

---

CS1020 Lecture Note #5:

**Object Oriented Programming**  
**Part 3**

---

Yet more about OOP

# Lecture Note #5: OOP Part 3

## ■ Objectives:

- Using arrays
- Introducing the Vector class
- Introducing Exception handling

## ■ References:

### □ Array:

- Chapter 1, Section 1.1, pages 35 – 38

### □ Exception:

- Chapter 1, Section 1.6, page 64 to 72

# Lecture Note #5: OOP Part 3

## ■ Programs used in this lecture

- TestArray.java
- TestArrayUsage.java
- TestArraySwap.java
- TestArrayReturn.java
- TestArray2D.java
- TestVector.java
- TestException.java
- TestExceptionRetry.java

## ■ Download programs from module website:

- [http://www.comp.nus.edu.sg/~cs1020/2\\_resources/lectures.html](http://www.comp.nus.edu.sg/~cs1020/2_resources/lectures.html)

---

# 1 Array

---

A collection of homogeneous data

# 1. Array

- Array is the simplest way to store a collection of data of the same type (homogeneous)
- In Java, array is an object.

TestArray.java

```
class TestArray {
```

```
    public static void main(String[] args) {
```

```
        int[] arrayRef;
```

Syntax:

`datatype[] array_reference`

```
        arrayRef = new int[3];
```

Construct the array

```
        System.out.println(arrayRef.length);
```

`length` is a **public attribute** of array reference type

```
        arrayRef[0] = 100;
```

```
        arrayRef[1] = arrayRef[0] - 38;
```

```
        arrayRef[2] = 88;
```

After construction, array indexing works similarly to C

```
    }  
}
```

# 1. Array: Simple Usage

TestArrayUsage.java

```
class TestArrayUsage {
```

```
    public static void main(String[] args) {
```

```
        int[] arrayRef = { 100, 62, 88 };
```

Shortcut to construct the array and initialize the elements at the same time

```
        for (int element: arrayRef) {  
            System.out.println(element);  
        }
```

**Syntax (Enhanced For-Loop):**

```
for (datatype e: array_ref)
```

Go through all elements in the array. "e" automatically refers to the array element sequentially in each iteration.

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

Equivalent version

```
    }  
}
```

# 1. Array: As a parameter

- As the reference to the array is passed into a method:
  - Any modification of the element in the method will affect the actual array

TestArraySwap.java

```
class TestArraySwap {  
    public static void swap(int[] A, int i, int j) {  
        int temp = A[i];  
        A[i] = A[j];  
        A[j] = temp;  
    }  
  
    public static void main(String[] args) {  
        int[] arrayRef = { 100, 62, 88 };  
  
        swap(arrayRef, 0, 2);  
  
        for (int element: arrayRef) {  
            System.out.println(element);  
        }  
    }  
}
```

What is the output?

# 1. Array: As a return type

- Array can be returned from a method

TestArrayReturn.java

```
class TestArrayReturn {
    public static int[] makeArray(int size, int value) {
        int[] array = new int[size];

        for (int i = 0; i < array.length; i++)
            array[i] = value - i;
        return array;
    }

    public static void main(String[] args) {
        int[] arrayRef;

        arrayRef = makeArray(5, 99);

        for (int element: arrayRef) {
            System.out.println(element);
        }
    }
}
```

What is the output?

# 1. Array: Common Mistakes (1 / 3)

- length versus length()
  - To obtain length of a `String` object, we use the `length()` method
    - Example: `str.length()`
  - To obtain length (size) of an array, we use the `length` attribute
    - Example: `arr.length`
- Array index out of range
  - Beware of `ArrayIndexOutOfBoundsException` (exception is covered in section 3)

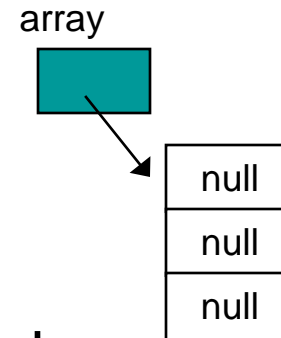
```
public static void main(String[] args) {
    int[] numbers = new int[10];
    . . .
    for (int i = 1; i <= numbers.length; i++)
        System.out.println(numbers[i]);
}
```

# 1. Array: Common Mistakes (2/3)

- When you have an array of objects, it's very common to forget to instantiate the array's objects.
- Programmers often instantiate the array itself and then think they're done – that leads to `java.lang.NullPointerException`

# 1. Array: Common Mistakes (3/3)

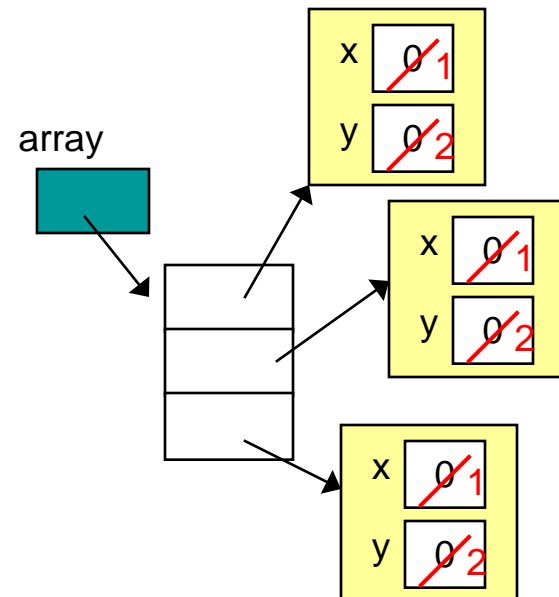
```
Point[] array = new Point[3];  
for (int i=0; i<array.length; i++) {  
    array[i].setLocation(1,2);  
}
```



There are no objects referred to by array[0], array[1], and array[2]!

Corrected code:

```
Point[] array = new Point[3];  
for (int i=0; i<array.length; i++) {  
    array[i] = new Point();  
    array[i].setLocation(1,2);  
}
```

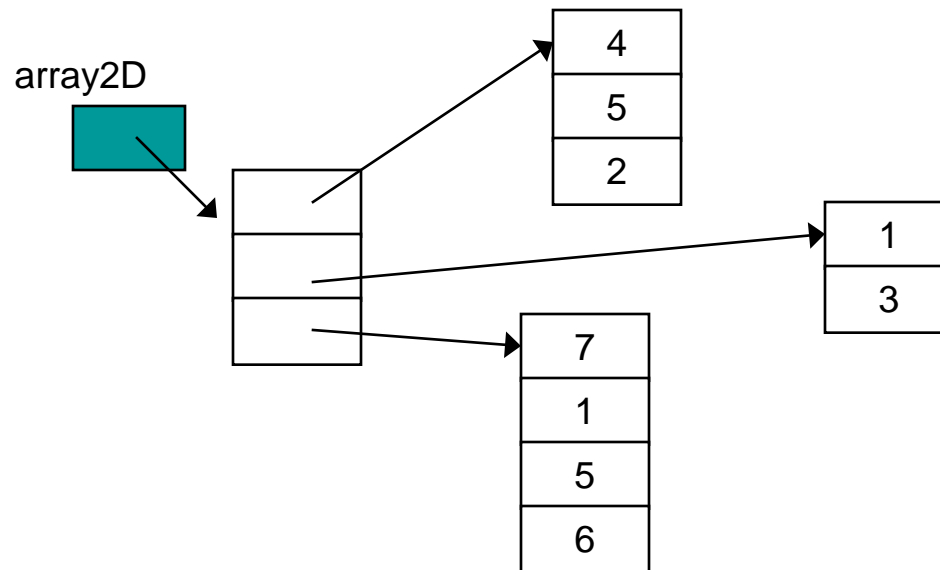


# 1. 2D Array (1/2)

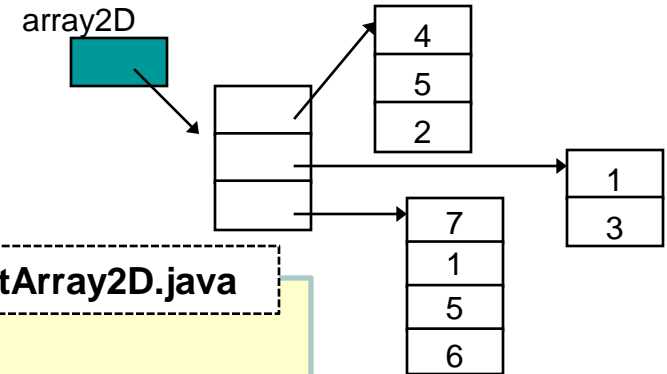
- A two-dimensional (2D) array is an array of array.
- This allows for rows of different lengths.

```
// an array of 12 arrays of int  
int[][] products = new int[12][];
```

```
int[][] array2D = { {4,5,2}, {1,3},  
                   {7,1,5,6} };
```



# 1. 2D Array (2/2)



TestArray2D.java

```
class TestArray2D {  
  
    public static void main(String[] args) {  
        int[][] array2D = { {4, 5, 2}, {1, 3}, {7, 1, 5, 6} };  
  
        System.out.println("array2D.length = " + array2D.length);  
        for (int i = 0; i < array2D.length; i++)  
            System.out.println("array2D[" + i + "].length = "  
                + array2D[i].length);  
  
        for (int row = 0; row < array2D.length; row++) {  
            for (int col = 0; col < array2D[row].length; col++)  
                System.out.print(array2D[row][col] + " ");  
            System.out.println();  
        }  
    }  
}
```

---

## 2 Vector

---

Dynamic-size array

## 2. Vector: Motivation

- Array has one major drawback:
  - Once initialized, the array size is **fixed**
  - Reconstruction is required if the array size changes
  - Note that Java has an **Array** class.
    - Check API documentation and explore it yourself
- Java offers a **Vector** class to provide:
  - Dynamic size
    - expands or shrinks automatically
  - Generic
    - allows any reference data types
  - Useful predefined methods
- Use array if the size is fixed, use **Vector** if the size may change.

## 2. Vector: API documentation (1/2)

PACKAGE

```
import java.util.Vector;
```

SYNTAX

```
//Declaration of a Vector reference  
Vector<E> myVector;
```

```
//Initialize a empty Vector object  
myVector = new Vector<E>;
```

### Commonly Used Method Summary

**boolean**

*isEmpty()*

Tests if this vector has no components.

**int**

*size()*

Returns the number of components in this vector.

## 2. Vector: API documentation (2/2)

### Commonly Used Method Summary (continued)

<b>boolean</b>	<b><i>add</i></b> ( <b>E</b> o) Appends the specified element to the end of this Vector.
<b>void</b>	<b><i>add</i></b> ( <b>int</b> index, <b>E</b> element) Inserts the specified element at the specified position in this Vector.
<b>E</b>	<b><i>remove</i></b> ( <b>int</b> index) Removes the element at the specified position in this Vector.
<b>boolean</b>	<b><i>remove</i></b> ( <b>Object</b> o) Removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged.
<b>E</b>	<b><i>get</i></b> ( <b>int</b> index) Returns the element at the specified position in this Vector.
<b>int</b>	<b><i>indexOf</i></b> ( <b>Object</b> elem) Searches for the first occurrence of the given argument, testing for equality using the equals method.
<b>boolean</b>	<b><i>contains</i></b> ( <b>Object</b> elem) Tests if the specified object is a component in this vector.

## 2. Vector: Example

TestVector.java

```
import java.util.Vector;

class TestVector {

    public static void main(String[] args) {

        Vector<String> courses;

        courses = new Vector<String>();

        courses.add("CS1020");
        courses.add(0, "CS1010");
        courses.add("CS2010");

        System.out.println(courses);
        System.out.println("At index 0: " + courses.get(0));

        if (courses.contains("CS1020"))
            System.out.println("CS1020 is in vector");

        courses.remove("CS1020");
        for (String c: courses)
            System.out.println(c);
    }
}
```

Vector class has a nice `toString()` method that prints all elements

The enhanced for-loop is applicable to vector object too!

---

# 3 Exception

---

Handling error events

### 3. Exception: Motivation (1/2)

- Consider the *factorial()* method:
  - What if the caller supply a negative parameter?

```
public static int factorial(int n) {  
    int ans = 1;  
    for (int i = 2; i <= n; i++) ans *= i;  
    return ans;  
}
```

What if n is negative?

- Should we terminate the program?

```
public static int factorial(int n) {  
    if (n < 0) {  
        System.out.println("n is negative");  
        System.exit(1);  
    }  
    //Other code not changed  
}
```

`System.exit()`  
terminates the program

- Note that the factorial method can be used by other programs
  - Difficult to cater to all possible scenarios

### 3. Exception: Motivation (2/2)

- Instead of deciding how to deal with an error, Java provides the **exception** mechanism:
  1. Indicate an error (**exception event**) has occurred
  2. Let the user decide how to handle the problem in a separate section of code specific for that purpose
  3. Crash the program if the error is not handled
- Exception mechanism consists of two components:
  - **Exception indication**
  - **Exception handling**
- Note that the preceding example of using exception for ( $n < 0$ ) is solely illustrative. Exceptions are more appropriate for harder to check cases such as when the value of  $n$  is too big, causing overflow in computation.

# 3. Exception Indication: Syntax (1/2)

- To indicate an error is detected:
  - Also known as **throwing an exception**
  - This allows the user to detect and handle the error

SYNTAX

```
throw ExceptionObject;
```

- Exception object must be:
  - An object of a class derived from **class Throwable**
  - Contain useful information about the error
- There are a number of useful predefined exception classes:
  - **ArithmeticException**
  - **NullPointerException**
  - **IndexOutOfBoundsException**
  - **IllegalArgumentException**

## 3. Exception Indication: **Syntax** (2/2)

- The different exception classes are used to **categorize the type of error**:
  - There is no major difference in the available methods

<b>Constructor</b>	
	<code>ExceptionClassName(String Msg)</code> Construct an exception object with the error message Msg
<b>Common methods for Exception classes</b>	
<code>String</code>	<code>getMessage()</code> Return the message stored in the object
<code>void</code>	<code>printStackTrace()</code> Print the calling stack

### 3. Exception Indication: Example

```
public static int factorial(int n)
    throws IllegalArgumentException {

    if (n < 0) {
        IllegalArgumentException exObj
            = new IllegalArgumentException(n + " is invalid!");
        throw exObj;
    }

    int ans = 1;
    for (int i = 2; i <= n; i++)
        ans *= i;
    return ans;
}
```

This declares that this method *may* throw ***IllegalArgumentException***

Actual act of throwing an exception (Note: no 's')  
Can be shorten to

```
throw new
    IllegalArgumentException(n + " is invalid!");
```

- Note:
  - A method can throw more than one type of exception

# 3. Exception Handling: Syntax

- As the user of a method that can throw exception(s):
  - It is your responsibility to handle the exception(s)
  - Also known as **exception catching**

```
try {  
    statement(s);  
}
```

```
// try block  
// exceptions might be thrown  
// followed by one or more catch block
```

```
catch (ExpClass1 obj1) {  
    statement(s);  
}  
catch (ExpClass2 obj2) {  
    statement(s);  
}
```

```
// a catch block  
// Do something about the exception  
// catch block for another type of  
    exception
```

```
finally {  
    statement(s);  
}
```

```
// finally block – for cleanup code
```

### 3. Exception Handling: Example

```
class TestException {  
  
    public static int factorial(int n)  
        throws IllegalArgumentException { //code not shown }  
  
    public static void main(String[] args) {  
        Scanner sc = new Scanner(System.in);  
        System.out.print("Enter n: ");  
        int input = sc.nextInt();  
  
        try {  
            System.out.println("Ans = " + factorial(input));  
        }  
        catch (IllegalArgumentException expObj) {  
            System.out.println(expObj.getMessage());  
        }  
    }  
}
```

We choose to print out the error message in this case. There are other ways to handle this error.

# 3. Exception: Execution Flow

TestException.java

```
public static int factorial(int n)
    throws IllegalArgumentException {
    System.out.println("Before Checking");
    if (n < 0) {
        throw new .....;
    }
    System.out.println("After Checking");
    //...Other code not shown
}
```

```
public static void main(String[] args) {
    // Other code not shown
    try {
        System.out.println("Before factorial()");
        System.out.println("Ans = " + factorial(input));
        System.out.println("After factorial()");
    } catch (IllegalArgumentException expObj){
        System.out.println("In Catch Block");
        System.out.println(expObj.getMessage());
    } finally {
        System.out.println("Finally!");
    }
}
```

## 3 Another Way?

- Modify the main method to:
  - Re-prompt the user if the user enters a negative number
- Reminder:
  - We are taking the perspective of the **user of `factorial()`**
  - So, the implementation of **`factorial()`** should not be changed by your solution

# 3 Another Way: Solution

TestExceptionRetry.java

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    int input;
    boolean retry = true;

    do {
        try {
            //Read input
            System.out.print("Enter n: ");
            input = sc.nextInt();
            System.out.println( "Ans = " + factorial(input));
            retry = false;

        } catch (IllegalArgumentException expObj){
            System.out.println(expObj.getMessage());
        }
    } while (retry);
}
```

a flag to determine whether we should re-prompt the user

If we can reach this point, then everything is ok, no need to retry

# 3 Exception: Guidelines

- When should we use exception?
  - If the error is not “expected”
    - Exceptional scenario that is uncommon or not anticipated
    - For expected scenarios, normal control flow statements are more appropriate
  - If the error cannot be resolved easily
    - Execution cannot continue in any way due to the error
  - If the error is serious
    - Exception forces the user to be aware of the error
- In a nutshell:
  - You should use it only with reasonable justification

# Summary

## Java Elements

### **Array:**

- Declaration and common usage

### **Vector:**

- Declaration and useful methods

### **Exception:**

- Raising Exception
- Handling Exception

---

End of file

---