Unit 3: **Basic Java Programming - 2**

Object-Oriented Programming (OOP)

CCIT 4023, 2025-2026

U3: Basic Java Programming - 2

- Numerical Types and Operations
  - Order of Evaluations (precedence and associativity)
  - Type Conversion / Casting (promotion rules)
  - Class **`Math`** for mathematical functions
- Control Flow – Repetition Statements
  - **`while, do-while, for`**
  - **`break`** and **`continue`**
- Simple Array of Primitive Types and Strings
- Multiple Methods - Modularizing Programs
- Working with Multiple Java Source Files
- Simple GUI, with Java Class **`JOptionPane`**

**Remark:* This unit provides a very condensed introduction of starting Java programming, and can much be treated as a revision for learners already with basic programming background.

Numerical Types and Operations

- Numeric primitive data types include: `byte`, `short`, `int`, `long`, `float`, and `double`.
  - These numeric data types from the top (`byte`) are regarded as having *lower precision* than those to the bottom (`double`).

Low Precision

Data Type	**Default value***	**Size**	**Range** (Minimum ~ Maximum)
`byte`	`0`	8-bit	$-2^7 \sim 2^7-1$ (-128 ~ 127)
`short`	`0`	16-bit	$-2^{15} \sim 2^{15}-1$ (-32,768 ~ 32,767)
`int`	`0`	32-bit	$-2^{31} \sim 2^{31}-1$
`long`	`0L`	64-bit	$-2^{63} \sim 2^{63}-1$
`float`	`0.0f`	32-bit	Approx. ±3.40282347E+38
`double`	`0.0`	64-bit	Approx. ±1.79769313486231570E+308

High Precision

* *Remark*: Default values are applied to fields of objects, not local variables.

Arithmetic Operators

- The following table shows some binary arithmetic operators in Java.
  - Data type of the result of these binary arithmetic operations is the same data type
    - of its operands, if both operands are of the same type.
    - of its operand which has a *higher precision*, if two operands are of different types (*promotion rule*).

Operator (Arithmetic)	Operation Description	Example Expression (suppose **a=8, b=5, c=2.5**)	Result
`*`	Multiplication	**a*b** **a*c**	40 20.0
`/`	Division	**a/b** **a/c**	1 3.2
`%`	Modulus (Remainder)	**a%b** **a%c**	3 0.5
`+`	Addition	**a+b** **a+c**	13 10.5
`-`	Subtraction	**a-b** **a-c**	3 5.5

Summary of Operators

Simple Assignment Operator

`=`	Simple assignment operator

Arithmetic Binary Operators

`+`	Additive operator (also used for String concatenation)
`-`	Subtraction operator
`*`	Multiplication operator
`/`	Division operator
`%`	Remainder (modulus) operator

Unary Operators

`+`	Unary plus operator; indicates positive value (**`+expr`**)
`-`	Unary minus operator; negates an expression (**`-expr`**)
`++`	Increment operator; increments a value by `1` (**`expr++`**)
`--`	Decrement operator; decrements a value by `1` (**`expr--`**)
`!`	Logical complement operator; inverts the value of a boolean

* Remark: Numbers are positive without + unary plus operator by default

Summary of Operators

Equality and Relational Operators

`==`	Equal to (Easily missed-up with = assignment operator)
`!=`	Not equal to
`>`	Greater than
`>=`	Greater than or equal to
`<`	Less than
`<=`	Less than or equal to

Logical Operators

`&&`	Conditional-AND
`||`	Conditional-OR
`!`	Logical complement operator; inverts the value of a boolean

Conditional Ternary Operators

`?:`	Ternary (shorthand for if-then-else statement)

Summary of Operators

Reference Only

Type Comparison Operator

`instanceof`	Compares an object to a specified type

Bitwise and Bit Shift Operators

`~`	Unary bitwise complement
`<<`	Signed left shift
`>>`	Signed right shift
`>>>`	Unsigned right shift
`&`	Bitwise AND
`^`	Bitwise exclusive OR
`|`	Bitwise inclusive OR

Order of Evaluations
(of Arithmetic Expression)

- Given an arithmetic expression with more than one operators, how does the expression get evaluated? Example: `x + 3 * y`
  - **Answer**: First `3*y`, then with `+` the `x` is added later

- Determine the order of evaluation by following the ***Precedence Rules***
  - Operators are ranked according to precedence order
  - A higher precedence operator is evaluated before the lower one
  - If two operators are in the same precedence level, they are then evaluated from left to right for most binary operators (*associativity*)

Operators	**Precedence** (From the highest at the top, to the lowest at the bottom)
Multiplicative	`* / %`
Additive	`+ -`
Relational	`< > <= >= instanceof`
Equality	`== !=`
Logical AND	`&&`
Logical OR	`||`
Assignment	`= += -= *= /= %= &= ^= |= <<= >>= >>>=`

Operator +: Applied to Numbers Vs. String
(Operator + Overloading)

- Given two numbers, e.g. `a=12.3` and `b=21`, the result of applying the operator + to them (`a+b`) is a number (`33.3`), adding one to another
- This + operator can also be applied to two Strings e.g. `a="12.3"` and `b= "21"`, and the result becomes a new string (`"12.321"`), appending one to another (concatenation)
- In general, the plus operator + can have different operations, depending on the context
  - Numbers: `<var1> + <var2>` is an ***addition***, if both are numbers
  - String: If any one of them is a `String`, it is a ***concatenation***
  - Evaluation goes from left to right (associativity) if more than one + operators

```
result = "ABC" + 12 + 3;
// produce "ABC123"
```

```
result = 12 + 3 + "ABC";
// produce "15ABC"
```

Precedence Rules

Reference Only

Operators	Precedence (From highest at top, to bottom)
postfix	`expr++ expr--`
unary (and prefix)	`++expr --expr +expr -expr ~ !`
multiplicative	`* / %`
additive	`+ -`
shift	`<< >> >>>`
relational	`< > <= >= instanceof`
equality	`== !=`
bitwise AND	`&`
bitwise exclusive OR	`^`
bitwise inclusive OR	`|`
logical AND	`&&`
logical OR	`||`
ternary	`? :`
assignment	`= += -= *= /= %= &= ^= |= <<= >>= >>>=`

** When operators of equal *Precedence* appear in the same expression, *Associativity* rules govern which is evaluated first. All binary operators except for the assignment operators are evaluated from left to right; assignment operators and unary operators are evaluated right to left.

Data Type Compatibility and Conversion

- If the value of one data type could be converted to that of another data type, these two data types are *compatible*.
- Converting (or casting) compatible types could be done in different ways, including:
  - Widening primitive conversion: In general, value of the original type would be preserved (no precision lost) in the new type.
    - Convert integer `int 123` to real number `double 123.0`
    - This type of conversion / casting could be done automatically - *Implicit Casting*. E.g. `double dNum = 123;`
  - Narrowing primitive conversion: In general, value of the original type may not be fully preserved (precision lost) in the new type.
    - Convert `double 12.3` to integer `int 12`
    - This conversion type normally could not be done automatically, but could be done expliicitly - *Explicit casting*. E.g.
    - `int aNum = (int) 12.3;   // OK, with casting`
    - `int bNum = 12.3;         // ERROR in Java`

Type Conversion / Casting

- Mixed expressions include elements of different data types
  - The data type of the evaluated result value is based on the programming language specification
- In Java, data types of operands in mixed expressions are converted or casted based on the **Promotion Rules**
  - Data type of an expression will be the same as the data type of an operand whose type has the **highest precision**, an thus promote all operands and result to this highest precision
- *Example 1*: If `x` is a `double` and `y` is an `int`, what data type of expression **`x*y`** ? → Answer: `double`
  - With Promotion Rules, `y` is promoted to the highest precision type `double` in this case from type `int` first, before multiplication ***** evaluates to type `double`
  - Similar to case: `1.2*`**`3`** promote→ `1.2*`**`3.0`** → `3.6`

Implicit Type Conversion / Casting

- *Example 2*:

```
double x = 3+5;
```

- Result of right-hand-side expression `3+5` is `8` of type `int`
- In this assignment statement with the operator **=**
  - The (left-hand-side) variable x is double
  - The (right-hand-side) assigned value `8` (original type `int`) is first promoted to `8.0` (casted type `double`), before assigned to `x`
- This **implicit type conversion** / **casting** is done *automatically*

Explicit Type Conversion / Casting

- *Example 3*:

```
int x = 3.5;
```

✗

A higher precision value cannot be assigned to a lower precision variable, it will cause loss of precision

  - (***Demotion*** is not allowed in Java!)

- We cannot rely on the promotion rules with "implicit" type casting in this case, but we can make an **explicit type conversion** / **casting** (or just simply called casting)
  - by prefixing the operand with the data type enclosed with parentheses round-bracket-pair `()`, using the following syntax:

```
( <data type> ) <expression>
```

  - Examples:

```
float a =(float) x / 3;
int b =(int) (x / y * 3.0);
```

Type of `x` is first casted to `float`, with `(float)` and then divide it by `3`.

Casting the result of the expression `x / y * 3.0` to `int`.

The `Math` Class (in the package of `java.lang`)

- Very often, we may need to solve more complicated mathematical problems
- The `Math` class contains class methods for commonly used mathematical functions
  - `Math` class is in the `java.lang` package, which is always imported by default. As such, we do not need to import this package.
- Below sample code example uses methods in `Math` class to generate a random number, calculate its square-root, and the area of circle.

```
double rNum = Math.random()*40.23; // range 0.0 ~ <40.23
double sqRoot = Math.sqrt(rNum); // square root
double area = Math.PI * rNum * rNum; // circle area

System.out.printf("Number:[%.2f], sqRoot:[%.2f], area:[%.2f]\n",
                      rNum, sqRoot, area);
```

Reference: https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/lang/Math.html

Commonly Used Constant and Class Methods in `Math`

Constant	Description
`static double E`	The double value that is closer than any other to e, the base of the natural logarithms.
`static double PI`	The double value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.

Method	Description
`exp(a)`	Natural number **e** raised to the power of **a**
`log(a)`	Natural logarithm (base **e**) of **a**
`floor(a)`	The largest double value less than or equal to **a**
`max(a,b)`	The greater of **a** and **b**
`pow(a,b)`	The number **a** raised to the power of **b**
`sqrt(a)`	The square root of **a**
`random()`	Returns a double value with a positive sign, greater than or equal to `0.0` and less than `1.0`

A list of class methods defined in the `Math` class:
https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/lang/Math.html

Control Flow – Repetition Statements

- Repetition statement controls a block of code to be executed repeatedly, until a certain condition is met
- Repetition statements are also called loop statements
- Three repetition statements in Java:
  - **`while`** statement *continually* executes a block of statements as long as the `boolean` condition is `true`. The `boolean` evaluation is done first (at top of the statement).
  - **`do-while`** statement works similarly as `while`, but the `boolean` evaluation is done last (at bottom of the statement).
  - **`for`** statement provides a more compact form of loop, especially useful for iterating over a range of values, with 3 sections for initialization, condition evaluation, and updating, as below

```
for(<initialization>; <condition>; <updating>){
    <loop-body>
}
```

* Similar to `if` statement, the brace pair `{}` (curly brackets) is optional if the loop-body block has only one single simple statement line

Syntax for the `while` Statement

- General syntax is similar to that of `if` statement:

```
while ( <boolean expression> ) {
    <loop-body statement(s)>
}
<other statements>
```

Loop-body, the true-Block: Statements are executed if `<boolean expression>` is (`true` condition), then loop back to evaluate again.

* Similar to `for` statement, the brace pair `{ }` (curly brackets) is optional if the loop-body block has only one single simple statement line

- The `while` statement *continually* executes a loop-body block of statements, as long as `boolean` condition is `true`

- Each loop starts with first evaluating `<boolean expression>`
  - `true` value of `<boolean expression>` leads to execute the `<while loop-body statement(s)>`, then starts another loop again
  - `false` value of `<boolean expression>` leads to end the `while` statement, and directly jump out of the loop.

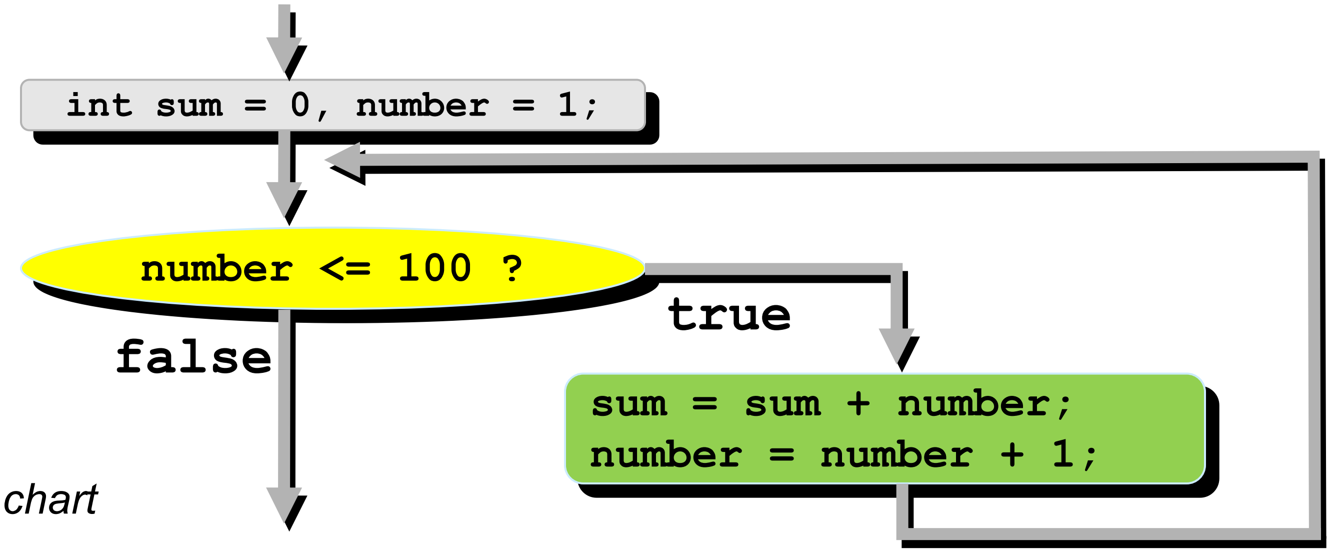
Example: Control Flow of `while`

```
int sum = 0, number = 1;
while (  number <= 100   ) {
    sum    =  sum + number;
    number = number + 1;
}
```

Boolean Expression

Statement (loop-body)

These statements are executed as long as `number` is less than or equal to `100`.



The flow chart

The do-while Statement

- General syntax:

```
do {
      <loop-body statement(s)>
} while ( <boolean expression> ) ;
<other statements>
```

Remember to add this ; (Semi-colon)

- The do-while statement *continually* executes a loop-body block of statements, as long as boolean condition is true

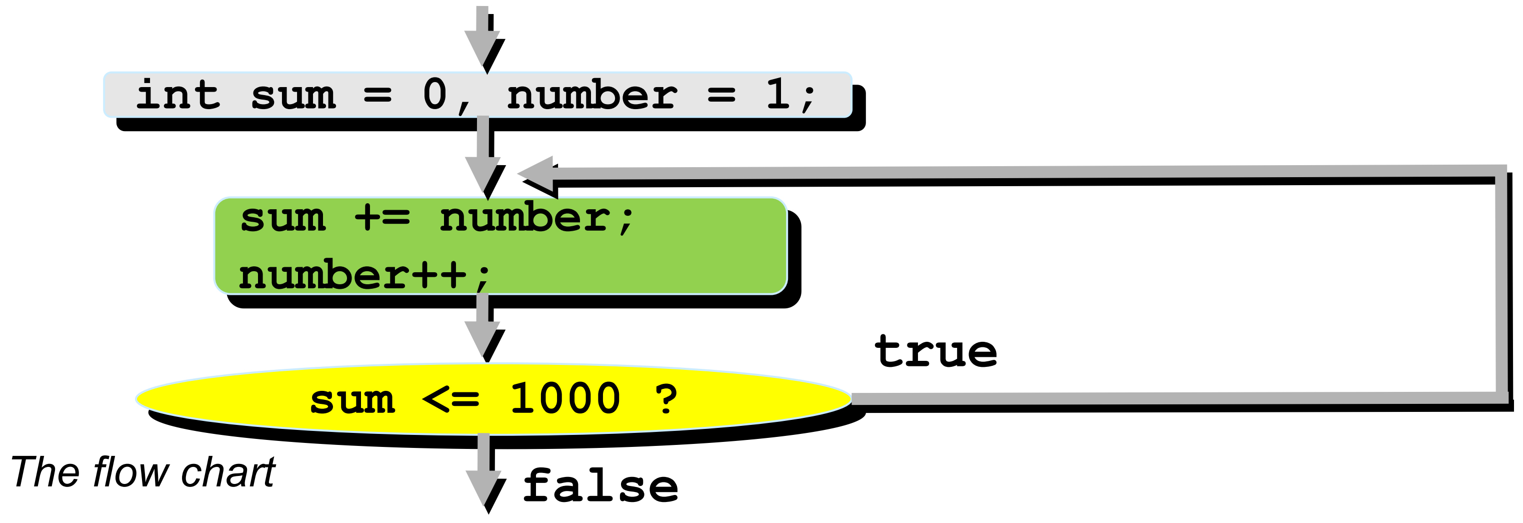
- Each loop starts with executing <loop-body statement(s)> and then evaluating <boolean expression>
  - true value of <boolean expression> leads to start another loop again
  - false value of <boolean expression> leads to end the do-while statement, and directly jump out of the loop.

Example: Control Flow of `do-while`

```
do    {
   sum += number;
   number++;
} while ( sum <= 1000 );
```

Statement (loop-body): These statements are executed as long as sum is less than or equal to 1000

Boolean Expression



The flow chart

Loop-and-a-Half Repetition Control

- *Loop-and-a-half repetition control* can be used to test a loop's terminating condition in the middle of the loop body
- This could be implements by using the reserved words `while`, `if` and **`break`**
  - `break` statements terminate the *innermost* `switch`, `for`, `while`, or `do-while` statements
    - *Be careful*, `break` is not associated to `if` statements
    - *For reference only:* there is another `break` type, *labeled* *`break`*, which may be used to terminates an outer loop statement.

Example: Loop-and-a-Half Control

- Example code below asks a valid string input from user
  - Although the `while` loop condition is `true` which “looks” like an infinite loop, there is a common way to end the loop with `break`
    - If user input is valid (not empty string), then `true` condition of `(name.length()>0)`, execute `if`-body → `break` loop `while`
    - If user input is invalid (empty string), then `false` condition of `(name.length()>0)` as `name.length()` is `0` , not execute `if`-body → keep doing with the `while` loop, and again

```
String name;
while (true){ // true all time, when check condition HERE
    name = JOptionPane.showInputDialog(null, "Your name");
    if (name.length() > 0)
        break; // HERE break the while loop
    JOptionPane.showMessageDialog(null, "Invalid Entry." +
                     "You must enter at least one character.");
}
```

The `for` Statement

- General syntax of `for` loop with 3 sections:

```
for(<initialization>; <boolean expression>; <updating>){
    <loop-body>
}
```

- `<initialization>` is done first and only once, before beginning the first loop.
  - This is commonly used for loop setting-up, such as initializing the counter value, e.g. `int i = 0`
- Each loop first evaluates the condition `<boolean expression>`
  - `true` value of `<boolean expression>` leads to execute the `<loop-body statement(s)>`, followed by `<updating>`, then starts another loop
  - `false` value of `<boolean expression>` leads to end the `for` statement, and directly jump out of the loop

Example: Control Flow of Statement `for`

<initialization>

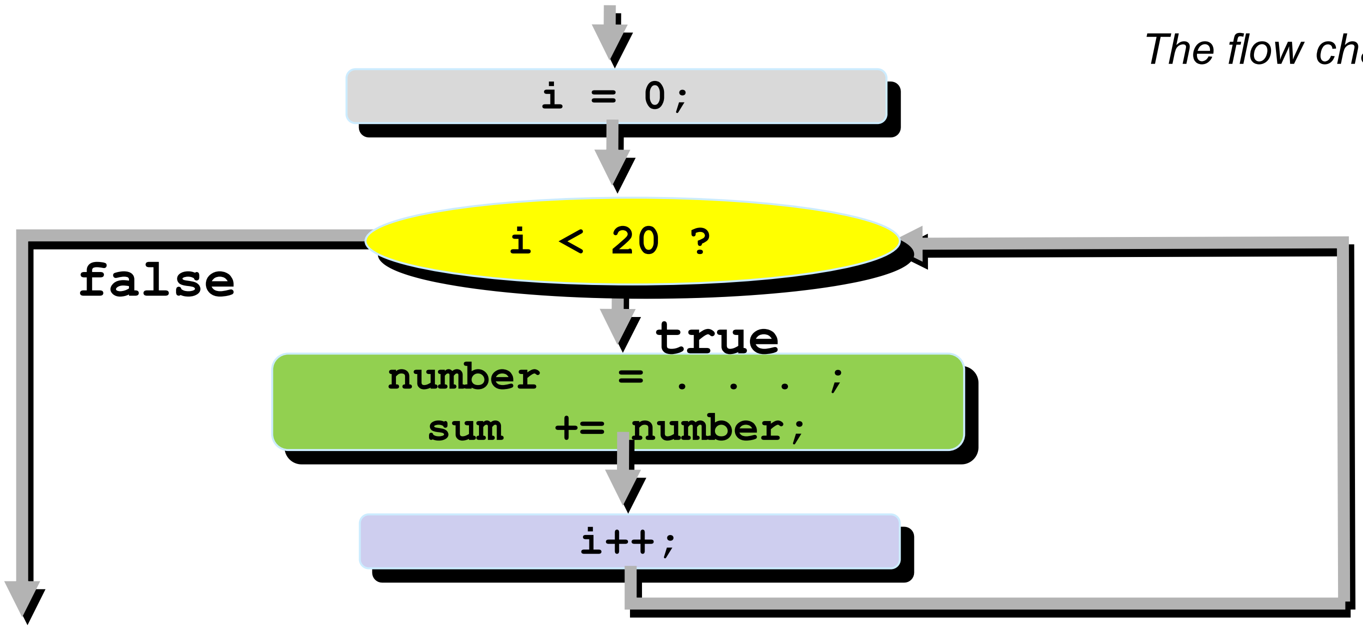
<boolean expression> *(Condition)*

<updating> *(Increment)*

```
for ( int i = 0 ;  i < 20  ;  i++      ) {
    number = scanner.nextInt();
    sum += number;
}
```

Statements (loop-body)

Loop-body statements are executed for 20 times (`i = 0, 1, 2, … , 19`)



The flow chart

More `for` Loop Examples

1

```
for (int i = 0; i < 100; i += 5)
```

```
i = 0, 5, 10, … , 95
```

2

```
for (int j = 2; j < 40; j *= 2)
```

```
j = 2, 4, 8, 16, 32
```

3

```
for (int k = 100; k >= 0; k--)
```

```
k = 100, 99, 98, 97, ..., 1, 0
```

`break` and `continue`

- Use **`break`** to *terminate* the *innermost* loop (`for`, `while`, or `do-while`) or `switch` statement.
  - *Be careful*, `break` is not associated to `if` statements
  - E.g. the following matches a number against elements in an array

```
boolean contain = false;
for (int i = 0; i < numArray.length; i++) { // numArray is an array
        if (numArray[i] == matchingNum) {
                contain = true;
                break; // this breaks the for loop (NOT if statement)
        }
}
```

- Use **`continue`** statement to *skip* the *innermost* loop (`for`, `while`, or `do-while`), and re-evaluate
  - It skips to the end of the *innermost* loop body, and continues another loop by first evaluating the `boolean` expression that controls the loop
  - Using `continue` is less common compared to `break`

Simple Array of Primitive Types and Strings

- It is common to use repetition in accessing array
  - An *array* is an indexed collection of data values of the same data type
  - E.g. we may use array to deal with 100 integers (`int`), 500 `Account` objects, 12 real numbers (`double`), etc.
  - In Java, an array is a reference type, similar to a class
- Array ***Declaration***: two ways in Java

```
<data type> [] <variable>  //variation 1,recommended in Java
<data type>  <variable> [] // variation 2, NOT recommended
```

- Array ***Creation*** (and ***Assignment***)

```
<variable>  = new <data type> [ <size> ];
```

```
double[] rainfall;
rainfall = new double[12];
```

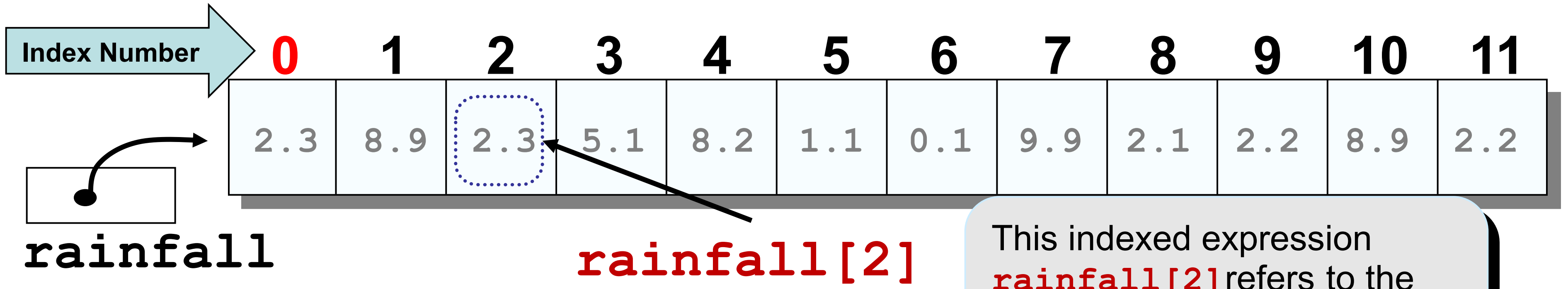
```
// declare and create(& assign) in one line
double[] rainfall = new double[12];
```

Initializing and Accessing Individual Elements

- To declare & initialize an array in a statement, e.g.

```
double[] rainfall = { 2.3, 8.9, 2.3, 5.1, 8.2, 1.1,
                        0.1, 9.9, 2.1, 2.2, 8.9, 2.2 };
String[] students = { "Amy", "Brenda", "Candy" };
```

- Individual elements in an array are accessed with the *indexed expression*, e.g.



This indexed expression `rainfall[2]` refers to the element at position #2 (with value `2.3`)

* The index of the first position in an array is **0**

Converting Array to String

- In general we should not print and display arrays directly, otherwise unreadable string information will normally be displayed, e.g.

```
double[] rainfall = { 2.3, 8.9, 2.3, 5.1, 8.2, 1.1,
    0.1, 9.9, 2.1, 2.2, 8.9, 2.2 };
System.out.println("Rainfall is " + rainfall);
```

```
Rainfall is [D@15aeb7ab
```

** This scenario also applies to some other reference types, including class types.*

- Instead, we may also use the method **`toString()`** of class **`Arrays`** in package **`java.util`** to convert an array to a proper string for further processing, such as for display, e.g.

```
double[] rainfall = { 2.3, 8.9, 2.3, 5.1, 8.2, 1.1,
    0.1, 9.9, 2.1, 2.2, 8.9, 2.2 };
String rfStr = java.util.Arrays.toString(rainfall); //convert array to str
System.out.println("Rainfall is " + rfStr);
```

```
Rainfall is [2.3, 8.9, 2.3, 5.1, 8.2, 1.1, 0.1, 9.9, 2.1, 2.2, 8.9, 2.2]
```

Example of Accessing Array with `for` Statement

- The following examples compute the average rainfall or display students, with `for` loop

```
double[] rainfall = new double[12];
double annualAverage, sum = 0.0;
//... Setting elements in array
for (int i = 0; i < rainfall.length; i++)
    sum += rainfall[i]; // Getting value from array
annualAverage = sum / rainfall.length;
```

The public constant `length` returns the size/capacity of an array

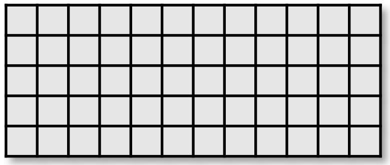
```
//... Creating and Setting elements in array of String (student names
System.out.println("Students in our class are:");
for (int i = 0; i < students.length; i++)
    System.out.println("Student " + i + students[i]);
```

** Remark:	Array of objects is even more powerful.*
More details of Java array will be introduced in later unit.

Simple 2D Array

- Simple 2D Array: Declare, Create (& Assign)

```
<Type>[][] <varName>  = new <Type> [<#row>][<#col>];
```



- 2D arrays are useful in representing 2D tabular data
  - E.g. Calculate average of each year (in 3-year data)

```
double[][] rainfalls = new double [5][12]; // in 5 years
// ...
double[] averages = { 0.0, 0.0, 0.0, 0.0 , 0.0};
for (int i = 0; i < rainfalls.length; i++) {
    for (int j = 0; j < rainfalls[i].length; j++) {
        averages[i] += rainfalls[i][j];
    }
    averages[i] = averages[i] / rainfalls[i].length;
}
```

- Simple higher-dimension array could be manipulated in a similar manner, e.g. 4D array:

```
double[][][][] data4D  = new double [10][20][30][40];
```

Enhanced-`for` Loop Statement

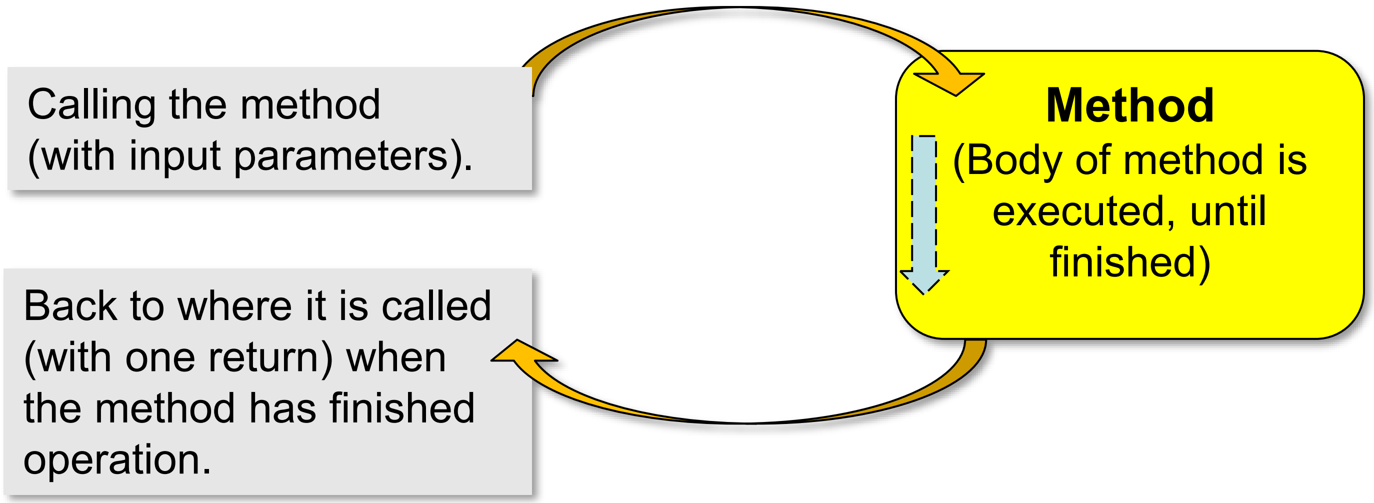
- The `for` statement has another form designed for iteration through arrays
- This form is referred to the "**enhanced-`for`**" statement, and often makes the loops more compact and easy to read (similar to Python's `for`)
  - However, this enhanced-`for` loop is lack of fine-control in the looping (and array indexing) in some cases.
- E.g. consider the array `rainfall` in the last example, and how the enhanced-for loop is used:

```
double[] rainfall = new double[12];
double sum = 0.0;
// ...
for (double rf : rainfall ) //enhanced-for loop
  sum += rf;
```

```
// "similar" to conventional for-loop below
for (int i=0; i < rainfall.length; i++)
  sum += rainfall[i];
```

Multiple Methods (Modularizing Programs)

- One important approach of developing a large program is to construct it from small, simple pieces, or modules
  - This is the basic idea in procedural programming, e.g. working with multiple functions in C
  - A method is an operational module. When it is called with possible inputs (parameters), it executes codes in the method body, until it ends and returns to where it is called (with certain return value).


Calling the method (with input parameters).
Method
(Body of method is executed, until finished)
Back to where it is called (with one return) when the method has finished operation.

Multiple Methods
(Modularizing Programs)

- The basic syntax of declaring a method:

```
<modifier(s)> <return type> <method name> (<parameter(s)>) {
      <method body>
}
```

A **parameter** has the same value of the passed argument, in the method declaration

- *E.g.*

```
public static int calSum(int a, int b, int c){
    int retSum = a + b + c;
    return retSum; // return value must match return type
}
```

- The basic syntax of calling a method:

```
<method name> (<argument(s)>)
```

E.g.

```
sum = calSum(12, 34, 56);
```

An **argument** is a value we pass to a method, in method calling

** Noted: The number of arguments and parameters must be the same, and their type must be matched (compatible)*

Multiple Methods
(Modularizing Programs)

- Examples of different forms of methods

```
// 1. example method, without parameter, without return value
public static void simMethod(){
// ... method body, without return value
}

// 2. example method, without parameter, with return value/type
public static double getValue(){
// ... method body, with return value
    return retV; //return value match return type, retV is double
}

// 3. example method, with parameter and without return value
public static void main(String[] args){
// ... method body, with return value
}

// 4. example method, with parameter and with return value/type
public static int calSum(int a, int b, int b){
// ... method body, with return value
    int retSum = a + b + c;
    return retSum; // return value must match return type
}
```

Multiple Methods
(Modularizing Programs)

- A similar matter exists in defining multiple methods in Java programming
  - We may define multiple methods in one Java class, similar to the `main()` method format, e.g.

```
public static int calSum(int[] nums)
public static void main(String[] args)
```

  - A method will not be executed, until it is called (e.g. by another method)
  - The sample program for numerical calculation in next slide `NumCal` creates an array of numbers, calculates their sum and average, and displays the result
  - Although we could do all works in the method `main`, this program shows how we modularize with three different methods, `main()`, `calSum()`, and `calAvg()`

** Remark: At the moment, we focus on `static` methods, which can be called without the need for an object of the class to exist. Details will be given in later units.*

Multiple Methods
(Sample Program for Numerical Calculation)

```java
public class NumCal{ // declare a class

// method to calculate total sum:
  public static int calSum(int [] nums){
    int retSum = 0;
    for (int i=0; i<nums.length; i++)
        retSum += nums[i];
    return retSum;
  }

// method to calculate average:
  public static float calAvg(int [] nums){
    return (float)calSum(nums)/nums.length;
  }

// method main, to start the program:
  public static void main(String[] args){
    int [] checkNums = {1,2,3,4,5,6,7,8,9};
    int sum = calSum(checkNums);
    float avg = calAvg(checkNums);
    System.out.println(
        "Total Sum is " + sum
        + ";  Average is " + avg);
  }
}
```

2.1: Calling the method **`calSum()`** (with input parameter: integer array).

2.2: Return back to where method **`calSum()`** is called (with a return: `int`) when this method has finished operation.

2. Method `calSum()` is called, from `main()`, and will return back there after finish

1. Program starts here, the entry point method `main()`

COMPILE, on console

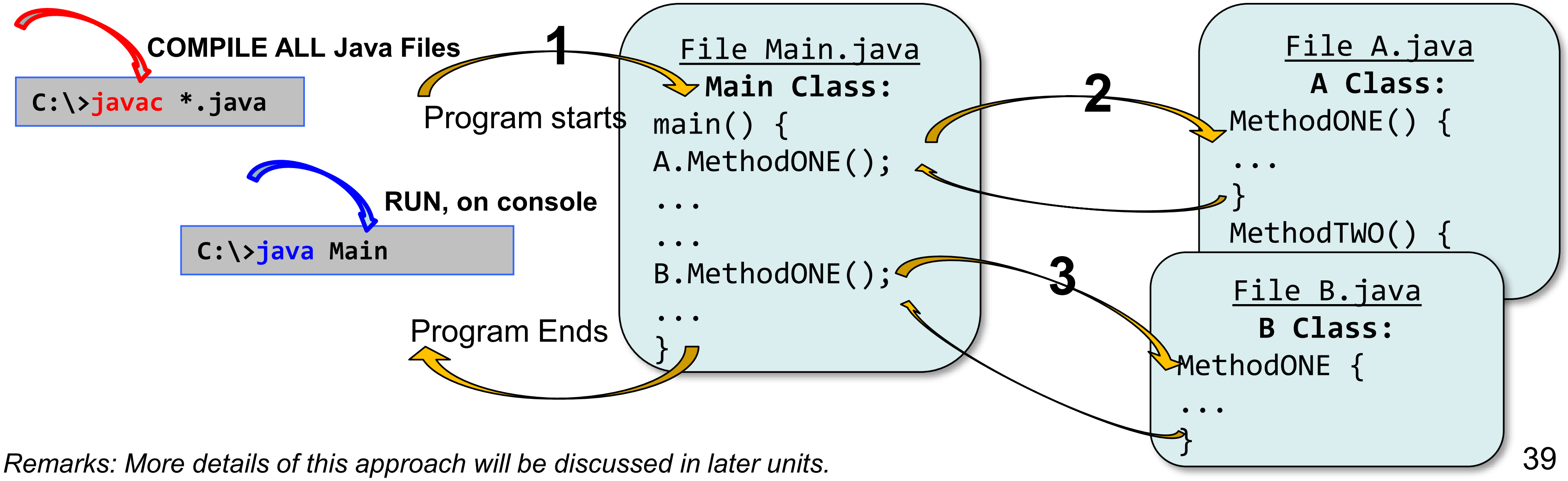
```
C:\>javac NumCal.java
```

RUN, on console

```
C:\>java NumCal
Total Sum is 45; Average is 5.0
```

Working with Multiple Java Source Files

- It is common to have more than one Java source files (`*.java`) for one program, even for simple programs
  - This approach with multiple Java files involves multiple Java classes, often one class in one java source file
  - For an executable program, it is common to separate a class module which mainly (if not only) includes the `main()` method where the program starts execution, and such class is often called the “Main” class
- Essentially the program starts with the main class, and within its `main()` method it would then call other classes to do works.



```
C:\>javac *.java
```

```
C:\>java Main
```

```
File Main.java
Main Class:
main() {
A.MethodONE();
...
...
B.MethodONE();
...
}
```

```
File A.java
A Class:
MethodONE() {
...
}
MethodTWO() {
```

```
File B.java
B Class:
MethodONE {
...
}
```

Remarks: More details of this approach will be discussed in later units.

Multiple Methods in Two Java Files
(Sample Program for Numerical Calculation – Two-Files Version)

```java
// File NumCal.java, class supporting numerical calculation
public class NumCal{ // declare a class
// method to calculate total sum:
  public static int calSum(int [] nums){
    int retSum = 0;
    for (int i=0; i<nums.length; i++)
         retSum += nums[i];
    return retSum;
  }
// method to calculate average:
  public static float calAvg(int [] nums){
    return (float)calSum(nums)/nums.length;
  }
}
```

COMPILE ALL Java Files

```
C:\>javac *.java
```

RUN, on console

```
C:\>java MainNumCal
Total Sum is 495; Average is 55.0
```

```java
// File MainNumCal.java, Main class for this numerical calculation program
public class MainNumCal{ // a Main class
  public static void main(String[] args){// method main, starts program
    int [] checkNums = {11,22,33,44,55,66,77,88,99};
    int sum = NumCal.calSum(checkNums);
    float avg = NumCal.calAvg(checkNums);
    System.out.println(
         "Total Sum is " + sum
         + ";  Average is " + avg);
  }
}
```

Use **Dot-Notation** to refer to (static) method of other classes: **`<ClassName>.<MethodName>`**

Batch Files: for Compiling and Running Java Program (Windows OS)

- Instead of repeatedly typing the same commands for compiling and running Java programs during program development, we may create and use simple batch files (on Windows platforms) for convenience.
  - The following shows 2 very simple example batch files (e.g. `jcU2Ex.bat` and `jrU2Ex.bat`), for compiling and running the `NumCal` example
  - Create and save these batch files in the same folder of our Java source files. We may then execute these batch files (in the same folder), by
    - Typing the name of the batch file in command prompt console, OR
    - Double-click directly the batch file, say in the File Explorer

```
REM File jcU2Ex.bat, for Java Compilation (All *.java files)

javac *.java

pause
```

```
REM File jrU2Ex.bat, for Java Running (Main class: MainNumCal)

java MainNumCal

pause
```

* *Remark*: Batch file script line starts with "`REM`" is a comment.

Working with Multiple Methods in a Given Class

- Apart from defining our own classes which implement a set of multiple methods, we can also make use of methods from given external classes if we are given enough information about these classes, e.g.
  - ***Methods*** and their descriptions: what they do & how to access them
- This is much related to the idea of Abstract Data Type (ADT) we have learnt in other courses:
  - External users may still use the given classes and their methods, without knowing how they are actually implemented.
- An example of a simplified ADT description of class **`NumCal`**:

Methods (`NumCal`)	**Description**
`calSum(int[]):int`	Calculate and return sum of integers
`calAvg(int[]):float`	Calculate and return average of integers
`.. And more ..`	.. And more ..
`main(String[]):`	Important method where the program starts

Working with Multiple Methods in a Given Class

Methods (`NumCal`)	Description
`calSum(int[]):int`	Calculate and return sum of integers, Java method header below: `public static int calSum(int [] nums)`
`calAvg(int[]):float`	Calculate and return average of integers, Java method header below: `public static float calAvg(int [] nums)`
`.. And more ..`	.. And more ..

- If we are given a proper class file of **`NumCal`** (`NumCal.class`, the bytecode file) with above information, even without the source code (`NumCal.java` file) external users may still make use of the class and its methods to achieve their own tasks with their own programs, e.g.

```
public class MyOwnClass{ // declare a class
// method main, to start the program:
  public static void main(String[] args){
    int [] checkNums = {1,2,3,4,5,6,7,8,9};
    int sum = NumCal.calSum(checkNums);
    float avg = NumCal.calAvg(checkNums);
    System.out.println(
         "Total Sum is " + sum
         + ";  Average is " + avg);
  }
}
```

Simple GUI, with Class `JOptionPane`

- It is less common nowadays to let public users to interact with software in a command line interface way, via standard input / output window for inputting / displaying text (on console)
- In Java, there is a useful class, for generating very simple programs with *Graphical User Interaction* (GUI) :
  - **`JOptionPane`** (in package `javax.swing`) and `import` statement required:

```
import javax.swing.JOptionPane;
```

Class: `JOptionPane` (In `javax.swing` package)

- `JOptionPane` is a class in the `javax.swing` package, for standard dialog boxes that prompt users for a value or informs them of something
- `JOptionPane` dialog is a *modal* dialog - while the dialog is on the screen, the user cannot interact with the rest of the application
- Despite of many methods, `JOptionPane` class is often used with `showXxxDialog` methods below:

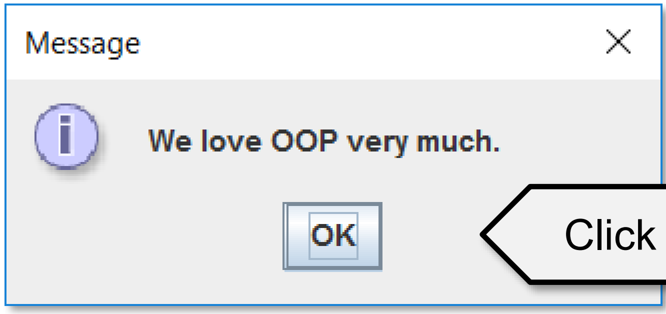
Method Name	Description
`showMessageDialog`	Tell the user about something that has happened.
`showConfirmDialog`	Asks a confirming question, like yes/no/cancel.
`showInputDialog`	Prompt for some input.

https://docs.oracle.com/en/java/javase/24/docs/api/java.desktop/javax/swing/JOptionPane.html

Class: `JOptionPane`
Method: `showMessageDialog()`

- Use `showMessageDialog` of the `JOptionPane` class is a simple way to display a result of a computation to the user
  - May display **multiple lines** of text by separating lines with a new line marker `'\n'`

```
JOptionPane.showMessageDialog(null, "We love OOP very much.");
```



We love OOP very much.

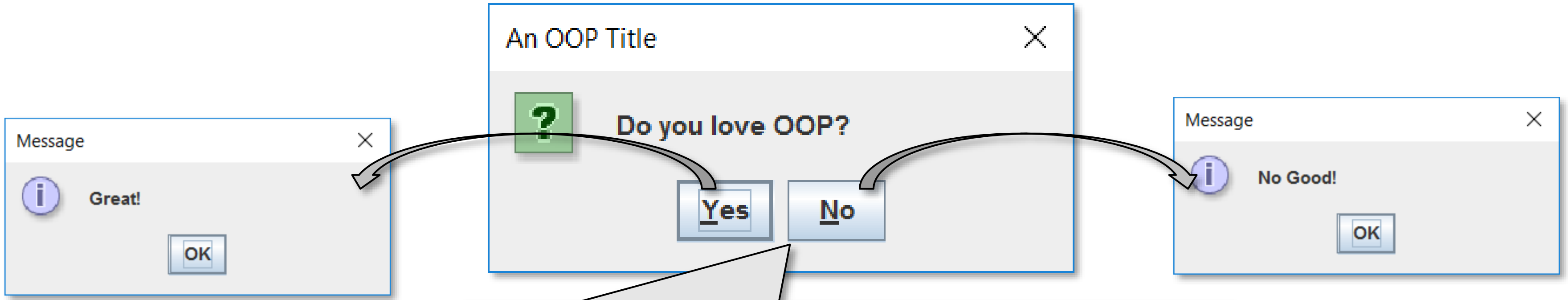
Click OK button to continue.

Reference:
public static void showMessageDialog(Component parentComponent, Object message)

Class: `JOptionPane`
Method: `showConfirmDialog()`

- Use `showConfirmDialog` of the `JOptionPane` class to bring up a dialog with the options (e.g. Yes, No and Cancel), with the title, e.g.

```
int ans = JOptionPane.showConfirmDialog( null,
    "Do you love OOP?", "An OOP Title",
    JOptionPane.YES_NO_OPTION);
if (ans==JOptionPane.YES_OPTION) //... Do Yes thing
    JOptionPane.showMessageDialog(null, "Great!");
if (ans==JOptionPane.NO_OPTION) //... Do No thing
    JOptionPane.showMessageDialog(null, "No Good!");
```



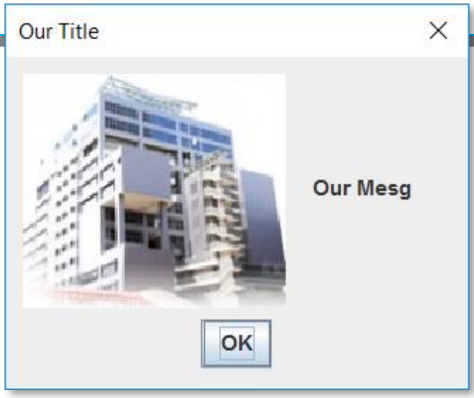
Yes, return `JOptionPane.YES_OPTION`.
No: return `JOptionPane.NO_OPTION`.

Ref: public static int showConfirmDialog(Component parentComponent, Object message, String title, int optionType)

Class: `JOptionPane`
Method: `showMessageDialog()`

- May create complex `showMessageDialog` display with more options, including image

```
JOptionPane.showMessageDialog(null,
        "Our Mesg", "Our Title",
        JOptionPane.INFORMATION_MESSAGE,
        new javax.swing.ImageIcon("cc.jpg"));
```



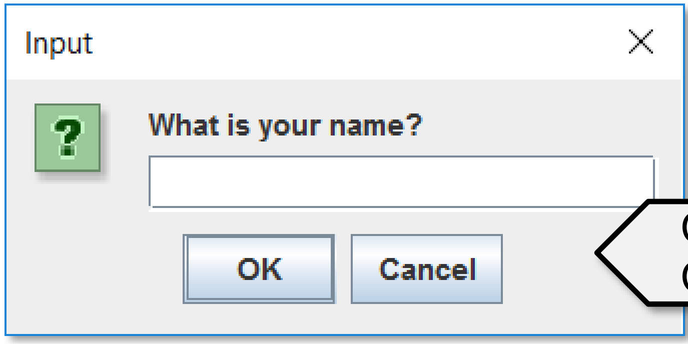
Our Mesg

Reference:
```
public static void showMessageDialog(Component parentComponent, Object message, String title,
        int messageType, Icon icon)
```

Class: `JOptionPane`
Method: `showInputDialog()`

- Use `showInputDialog` of the `JOptionPane` class is a simple way to let user input a *string*

```
String name = JOptionPane.showInputDialog
                    (null, "What is your name?");
```



Click `OK` to return a string typed.
Click `Cancel` to return `null` string.

- May convert string to other types, e.g. `int` as below

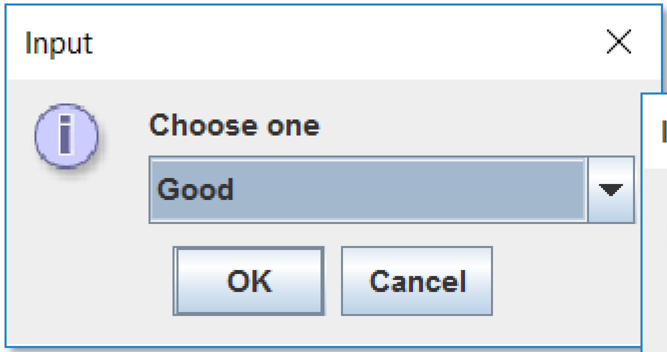
```
String inputStr = JOptionPane.showInputDialog
                    (null, "What is your age?");
int age = Integer.parseInt(inputStr);// convert String to int
```

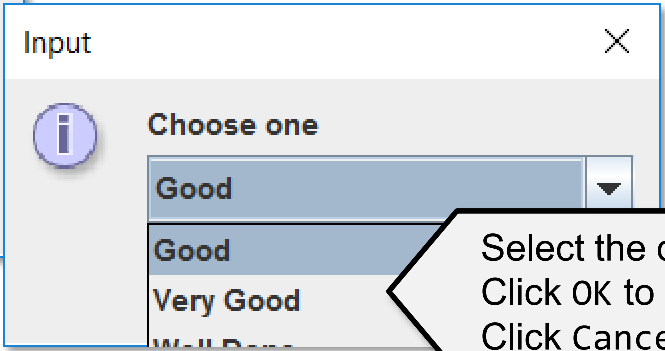
Ref: public static String showInputDialog(Component parentComponent, Object message)

Class: `JOptionPane`
Method: `showInputDialog()`

- May also use `showInputDialog` to show a dialog asking the user to select a String:

```
String[] optVal = { "Good",
                    "Very Good",
                    "Well Done" };    // option values
String selVal = (String) JOptionPane.showInputDialog(null,
               "Choose one", "Input",JOptionPane.INFORMATION_MESSAGE,
               null, optVal, optVal[0]);
```

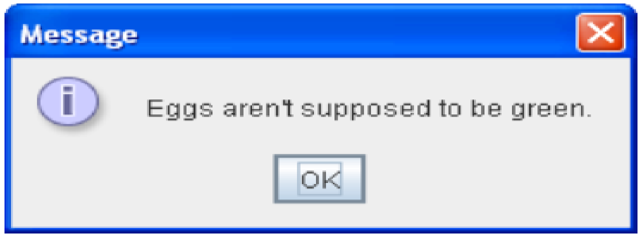




Select the desired option string (Object)
Click `OK` to return a string (Object) selected.
Click `Cancel` to return null string (Object).

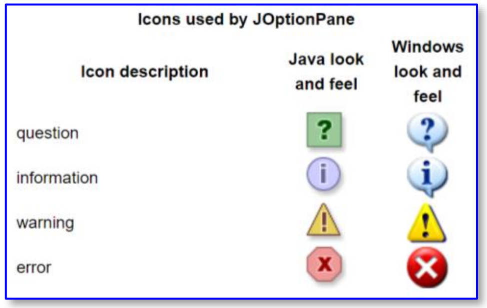
Ref: public static Object showInputDialog(Component parentComponent, Object message, String title, int messageType, Icon icon, Object[] selectionValues, Object initialSelectionValue)

Icons used by `JOptionPane`

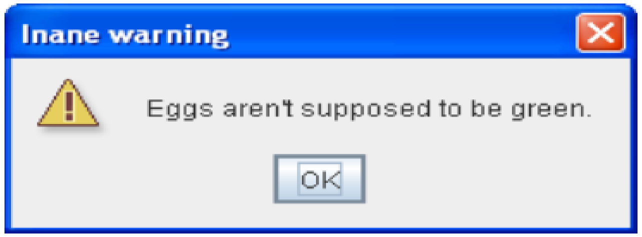

Message

Eggs aren't supposed to be green.

```
//default title and icon
JOptionPane.showMessageDialog(frame,
    "Eggs are not supposed to be green.");
```

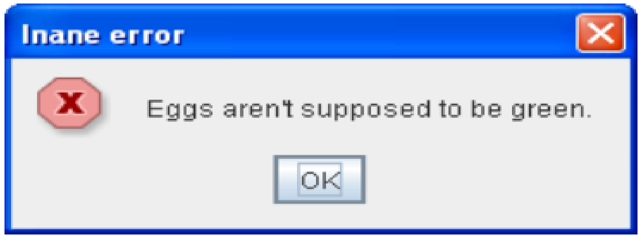

Icons used by JOptionPane

Icon description	Java look and feel	Windows look and feel
question	?	?
information	i	i
warning	!	!
error	x	x


Inane warning

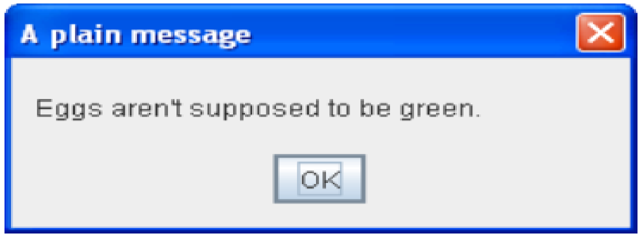
Eggs aren't supposed to be green.

```
//custom title, warning icon
JOptionPane.showMessageDialog(frame,
    "Eggs are not supposed to be green.",
    "Inane warning",
    JOptionPane.WARNING_MESSAGE);
```


Inane error

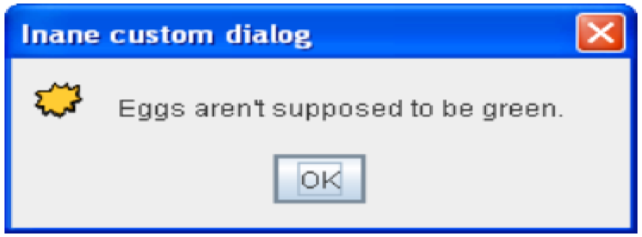
Eggs aren't supposed to be green.

```
//custom title, error icon
JOptionPane.showMessageDialog(frame,
    "Eggs are not supposed to be green.",
    "Inane error",
    JOptionPane.ERROR_MESSAGE);
```


A plain message

Eggs aren't supposed to be green.

```
//custom title, no icon
JOptionPane.showMessageDialog(frame,
    "Eggs are not supposed to be green.",
    "A plain message",
    JOptionPane.PLAIN_MESSAGE);
```


Inane custom dialog

Eggs aren't supposed to be green.

```
//custom title, custom icon
JOptionPane.showMessageDialog(frame,
    "Eggs are not supposed to be green.",
    "Inane custom dialog",
    JOptionPane.INFORMATION_MESSAGE,
    icon);
```

Common Problems in Java Programming

- **Division / evaluates to `int`**, if both operands are `int`

  `1/2` → `0` `(not 0.5)` but `1.0/2` → `0.5`

- **Omitting `void`** return type, in method without return

- **Omitting `return`** statement, in method with return

- **Wrong `return` type value**, in method with return

- **`break` works** with `switch` & loop statements, **not `if`**

- **Wrong `main()` header**, for program entry point method

  ```
  public static void main(String[] arg)
  ```

References

- This set of slides is only for educational purpose.

- Part of this slide set is referenced, extracted, and/or modified from the followings:

  - Deitel, P. and Deitel H. (2017) “Java How To Program, Early Objects”, 11ed, Pearson.
  - Liang, Y.D. (2017) “Introduction to Java Programming and Data Structures”, Comprehensive Version, 11ed, Prentice Hall.
  - Wu, C.T. (2010) “An Introduction to Object-Oriented Programming with Java”, 5ed, McGraw Hill.
  - Oracle Corporation, “Java Language and Virtual Machine Specifications” https://docs.oracle.com/javase/specs/
  - Oracle Corporation, “The Java Tutorials” https://docs.oracle.com/javase/tutorial/
  - Wikipedia, Website: https://en.wikipedia.org/