age) {
9      age = age;
10   }
11 }
```

- Why? [variable **shadowing**]
  Target (LHS) of the assignment (L5) refers to parameter `name` (L4).
- Fix?

The `this` Reference (6.2): Common Error

Always remember to use `this` when *input parameter* names clash with *class attribute* names.

```
public class Person {
  private String name;
  private int age;
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
  public void setAge(int age) {
    this.age = age;
  }
}
```


OOP: Mutator Methods

- These methods *change* values of attributes.
- We call such methods *mutators* (with `void` return type).

```
public class Person {
  ...
  public void gainWeight(double units) {
    this.weight = this.weight + units;
  }
}
```

```
public class Point {
  ...
  public void moveUp() {
    this.y = this.y + 1;
  }
}
```

OOP: Accessor Methods

- These methods *return* the result of computation based on attribute values.
- We call such methods *accessors* (with non-`void` return type).

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

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


OOP: Use of Mutator vs. Accessor Methods

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

OOP: Method Calls

```
1 Point p1 = new Point(3, 4);
2 Point p2 = new Point(-4, -3);
3 System.out.println(p1.getDistanceFromOrigin());
4 System.out.println(p2.getDistanceFromOrigin());
5 p1.moveUp(1);
6 p2.moveUp(1);
7 System.out.println(p1.getDistanceFromOrigin());
8 System.out.println(p2.getDistanceFromOrigin());
```

- **Lines 1 and 2** create two different instances of `Point`
- **Lines 3 and 4:** invoking the same accessor method on two different instances returns *distinct* values
- **Lines 5 and 6:** invoking the same mutator method on two different instances results in *independent* changes
- **Lines 3 and 7:** invoking the same accessor method on the same instance *may* return *distinct* values, why? **Line 5**

OOP: 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)`

OOP: Reference Aliasing (1)

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

- **Line 2** copies the number stored in `i` to `j`.
- After **Line 4**, `i`, `j`, `k` refer to three separate integer placeholder, which happen to store the same value `3`.

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

- **Line 2** copies the *address* stored in `p1` to `p2`.
- Both `p1` and `p2` refer to the same object in memory!
- `p3`, whose *contents* are same as `p1` and `p2`, refer to a different object in memory.

OOP: Reference Aliasing (2.1)

Problem: Consider assignments to *primitive* variables:

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


OOP: Reference Aliasing (2.2)

Problem: Consider assignments to *reference* variables:

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


Java Data Types (1)

A (data) type denotes a set of related *runtime values*.

1. *Primitive Types*
   - *Integer* Type
     - `int` [set of 32-bit integers]
     - `long` [set of 64-bit integers]
   - *Floating-Point Number* Type
     - `double` [set of 64-bit FP numbers]
   - *Character* Type
     - `char` [set of single characters]
   - *Boolean* Type
     - `boolean` [set of `true` and `false`]
2. *Reference Type*: *Complex Type with Attributes and Methods*
   - *String* [set of references to character sequences]
   - *Person* [set of references to Person objects]
   - *Point* [set of references to Point objects]
   - *Scanner* [set of references to Scanner objects]

Java Data Types (2)

- A variable that is declared with a *type* but *uninitialized* is implicitly assigned with its *default value*.
  - **Primitive Type**
    - `int i;` [*0* is implicitly assigned to `i`]
    - `double d;` [*0.0* is implicitly assigned to `d`]
    - `boolean b;` [*false* is implicitly assigned to `b`]
  - **Reference Type**
    - `String s;` [*null* is implicitly assigned to `s`]
    - `Person jim;` [*null* is implicitly assigned to `jim`]
    - `Point p1;` [*null* is implicitly assigned to `p1`]
    - `Scanner input;` [*null* is implicitly assigned to `input`]
- You *can* use a primitive variable that is *uninitialized*. Make sure the *default value* is what you want!
- Calling a method on a *uninitialized* reference variable crashes your program. [*NullPointerException*] Always initialize reference variables!

Java Data Types (3.1)

- An *attribute* may store the reference to another object.

```
public class Person { private Person spouse; }
```

- Methods may take as *parameters* references to other objects.

```
public class Person {
  public void marry(Person other) { ... } }
```

- *Return values* from methods may be references to objects.

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


Java Data Types (3.2.1)

An attribute may be ***multi***-valued, ***reference***-typed e.g., of type `Point[]`, storing references to `Point` objects.

```
1  public class PointCollector {
2    private Point[] points; private int nop;/* number of points */
3    public PointCollector() { this.points = new Point[100]; }
4    public void addPoint(double x, double y) {
5      this.points[this.nop] = new Point(x, y); this.nop++; }
6    public Point[] getPointsInQuadrantI() {
7      Point[] ps = new Point[this.nop];
8      int count = 0; /* number of points in Quadrant I */
9      for(int i = 0; i < this.nop; i++) {
10       Point p = this.points[i];
11       if(p.x > 0 && p.y > 0) { ps[count] = p; count++; } }
12     Point[] q1Points = new Point[count];
13     /* ps contains null if count < nop */
14     for(int i = 0; i < count; i++) { q1Points[i] = ps[i] }
15     return q1Points;
16   } }
```

Required Reading: Point and PointCollector

Java Data Types (3.2.2)

```
1  public class PointCollectorTester {
2    public static void main(String[] args) {
3      PointCollector pc = new PointCollector();
4      System.out.println(pc.getNumberOfPoints()); /* 0 */
5      pc.addPoint(3, 4);
6      System.out.println(pc.getNumberOfPoints()); /* 1 */
7      pc.addPoint(-3, 4);
8      System.out.println(pc.getNumberOfPoints()); /* 2 */
9      pc.addPoint(-3, -4);
10     System.out.println(pc.getNumberOfPoints()); /* 3 */
11     pc.addPoint(3, -4);
12     System.out.println(pc.getNumberOfPoints()); /* 4 */
13     Point[] ps = pc.getPointsInQuadrantI();
14     System.out.println(ps.length);  /* 1 */
15     System.out.println("(" +
16       ps[0].getX() + ", " + ps[0].getY() + ")"); /* (3, 4) */
17   }
18 }
```

Java Data Types (3.3.1)

An attribute may be of type `ArrayList<Point>`, storing references to `Point` objects.

```
1  class PointCollector {
2    ArrayList<Point> points;
3    PointCollector() { points = new ArrayList<>(); }
4    void addPoint(Point p) {
5      points.add (p); }
6    void addPoint(double x, double y) {
7      points.add (new Point(x, y)); }
8    ArrayList<Point> getPointsInQuadrantI() {
9      ArrayList<Point> q1Points = new ArrayList<>();
10     for(int i = 0; i < points.size(); i ++) {
11       Point p = points.get(i);
12       if(p.x > 0 && p.y > 0) { q1Points.add (p); } }
13     return q1Points;
14   } }
```

L8 & L9 may be replaced by:

```
for(Point p : points) { q1Points.add(p); }
```


Java Data Types (3.3.2)

```
1  class PointCollectorTester {
2    public static void main(String[] args) {
3      PointCollector pc = new PointCollector();
4      System.out.println(pc.points.size());  /* 0 */
5      pc.addPoint(3, 4);
6      System.out.println(pc.points.size());  /* 1 */
7      pc.addPoint(-3, 4);
8      System.out.println(pc.points.size());  /* 2 */
9      pc.addPoint(-3, -4);
10     System.out.println(pc.points.size());  /* 3 */
11     pc.addPoint(3, -4);
12     System.out.println(pc.points.size());  /* 4 */
13     ArrayList<Point> ps = pc.getPointsInQuadrantI();
14     System.out.println(ps.length);  /* 1 */
15     System.out.println("(" + ps[0].x + ", " + ps[0].y + ")");
16     /* (3, 4) */
17   }
18 }
```


Anonymous Objects (1)

- What's the difference between these two fragments of code?

```
1  double square(double x) {
2    double sqr = x * x;
3    return sqr; }
```

```
1  double square(double x) {
2    return x * x; }
```

After **L2**, the result of `x * x`:

- LHS: it can be reused (without recalculating) via the name `sqr`.
- RHS: it is not stored anywhere and returned right away.

- Same principles applies to objects:

```
1  Person getP(String n) {
2    Person p = new Person(n);
3    return p; }
```

```
1  Person getP(String n) {
2    return new Person(n); }
```

***new** Person(n)* is an object whose address is not stored in a variable.

- LHS: **L2** stores the address of this anonymous object in `p`.
- RHS: **L2** returns the address of this anonymous object directly.

Anonymous Objects (2.1)

Anonymous objects can also be used as *assignment sources* or *argument values*:

```
class Member {
  private Order[] orders;
  private int noo;
  /* constructor ommitted */
  public void addOrder(Order o) {
    this.orders[this.noo] = o;
    this.noo++;
  }
  public void addOrder(String n, double p, double q) {
    this.addOrder( new Order(n, p, q) );
    /* Equivalent implementation:
     * this.orders[this.noo] = new Order(n, p, q); noo ++;
     */
  }
}
```

Anonymous Objects (2.2)

One more example on using anonymous objects:

```
public class MemberTester {
  public static void main(String[] args) {
    Member m = new Member("Alan");
    Order o = new Order("Americano", 4.7, 3);
    m.addOrder(o);
    m.addOrder( new Order("Cafe Latte", 5.1, 4) );
  }
}
```


The `this` Reference (7.1): Exercise

Consider the `Person` class

```
public class Person {
  private String name;
  private Person spouse;
  public Person(String name) {
    this.name = name;
  }
}
```

How do you implement a mutator method `marry` which marries the current `Person` object to an input `Person` object?

The `this` Reference (7.2): Exercise

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

When we call `jim.marry(elsa)`: `this` is substituted by the ***context object*** `jim`, and `other` by the ***argument*** `elsa`.

```
public void marry(Person ~~other~~ elsa) {
  ...
    jim.spouse = elsa;
    elsa.spouse = jim;
  ...
}
```


OOP: The Dot Notation (2)

- **LHS** of dot *can be more complicated than a variable*:
  - It can be a *path* that brings you to an object

```
public class Person {
  private String name;/* public accessor: name() */
  private Person spouse;/* public accessor: spouse() */
}
```

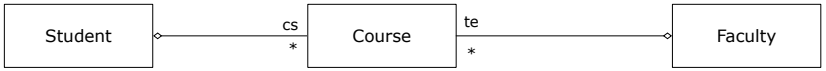
  - Say we have `Person jim = new Person("Jim Davies")`
  - Inquire about `jim`'s name? `[jim.name()]`
  - Inquire about `jim`'s spouse's name? `[jim.spouse().name()]`
  - But what if `jim` is single (i.e., `jim.spouse() == null`)? Calling `jim.spouse().name()` will cause *NullPointerException*!!
  - **Quesion.** Assuming that:
    - `jim` is not single. `[jim.spouse() != null]`
    - The marriage is mutual. `[jim.spouse().spouse() != null]`

    What does `jim.spouse().spouse().name()` mean?

    Answer. `jim.name()`

OOP: The Dot Notation (3.1)

In real life, the relationships among classes are sophisticated.




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

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

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

- Assume: All attributs are ***private*** with the corresponding ***public*** accessor methods.
- In the context of class `Student`:
  - Writing ***`cs`*** denotes the array of registered courses.
  - Writing ***`cs[i]`*** (where `i` is a valid index) navigates to the class `Course`, which changes the context to class `Course`.

OOP: The Dot Notation (3.2)

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

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

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

```
class Student {
  ... /* attributes */
  /* Get the student's id */
  String getID() { return this.id; }
  /* Get the title of the ith course */
  String getTitle(int i) {
    return this.cs[i].getTitle();
  }
  /* Get the instructor's name of the ith course */
  String getName(int i) {
    return this.cs[i].getProf.getName();
  }
}
```


OOP: The Dot Notation (3.3)

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

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

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

```
class Course {
  ... /* attributes */
  /* Get the course's title */
  String getTitle() { return this.title; }
  /* Get the instructor's name */
  String getName() {
    return this.prof.getName();
  }
  /* Get title of ith teaching course of the instructor */
  String getTitle(int i) {
    return this.prof.getTE()[i].getTitle();
  }
}
```


OOP: The Dot Notation (3.4)

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

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

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

```
class Faculty {
  ... /* attributes */
  /* Get the instructor's name */
  String getName() {
    return this.name;
  }
  /* Get the title of ith teaching course */
  String getTitle(int i) {
    return this.te[i].getTitle();
  }
}
```

OOP: Equality (1)

```
Point p1 = new Point(2, 3);
Point p2 = new Point(2, 3);
boolean sameLoc = ( p1 == p2 );
System.out.println("p1 and p2 same location?" + sameLoc);
```

```
p1 and p2 same location?  false
```


OOP: Equality (2)

- Recall that
  - A *primitive* variable stores a primitive *value*
    e.g., `double d1 = 7.5; double d2 = 7.5;`
  - A *reference* variable stores the *address* to some object (rather than storing the object itself)
    e.g., `Point p1 = new Point(2, 3)` assigns to `p1` the address of the new `Point` object
    e.g., `Point p2 = new Point(2, 3)` assigns to `p2` the address of *another* new `Point` object
- The binary operator `==` may be applied to compare:
  - *Primitive* variables: their *contents* are compared
    e.g., `d1 == d2` evaluates to *true*
  - *Reference* variables: the *addresses* they store are compared (**rather than** comparing contents of the objects they refer to)
    e.g., `p1 == p2` evaluates to *false* because `p1` and `p2` are addresses of *different* objects, even if their contents are *identical*.

Static Variables (1)

```
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;
  }
}
```

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

But, managing the unique id's *manually* is *error-prone*!

Static Variables (2)

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

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

- Each instance of a class (e.g., `acc1`, `acc2`) has a *local* copy of each attribute or instance variable (e.g., `id`).
  - Changing `acc1.id` does not affect `acc2.id`.
- A ***static*** variable (e.g., `globalCounter`) belongs to the class.
  - All instances of the class share a *single* copy of the ***static*** variable.
  - Change to `globalCounter` via `acc1` is also visible to `acc2`.

Static Variables (3)

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

- *Static* variable `globalCounter` is not instance-specific like *instance* variable (i.e., attribute) `id` is.
- To access a *static* variable:
  - ***No*** context object is needed.
  - Use of the class name suffices, e.g., `Account.globalCounter`.
- Each time `Account`'s constructor is called to create a new instance, the increment effect is *visible to all existing objects* of `Account`.

Static Variables (4.1): Common Error

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

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


Static Variables (4.2): Common Error

- Attribute `numberOfAccounts` should **not** be declared as `static` as its value should be specific to the client object.
- If it were declared as `static`, then every time the `addAccount` method is called, although on different objects, the increment effect of `numberOfAccounts` will be visible to all `Client` objects.
- Here is the correct version:

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


Static Variables (5.1): Common Error

```
1 public class Bank {
2    private string branchName;
3    public String getBrachName() { return this.branchName; }
4    private static int nextAccountNumber = 0;
5    public static String getInfo() {
6       nextAccountNumber++;
7       return this.branchName + nextAccountNumber;
8    }
9 }
```

- *Non-static method cannot be referenced from a static context*
- **Line 4** declares that we *can* call the method `getInfo` without instantiating an object of the class `Bank`.
- However, in **Line 7**, the *static* method references a *non-static* attribute, for which we *must* instantiate a `Bank` object.

Static Variables (5.2): Common Error

```
1 public class Bank {
2     private String branchName;
3     public String getBrachName() { return this.branchName; }
4     private static int nextAccountNumber = 0;
5     public static String getInfo() {
6         nextAccountNumber++;
7         return this.branchName + nextAccountNumber;
8     }
9 }
```

- To call `getInfo()`, no instances of `Bank` are required:

```
Bank.getInfo();
```

- *Contradictorily*, to access `branchName`, a *context object* is required:

```
Bank b = new Bank(); b.setBranch("Songdo IBK");
System.out.println(b.getBranchName());
```


Static Variables (5.3): Common Error

There are two possible ways to fix:

1. Remove all uses of *non-static* variables (i.e., `branchName`) in the *static* method (i.e., `useAccountNumber`).
2. Declare `branchName` as a *static* variable.
   - This does not make sense.
     ∵ `branchName` should be a value specific to each `Bank` instance.

OOP: Helper Methods (1)

- After you complete and test your program, feeling confident that it is *correct*, you may find that there are lots of *repetitions*.
- When similar fragments of code appear in your program, we say that your code "*smells*"!
- We may eliminate *repetitions* of your code by:
  - *Factoring out* recurring code fragments into a new method.
  - This new method is called a *helper method*:
    - You can replace every occurrence of the recurring code fragment by a *call* to this helper method, with appropriate argument values.
    - That is, we *reuse* the body implementation, rather than repeating it over and over again, of this helper method via calls to it.
- This process is called *refactoring* of your code:
  Modify the code structure **without** compromising *correctness*.

OOP: Helper (Accessor) Methods (2.1)

```
public class PersonCollector {
  private Person[] ps;
  private final int MAX = 100;/* max # of persons to store */
  private int nop; /* number of persons */
  public PersonCollector() {
    this.ps = new Person[MAX];
  }
  public void addPerson(Person p) {
    this.ps[this.nop] = p;
    this.nop++;
  }
  /* Tasks:
   * 1. An accessor: boolean personExists(String n)
   * 2. A mutator: void changeWeightOf(String n, double w)
   * 3. A mutator: void changeHeightOf(String n, double h)
   */
}
```

OOP: Helper (Accessor) Methods (2.2.1)

```
public class PersonCollector {
  /* ps, MAX, nop, PersonCollector(), addPerson */
  public boolean personExists(String n) {
    boolean found = false;
    for(int i = 0; i < nop; i ++) {
      if(ps[i].getName().equals(n)) { found = true; } }
    return found;
  }
  public void changeWeightOf(String n, double w) {
    for(int i = 0; i < nop; i ++) {
      if(ps[i].getName().equals(n)) { ps[i].setWeight(w); } }
  }
  public void changeHeightOf(String n, double h) {
    for(int i = 0; i < nop; i ++) {
      if(ps[i].getName().equals(n)) { ps[i].setHeight(h); } }
  }
}
```


OOP: Helper (Accessor) Methods (2.2.2)

```
public class PersonCollector {/* code smells:repetitions! */
  /* ps, MAX, nop, PersonCollector(), addPerson */
  public boolean personExists( String n ) {
    boolean found = false;
    for(int i = 0; i < nop; i ++) {
      if(ps[i].getName().equals(n)) { found = true; } }
    return found;
  }
  public void changeWeightOf( String n, double w) {
    for(int i = 0; i < nop; i ++) {
      if(ps[i].getName().equals(n)) { ps[i].setWeight(w);} }
  }
  public void changeHeightOf( String n, double h) {
    for(int i = 0; i < nop; i ++) {
      if(ps[i].getName().equals(n)) { ps[i].setHeight(h);} }
  }
}
```


OOP: Helper (Accessor) Methods (2.3)

```
public class PersonCollector { /* Code Smell Eliminated */
  /* ps, MAX, nop, PersonCollector(), addPerson */
  private int indexOf (String n) { /* Helper Methods */
    int i = -1;
    for(int j = 0; j < nop; j ++) {
      if(ps[j].getName().equals(n)) { i = j; }
    }
    return i; /* -1 if not found; >= 0 if found. */
  }
  public boolean personExists(String n) {
    return this.indexOf (n) >= 0; }
  public void changeWeightOf(String n, double w) {
    int i = indexOf (n); if(i >= 0) { ps[i].setWeight(w); }
  }
  public void changeHeightOf(String n, double h) {
    int i = indexOf (n); if(i >= 0) { ps[i].setHeight(h); }
  }
}
```


OOP: Helper (Accessor) Methods (3.1)

Problems:

- A `Point` class with `x` and `y` coordinate values.
- Accessor `double getDistanceFromOrigin()`.
  `p.getDistanceFromOrigin()` returns the distance between `p` and `(0, 0)`.
- Accessor `double getDistancesTo(Point p1, Point p2)`.
  `p.getDistancesTo(p1, p2)` returns the sum of distances between `p` and `p1`, and between `p` and `p2`.
- Accessor `double getTriDistances(Point p1, Point p2)`.
  `p.getDistancesTo(p1, p2)` returns the sum of distances between `p` and `p1`, between `p` and `p2`, and between `p1` and `p2`.

OOP: Helper (Accessor) Methods (3.2)

```
class Point { /* code smells:repetitions! */
  double x; double y;

  double getDistanceFromOrigin() {
    return Math.sqrt(Math.pow(this.x - 0, 2) + Math.pow(this.y - 0, 2));

  double getDistancesTo(Point p1, Point p2) {
    return
      Math.sqrt(Math.pow(this.x - p1.x, 2) + Math.pow(y - p1.y, 2))
      +
      Math.sqrt(Math.pow(this.x - p2.x, 2) + Math.pow(y - p2.y, 2)); }

  double getTriDistances(Point p1, Point p2) {
    return
      Math.sqrt(Math.pow(this.x - p1.x, 2) + Math.pow(y - p1.y, 2))
      +
      Math.sqrt(Math.pow(this.x - p2.x, 2) + Math.pow(y - p2.y, 2))
      +
      Math.sqrt(Math.pow(p1.x - p2.x, 2) + Math.pow(p1.y - p2.y, 2));
  }
}
```


OOP: Helper (Accessor) Methods (3.3)

- The code pattern

  `Math.sqrt(Math.pow(... - ..., 2) + Math.pow(... - ..., 2))`

  is written down explicitly every time we need to use it.
- Create a *helper method* out of it, with the right *parameter* and *return* types:

```
double getDistanceFrom(double otherX, double otherY) {
   return Math.sqrt(
     Math.pow(ohterX - this.x, 2)
     +
     Math.pow(otherY - this.y, 2));
}
```


OOP: Helper (Accessor) Methods (3.4)

```
public class Point { /* Code Smell Eliminated */
  private double x; private double y;
  double getDistanceFrom(double otherX, double otherY) {
    return Math.sqrt(Math.pow(ohterX - this.x, 2) +
           Math.pow(otherY - this.y, 2));
  }
  double getDistanceFromOrigin() {
    return this.getDistanceFrom(0, 0);
  }
  double getDistancesTo(Point p1, Point p2) {
    return this.getDistanceFrom(p1.x, p1.y) +
           this.getDistanceFrom(p2.x, p2.y);
  }
  double getTriDistances(Point p1, Point p2) {
    return this.getDistanceFrom(p1.x, p1.y) +
           this.getDistanceFrom(p2.x, p2.y) +
           p1.getDistanceFrom(p2.x, p2.y)
  }
}
```


OOP: Helper (Mutator) Methods (4.1)

```
public class Student {
  private String name;
  private double balance;
  public Student(String n, double b) {
    name = n;
    balance = b;
  }

  /* Tasks:
   * 1. A mutator void receiveScholarship(double val)
   * 2. A mutator void payLibraryOverdue(double val)
   * 3. A mutator void payCafeCoupons(double val)
   * 4. A mutator void transfer(Student other, double val)
   */
}
```

OOP: Helper (Mutator) Methods (4.2.1)

```
public class Student {
  /* name, balance, Student(String n, double b) */
  public void receiveScholarship(double val) {
    balance = balance + val;
  }
  public void payLibraryOverdue(double val) {
    balance = balance - val;
  }
  public void payCafeCoupons(double val) {
    balance = balance - val;
  }
  public void transfer(Student other, double val) {
    balance = balance - val;
    other.balance = other.balance + val;
  }
}
```


OOP: Helper (Mutator) Methods (4.2.2)

```
public class Student { /* code smells:repetitions! */
  /* name, balance, Student(String n, double b) */
  public void receiveScholarship(double val) {
    balance = balance + val;
  }
  public void payLibraryOverdue(double val) {
    balance = balance - val;
  }
  public void payCafeCoupons(double val) {
    balance = balance - val;
  }
  public void transfer(Student other, double val) {
    balance = balance - val;
    balance = other.balance + val;
  }
}
```


OOP: Helper (Mutator) Methods (4.3)

```
public class Student { /* Code Smell Eliminated */
  /* name, balance, Student(String n, double b) */
  public void deposit(double val) { /* Helper Method */
    balance = balance + val;
  }
  public void withdraw(double val) { /* Helper Method */
    balance = balance - val;
  }
  public void receiveScholarship(double val) { this.deposit(val); }
  public void payLibraryOverdue(double val) { this.withdraw(val); }
  public void payCafeCoupons(double val) { this.withdraw(val) }
  public void transfer(Student other, double val) {
    this.withdraw(val);
    other.deposit(val);
  }
}
```


Index (1)

Assumptions

Learning Outcomes

Where are we? Where will we go?

Object Orientation:
Observe, Model, and Execute

Object-Oriented Programming (OOP)

OO Thinking: Templates vs. Instances (1.1)

OO Thinking: Templates vs. Instances (1.2)

OO Thinking: Templates vs. Instances (2.1)

OO Thinking: Templates vs. Instances (2.2)

OOP: Classes ≈ Templates

Index (2)

OOP: Methods (1.1)
OOP: Methods (1.2)
OOP: Methods (2)
OOP: Methods (3)
OOP: Class Constructors (1.1)
OOP: Class Constructors (1.2)
OOP: Class Constructors (2.1)
OOP: Class Constructors (2.2)
Visualizing Objects at Runtime (1)
Visualizing Objects at Runtime (2.1)
Visualizing Objects at Runtime (2.2)

Index (3)

Visualizing Objects at Runtime (2.3)
Visualizing Objects at Runtime (2.4)
Object Creation (1.1)
Object Creation (1.2)
Object Creation (2)
OOP: Object Creation (3.1.1)
OOP: Object Creation (3.1.2)
OOP: Object Creation (3.2.1)
OOP: Object Creation (3.2.2)
OOP: Object Creation (4)
OOP: The Dot Notation (1)

Index (4)

The `this` Reference (1)
The `this` Reference (2)
The `this` Reference (3)
The `this` Reference (4)
The `this` Reference (5)
The `this` Reference (6.1): Common Error
The `this` Reference (6.2): Common Error
OOP: Mutator Methods
OOP: Accessor Methods
OOP: Method Calls
OOP: Use of Mutator vs. Accessor Methods

Index (5)

OOP: Method Parameters
OOP: Reference Aliasing (1)
OOP: Reference Aliasing (2.1)
OOP: Reference Aliasing (2.2)
Java Data Types (1)
Java Data Types (2)
Java Data Types (3.1)
Java Data Types (3.2.1)
Java Data Types (3.2.2)
Java Data Types (3.3.1)
Java Data Types (3.3.2)

Index (6)

Anonymous Objects (1)
Anonymous Objects (2.1)
Anonymous Objects (2.2)
The `this` Reference (7.1): Exercise
The `this` Reference (7.2): Exercise
OOP: The Dot Notation (2)
OOP: The Dot Notation (3.1)
OOP: The Dot Notation (3.2)
OOP: The Dot Notation (3.3)
OOP: The Dot Notation (3.4)
OOP: Equality (1)

Index (7)

OOP: Equality (2)
Static Variables (1)
Static Variables (2)
Static Variables (3)
Static Variables (4.1): Common Error
Static Variables (4.2): Common Error
Static Variables (5.1): Common Error
Static Variables (5.2): Common Error
Static Variables (5.3): Common Error
OOP: Helper Methods (1)
OOP: Helper (Accessor) Methods (2.1)

Index (8)

OOP: Helper (Accessor) Methods (2.2.1)
OOP: Helper (Accessor) Methods (2.2.2)
OOP: Helper (Accessor) Methods (2.3)
OOP: Helper (Accessor) Methods (3.1)
OOP: Helper (Accessor) Methods (3.2)
OOP: Helper (Accessor) Methods (3.3)
OOP: Helper (Accessor) Methods (3.4)
OOP: Helper (Mutator) Methods (4.1)
OOP: Helper (Mutator) Methods (4.2.1)
OOP: Helper (Mutator) Methods (4.2.2)
OOP: Helper (Mutator) Methods (4.3)

Two-Dimensional Arrays

EECS1022 Sections M & N:
Programming for Mobile Computing
Winter 2021

CHEN-WEI WANG

Learning Outcomes

Understand:

- Nested loops
- Two-Dimensional Arrays: Why?
- Two-Dimensional Arrays: Syntax and Semantics
- Two-Dimensional Arrays: Examples

Assumptions

It is assumed that you also complete:

- **Java Tutorial Videos**:
  - ***Weeks 10*** [ink]
  - ***Weeks 11*** [ink]

Sequential Loops vs. Nested Loops

- *Sequential Loops*:
  Each loop completes an *independent* phase of work.
  e.g., Print an array from left to right, then right to left.

```
System.out.println("Left to right:");
for(int i = 0; i < a.length; i ++) {
  System.out.println(a[i]); }
System.out.println("Right to left:");
for(int i = 0; i < a.length; i ++) {
  System.out.println(a[a.length - i - 1]); }
```

- *Nested Loops*:
  Loop counters form *all combinations* of indices.

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


Nested Loops: Finding Duplicates (1)

- Given an integer array `a`, determine if it contains any duplicates.
  e.g., Print *false* for $\{1,2,3,4\}$. Print *true* for $\{1,4,2,4\}$.
- **Hint:** When can you conclude that there are duplicates?
  As soon as we find that two elements at difference indices happen to be the same

```
1 boolean hasDup = false;
2 for(int i = 0; i < a.length; i ++) {
3   for(int j = 0; j < a.length; j ++) {
4     hasDup = hasDup || (i != j && a[i] == a[j]);
5   } /* end inner for */ } /* end outer for */
6 System.out.println(hasDup);
```

- **Question:** How do you modify the code, so that we exit from the loops *as soon as* the array is found containing duplicates?
  - **L2**: `for(...; !hasDup && i < a.length; ...)`
  - **L3**: `for(...; !hasDup && j < a.length; ...)`
  - **L4**: `hasDup = (i != j && a[i] == a[j]);`

Nested Loops: Finding Duplicates (2)

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

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

Nested Loops: Finding Duplicates (3)

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

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

Nested Loops: Finding Duplicates (4)

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

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

Nested Loops: Finding Duplicates (5)

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

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

Nested Loops: Finding Duplicates (6)

The previous two versions scan all pairs of array slots, but with redundancy: e.g., `a[0] == a[2]` and `a[2] == a[0]`.

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

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*

Nested Loops: Finding Duplicates (7)

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

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

2-D Arrays: Motivating Example (1)

Consider a table of distances between seven cities:

	Chicago	Boston	New York	Atlanta	Miami	Dallas	Houston
Chicago	0	983	787	714	**1375**	967	1087
Boston	**983**	0	214	1102	1763	1723	1842
New York	787	214	0	888	1549	1548	1627
Atlanta	714	1102	888	0	661	781	810
Miami	1375	1763	1549	661	0	1426	**1187**
Dallas	967	1723	1548	781	1426	0	239
Houston	1087	1842	1627	810	1187	239	0

As part of the program for an airline reservation system, the *distance of a trip* with *multiple stop-overs* is to be calculated in order to accumulate the milage of frequent flyers.

e.g., A trip {`Boston, Chicago, Miami, Houston`} takes 983 (B-to-C) + 1375 (C-to-M) + 1187 (M-to-H) = 3545 miles

Question: How do you manipulate such information in Java?

2-D Arrays: Motivating Example (2.1)

Here is a solution based on what we've learnt so far:

- Fix the "positions" of cities in the table as constants:

```
final int CHICAGO = 0;
final int BOSTON = 1;
final int MIAMI = 4;
```

- Represent each (horizontal) row using a one-dimensional array:

```
int[] fromChicago = {0, 983, 787, 714, 1375, 967, 1087}
int[] fromBoston = {983, 0, 214, 1102, 1763, 1723, 1842}
int[] fromMiami = {1375, 1763, 1549, 661, 0, 1426, 1187}
```

- Given an itinerary {`Boston, Chicago, Miami, Houston`}, choose the corresponding arrays in the right order:

```
int[] dist = fromBoston[CHICAGO]
           + fromChicago[MIAMI]
           + fromMiami[HUSTON];
```

2-D Arrays: Motivating Example (2.2)

What if cities of an itinerary are read from the user?

```
1  Scanner input = new Scanner(System.in);
2  System.out.println("How many cities?");
3  int howMany = input.nextInt(); input.nextLine();
4  String[] trip = new String[howMany];
5  /* Read cities in the trip from the user. */
6  for(int i = 0; i < howMany; i ++) {
7    System.out.println("Enter a city:");
8    trip[i] = input.nextLine();
9  }
10 /* Add up source-to-destination distances. */
11 int dist = 0;
12 for(int i = 0; i < howMany - 1; i ++) {
13   String src = trip[i];
14   String dst = trip[i + 1];
15   /* How to accumulate the distance between src and dst?  */
16 }
```


2-D Arrays: Motivating Example (2.3)

Given a source and a destination, we need to *explicitly* select:

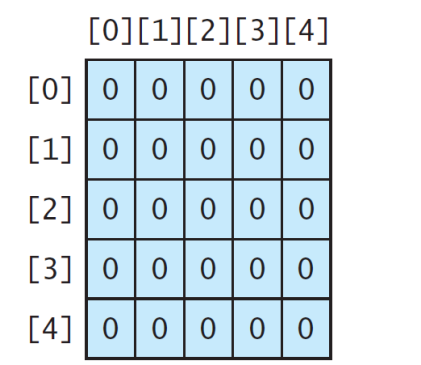
- The corresponding *source row* [e.g., `fromBoston`]
- The corresponding *destination index* [e.g., `CHICAGO`]

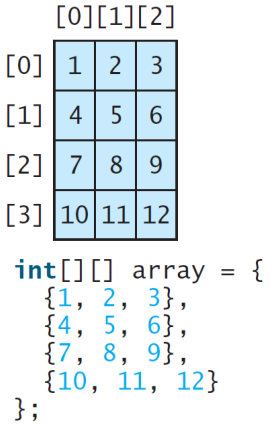
```
13 String src = trip[i];
14 String dst = trip[i + 1];
15 if(src.equals("Chicago")) {
16   if(dst.equals("Boston")) {dist += fromChicago[BOSTON];}
17   else if(dst.equals("New York")) {dist += fromChicago[NY];}
18   ...
19 }
20 else if(src.equals("Boston")) {
21   if(dst.equals("Chicago")) {dist += fromBoston[CHICAGO];}
22   else if(dst.equals("NEW YORK")) {dist += fromBoston[NY];}
23   ...
24 }
25 ...
```

- Drawback? $7 \times (7 - 1)$ possibilities to program!

2-D Arrays: Initialization (1)

A *2D array* is really *an array of arrays*





2-D Arrays: Initialization (1)

A *2D array* may be initialized either at the time of declaration, or after declaration.

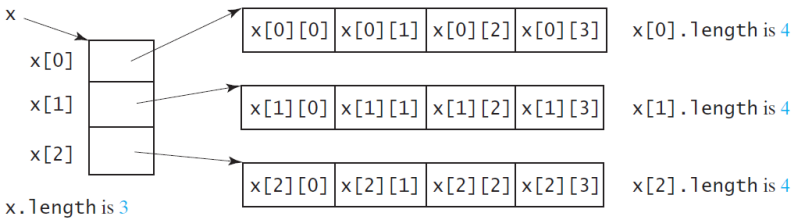
```
int[][] array = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9},
  {10, 11, 12}
};
```

Same as

```
int[][] array = new int[4][3];
array[0][0] = 1; array[0][1] = 2; array[0][2] = 3;
array[1][0] = 4; array[1][1] = 5; array[1][2] = 6;
array[2][0] = 7; array[2][1] = 8; array[2][2] = 9;
array[3][0] = 10; array[3][1] = 11; array[3][2] = 12;
```

2-D Arrays: Lengths (1)

For a *2D array*, you may query about its *size*, or *sizes* of its component arrays.

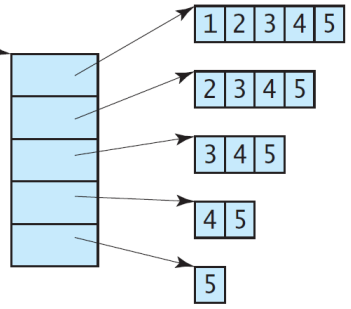


2-D Arrays: Lengths (2)

For a *2D array*, its components may have different *sizes*.

e.g.,

```
int[][] triangleArray = {
  {1, 2, 3, 4, 5},
  {2, 3, 4, 5},
  {3, 4, 5},
  {4, 5},
  {5}
};
```



2-D Arrays: Assignments

For a *2D array*, access a slot via its *row* and *column*.

e.g.,

	[0]	[1]	[2]	[3]	[4]
[0]	0	0	0	0	0
[1]	0	0	0	0	0
[2]	0	7	0	0	0
[3]	0	0	0	0	0
[4]	0	0	0	0	0

```
matrix[2][1] = 7;
```


Revisiting the Motivating Example

```
double[][] distances = {
  {0, 983, 787, 714, 1375, 967, 1087},
  {983, 0, 214, 1102, 1763, 1723, 1842},
  {787, 214, 0, 888, 1549, 1548, 1627},
  {714, 1102, 888, 0, 661, 781, 810},
  {1375, 1763, 1549, 661, 0, 1426, 1187},
  {967, 1723, 1548, 781, 1426, 0, 239},
  {1087, 1842, 1627, 810, 1187, 239, 0},
};

final int CHICAGO = 0;
final int BOSTON = 1;
...
final int HOUSTON = 6;

int MiamiToBoston = distances[MIAMI][BOSTON];
int BostonToNewYork = distances[BOSTON][NEWYORK];
int MiamiToNewYork = MiamiToBoston + BostonToNewYork;
```

Two-Dimensional Arrays: Example (1)

Problem: Given a 2D array `a` of integers, print out all its values: first row, second row, third row, and so on.

```
1 for(int row = 0; row < a.length; row ++) {
2   System.out.print("Row" + row);
3   for(int col = 0; col < a[row].length; col ++) {
4     System.out.print(a[row][col]);
5   }
6   System.out.println(); }
```

- In **L1**, we write *a.length* so that it will print out exactly that many rows in the matrix.
- In **L3**, we write *a[row].length* so that it will print out according to how large the row `a[row]` is.

Two-Dimensional Arrays: Example (2)

Problem: Given a 2D array `a` of integers, calculate the average of its values.

```
int total = 0;
int numOfElements = 0;
for(int row = 0; row < a.length; row ++) {
  for(int col = 0; col < a[row].length; col ++) {
    total += a[row][col];
    numOfElements ++;
  }
}
double average = ((double) total) / numOfElements;
System.out.println("Average is " + average);
```

- Why is the `numOfElements` counter necessary?
- Divide `total` by `a.length * a[0].length` instead?

Two-Dimensional Arrays: Example (3)

Problem: Given a 2D array `a` of integers, find out its *maximum* and *minimum* values.

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


Two-Dimensional Arrays: Example (4)

Problem: Given a 2D array `a` of integers, find out the row which has the *maximum* sum.

```
1  int maxRow = 0; int maxSum = 0;
2  for(int col=0; col < a[0].length; col ++){maxSum += a[0][col];}
3  for(int row = 1; row < a.length; row ++) {
4    int sum = 0;
5    for(int col = 0; col < a[row].length; col ++) {
6      sum += a[row][col];
7    }
8    if (sum > maxSum) {
9      maxRow = row;
10     maxSum = sum;
11   }
12 }
13 System.out.print("Row at index " + maxRow);
14 System.out.println(" has the maximum sum " + maxSum);
```

Q: What if statement `int sum = 0;` at L4 is moved, outside the for-loop, between L2 and L3?

Two-Dimensional Arrays: Example (5)

Problem: Given a 2D array `a` of integers, determine if all elements are positive.

```
boolean allPos = true;
for(int row = 0; row < a.length; row ++) {
  for(int col = 0; col < a[row].length; col ++) {
    allPos = allPos && a[row][col] > 0;
  } }
if (allPos) { /* print */ } else { /* print */ }
```

Alternatively (with *early exit*):

```
boolean allPos = true;
for(int row = 0; allPos && row < a.length; row ++) {
  for(int col = 0; allPos && col < a[row].length; col ++) {
    allPos = a[row][col] > 0;
  } }
if (allPos) { /* print */ } else { /* print */ }
```


Two-Dimensional Arrays: Example (6)

Problem: Given a 2D array `a` of integers, determine if it is a *rectangle* (i.e., each row has the same number of columns).

```
if(a.length == 0) { /* empty array can't be a rectangle */ }
else { /* a.length > 0 */
  int assumedLength = a[0].length;
  boolean isRectangle = true;
  for(int row = 0; row < a.length; row ++) {
    isRectangle =
      isRectangle && a[row].length == assumedLength;
  }
  if (isRectangle) { /* print */ } else { /* print */ }
}
```

Exercise: Change the above code so that it exits from the loop *as soon as* it is found that the 2-D array is not a rectangle.

Two-Dimensional Arrays: Example (7)

Problem: Given a 2D array `a` of integers, determine if it is a *square* (i.e., each row has the same number of columns, and that number is equal to the number of rows of the 2-D array).

```
if(a.length == 0) { /* empty array can't be a square */ }
else { /* a.length > 0 */
  int assumedLength = a.length;
  boolean isSquare = a[0].length == assumedLength;
  for(int row = 0; row < a.length; row ++) {
    isSquare =
      isSquare && a[row].length == assumedLength;
  }
  if (isSquare) { /* print */ } else { /* print */ }
}
```

Exercise: Change the above code so that it exits from the loop *as soon as* it is found that the 2-D array is not a square.

Two-Dimensional Arrays: Example (8)

- **Problem:** Given a 2D array `a` of integers, print out the *lower-left triangular* area of elements.

  Assumption: The input 2D array is of a *square* shape.

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

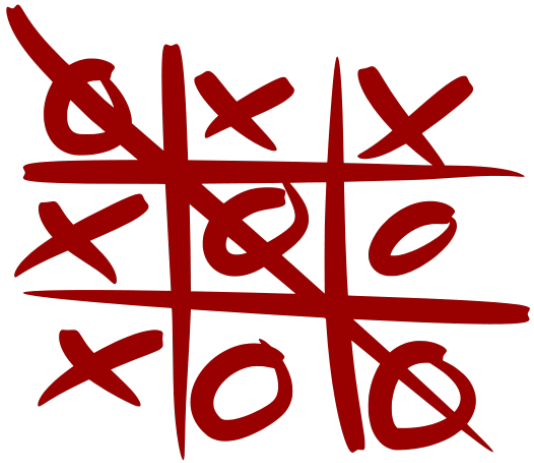
- **Problem:** *upper-left triangular* area?

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

Exercises: *upper-right triangle*? *lower-right triangle*?

Two-Dimensional Arrays: Example (9)

Consider the tic-tac-toe game:



Index (1)

Learning Outcomes
Assumptions
Sequential Loops vs. Nested Loops
Nested Loops: Finding Duplicates (1)
Nested Loops: Finding Duplicates (2)
Nested Loops: Finding Duplicates (3)
Nested Loops: Finding Duplicates (4)
Nested Loops: Finding Duplicates (5)
Nested Loops: Finding Duplicates (6)
Nested Loops: Finding Duplicates (7)
2-D Arrays: Motivating Example (1)

Index (2)

2-D Arrays: Motivating Example (2.1)
2-D Arrays: Motivating Example (2.2)
2-D Arrays: Motivating Example (2.3)
2-D Arrays: Initialization (1)
2-D Arrays: Initialization (1)
2-D Arrays: Lengths (1)
2-D Arrays: Lengths (2)
2-D Arrays: Assignments
Revisiting the Motivating Example
Two-Dimensional Arrays: Example (1)
Two-Dimensional Arrays: Example (2)

Index (3)

Two-Dimensional Arrays: Example (3)
Two-Dimensional Arrays: Example (4)
Two-Dimensional Arrays: Example (5)
Two-Dimensional Arrays: Example (6)
Two-Dimensional Arrays: Example (7)
Two-Dimensional Arrays: Example (8)
Two-Dimensional Arrays: Example (9)

Using API in Java

YORK UNIVERSITÉ UNIVERSITY

EECS1022 Sections M & N:
Programming for Mobile Computing
Winter 2021

CHEN-WEI WANG

Learning Outcomes

Understand:

- Self-Exploration of Java API
- Method Header
- Parameters vs. Arguments
- Static Methods and Math Library
- Non-Static Methods and Collection Library

Application Programming Interface (API)

- Each time before you start solving a problem:
  - As a ***beginner***, it is crucial to implement **everything** by yourself.
  - As you get more ***experienced***, first check to see if it is already solved by one of the library classes or methods.
    Rule of the Thumb: DO NOT REINVENT THE WHEEL!
- An *Application Programming Interface (API)* is a collection of *programming facilities* for *reuse* and building your applications.
- Java API contains a library of *classes* (e.g., `Math`, `ArrayList`, `HashMap`) and *methods* (e.g., `sqrt`, `add`, `remove`):

  `https://docs.oracle.com/javase/8/docs/api/`
- To use a library class, put a corresponding `import` statement:

```
import java.util.ArrayList;
class MyClass {
 ArrayList myList;
 ...
}
```


Classes vs. Methods

- A *method* is a ***named*** block of code *reusable* by its name.
  e.g., As a user of the `sqrt` method (from the `Math` class):
  - Implementation code of `sqrt` is *hidden* from you.
  - You only need to know how to *call* it in order to use it.
  - A *non-static method* must be called using a context object.
    e.g., Illegal to call `ArrayList.add("Suyeon")`. Instead:

```
ArrayList<String> list = new ArrayList<String>();
list.add("Suyeon")
```

  - A *static method* can be called using the name of its class.
    e.g., By calling `Math.sqrt(1.44)`, you are essentially *reusing* a block of code, *hidden* from you, that will be executed and calculate the square root of the input value you supply (i.e., `1.44`).
- A *class* contains a collection of *related* methods.
  e.g., The `Math` *class* supports *methods* related to more advanced mathematical computations beyond the simple arithmetical operations we have seen so far (i.e., `+`, `-`, `*`, `/`, and `%`).

Parameters vs. Arguments

- *Parameters* of a *method* are its *input variables* that you read from the API page.

  e.g., `double pow(double a, double b)` has:
  - two parameters `a` and `b`, both of type `double`
  - one output/return value of type `double`
- *Arguments* of a *method* are the specific *input values* that you supply/pass in order to use it.

  e.g., To use the `pow` method to calculate 3.4^5, we call it by writing `Math.pow(3.4, 5)`.
- *Argument values* must conform to the corresponding *parameter types*.

  e.g., `Math.pow("three point four", "5")` is an invalid call!

Example Method Headers: Math Class

- The class Math contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

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.

- *Method Overloading*: multiple methods sharing the *same name*, but with *distinct lists* of parameters (e.g., `abs` method).
- The `abs` method being `static` allows us to write `Math.abs(-2.5)`.

Header of a Method

Header of a *method* informs users of the *intended usage*:
- *Name* of method
- List of *inputs* (a.k.a. *parameters*) and their types
- Type of the *output* (a.k.a. *return type*)
  - Methods with the `void` return type are mutators.
  - Methods with non-`void` return types are accessors.

e.g. In Java API, the **Method Summary** section lists *headers* and descriptions of methods.

Case Study: Guessing a Number

Problem: Your program:
- *internally* and *randomly* sets a number between 0 and 100
- *repeatedly* asks the user to enter a guess, and hints if they got it, or should try something smaller or larger
- once the user got it and still wishes to continue, *repeat* the game with a different number

Hints:

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

```
(int) Math.random() * 100
```

or

```
(int) (Math.random() * 100)
```

??

Example Method Headers: ArrayList Class

An ArrayList acts like a “resizable” array (indices start at 0).
Extra tutorial here.

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

Example Method Headers: HashTable Class

A HashTable acts like a two-column table of (searchable) keys and values. ***Extra tutorial here.***

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

Case Study: Using an ArrayList

```
1  import java.util.ArrayList;
2  public class ArrayListTester {
3    public static void main(String[] args) {
4      ArrayList<String> list = new ArrayList<String>();
5      println(list.size());
6      println(list.contains("A"));
7      println(list.indexOf("A"));
8      list.add("A");
9      list.add("B");
10     println(list.contains("A")); println(list.contains("B")); println(list.contains("C"));
11     println(list.indexOf("A")); println(list.indexOf("B")); println(list.indexOf("C"));
12     list.add(1, "C");
13     println(list.contains("A")); println(list.contains("B")); println(list.contains("C"));
14     println(list.indexOf("A")); println(list.indexOf("B")); println(list.indexOf("C"));
15     list.remove("C");
16     println(list.contains("A")); println(list.contains("B")); println(list.contains("C"));
17     println(list.indexOf("A")); println(list.indexOf("B")); println(list.indexOf("C"));
18
19     for(int i = 0; i < list.size(); i ++) {
20       println(list.get(i));
21     }
22   }
23 }
```

See ***Java Data Types*** (3.3.1) – (3.3.2) in ***Classes and Objects*** for another example on ArrayList.

Case Study: Using a HashTable

```
1  import java.util.Hashtable;
2  public class HashTableTester {
3    public static void main(String[] args) {
4      Hashtable<String, String> grades = new Hashtable<String, String>();
5      System.out.println("Size of table: " + grades.size());
6      System.out.println("Key Alan exists: " + grades.containsKey("Alan"));
7      System.out.println("Value B+ exists: " + grades.containsValue("B+"));
8      grades.put("Alan", "A");
9      grades.put("Mark", "B+");
10     grades.put("Tom", "C");
11     System.out.println("Size of table: " + grades.size());
12     System.out.println("Key Alan exists: " + grades.containsKey("Alan"));
13     System.out.println("Key Mark exists: " + grades.containsKey("Mark"));
14     System.out.println("Key Tom exists: " + grades.containsKey("Tom"));
15     System.out.println("Key Simon exists: " + grades.containsKey("Simon"));
16     System.out.println("Value A exists: " + grades.containsValue("A"));
17     System.out.println("Value B+ exists: " + grades.containsValue("B+"));
18     System.out.println("Value C exists: " + grades.containsValue("C"));
19     System.out.println("Value A+ exists: " + grades.containsValue("A+"));
20     System.out.println("Value of existing key Alan: " + grades.get("Alan"));
21     System.out.println("Value of existing key Mark: " + grades.get("Mark"));
22     System.out.println("Value of existing key Tom: " + grades.get("Tom"));
23     System.out.println("Value of non-existing key Simon: " + grades.get("Simon"));
24     grades.put("Mark", "F");
25     System.out.println("Value of existing key Mark: " + grades.get("Mark"));
26     grades.remove("Alan");
27     System.out.println("Key Alan exists: " + grades.containsKey("Alan"));
28     System.out.println("Value of non-existing key Alan: " + grades.get("Alan"));
29     }
30 }
```

Beyond this lecture...

- Your ***Week 12 Java Tutorial***
  How to build a Birthday Book application using both `ArrayList` and `HashTable`
- Extra Tutorials on:
  - `ArrayList` [link]
  - `HashMap` (similar to `Hashtable`) [link]

Index (1)

Learning Outcomes
Application Programming Interface (API)
Classes vs. Methods
Parameters vs. Arguments
Header of a Method
Example Method Headers: Math Class
Case Study: Guessing a Number
Example Method Headers: ArrayList Class
Case Study: Using an ArrayList
Example Method Headers: HashTable Class
Case Study: Using a HashTable

Index (2)

Beyond this lecture...