OBJECT-ORIENTED SYSTEMS DESIGN: Arrays (2)

2022/04/13

Department of Computer Science,
Hanyang University

Taeuk Kim

Announcement

- **Midterm**
  - **Subjects:** All the contents covered in both theory and exercise sections so far.
    - From Chapter 1 to Chapter 6.
  - **Time:** 75 minutes (11:10 ~ 12:25).
  - **Place:** IT/BT 207.
    - You must come to the classroom **until 11:00.**
    - Another announcement will be made if changed.
  - Take along your ID card or student card.

- **HW1**
  - Today is the due!

Plan for Today

- **Theory (20 minutes)**
  - Chapter 6: Arrays

- **Exercise (50 minutes)**
  - **Review:** Chapter 5 (Defining Classes II)
  - **Practice**

Programming with Arrays

Chapter 6.3

Privacy Leaks with Array Instance Variables

- **If an accessor method does return the contents of an array, special care must be taken.**
  - Just as when an accessor returns a reference to any private object.
  - The example below will result in a *privacy leak*.

```
public double[] getArray()
{
  return anArray; // BAD!
}
```

Privacy Leaks with Array Instance Variables

- **The previous accessor method would simply return a reference to the array `anArray` itself.**
  - Instead, an accessor method should return a reference to a *deep copy* of the private array object.
  - Below, both `a` and `count` are instance variables of the class containing the `getArray` method.

```
public double[] getArray()
{
  double[] temp = new double[count];
  for (int i = 0; i < count; i++)
    temp[i] = a[i];
  return temp;
}
```

copy every element in the old array to the new array temp

Privacy Leaks with Array Instance Variables

- If a private instance variable is an array that has a class as its base type, then copies must be made of each class object in the array when the array is copied:

```
public ClassType[] getArray()
{
   ClassType[] temp = new ClassType[count];
   for (int i = 0; i < count; i++)
      temp[i] = new ClassType(someArray[i]);
   return temp;
}
```

Sorting an Array

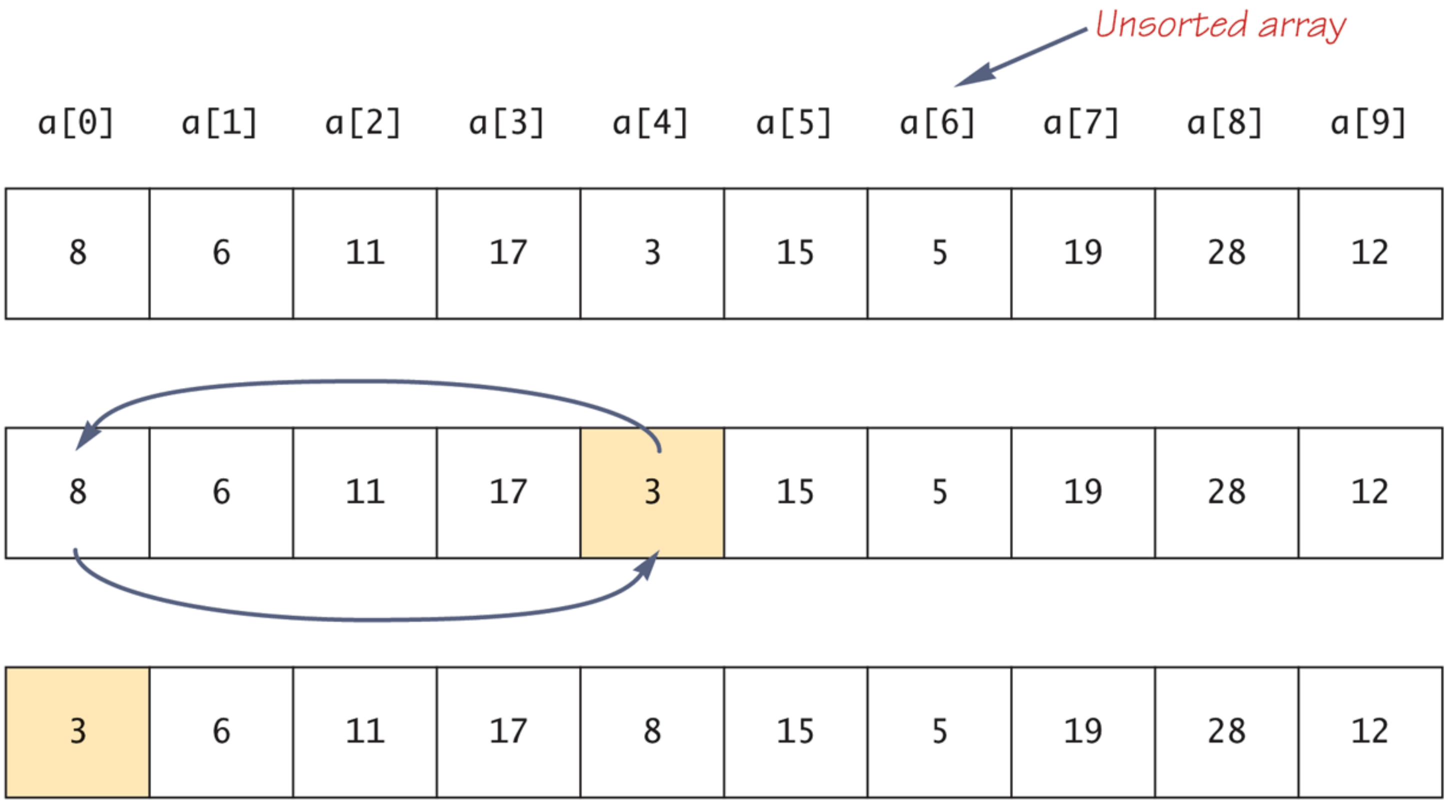
- A sort method takes in an array parameter `a`, and rearranges the elements in `a`, so that after the method call is finished, the elements of `a` are sorted in ascending order.

- A *selection sort* accomplishes this by using the following algorithm:

```
for (int index = 0; index < count; index++)
  Place the indexth smallest element in a[index]
```

Selection Sort (Part 1 of 2)

Display 6.10 **Selection Sort**



Unsorted array

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
8	6	11	17	3	15	5	19	28	12

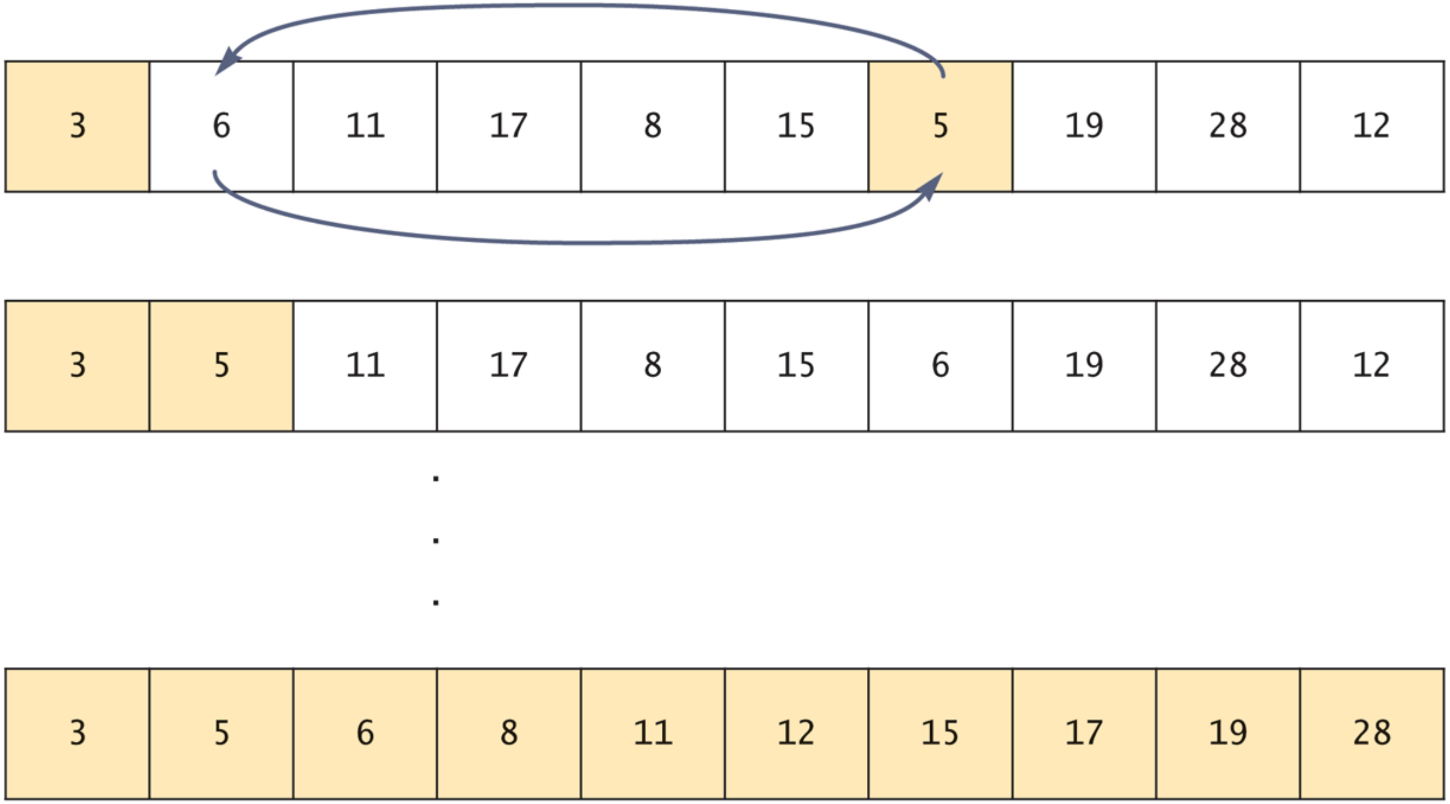
8	6	11	17	3	15	5	19	28	12

3	6	11	17	8	15	5	19	28	12

(continued)

Selection Sort (Part 2 of 2)

Display 6.10 **Selection Sort**



3	6	11	17	8	15	5	19	28	12

3	5	11	17	8	15	6	19	28	12

.

.

.

3	5	6	8	11	12	15	17	19	28

`SelectionSort` Class (Part 1 of 4)

```java
public class SelectionSort
{
    /**
     Precondition: numberUsed <= a.length;
                 The first numberUsed indexed variables have values.
     Action: Sorts a so that a[0] <= a[1] <= ... <= a[numberUsed - 1].
    */
    public static void sort(double[] a, int numberUsed)
    {
       int index, indexOfNextSmallest;
       for (index = 0; index < numberUsed - 1; index++)
       {//Place the correct value in a[index]:
          indexOfNextSmallest = indexOfSmallest(index, a, numberUsed);
          interchange(index,indexOfNextSmallest, a);
          //a[0] <= a[1] <= ...<= a[index] and these are the smallest
          //of the original array elements. The remaining positions
          //contain the rest of the original array elements.
       }
    }
```

SelectionSort Class (Part 2 of 4)

```
/**
 Returns the index of the smallest value among
 a[startIndex], a[startIndex+1], ... a[numberUsed - 1]
*/
private static int indexOfSmallest(int startIndex,
                                         double[] a, int numberUsed)
{
    double min = a[startIndex];
    int indexOfMin = startIndex;
    int index;
    for (index = startIndex + 1; index < numberUsed; index++)
      if (a[index] < min)
      {
          min = a[index];
          indexOfMin = index;
          //min is smallest of a[startIndex] through a[index]
      }
    return indexOfMin;
}
```

SelectionSort Class (Part 3 of 4)

```
    /**
     Precondition: i and j are legal indices for the array a.
     Postcondition: Values of a[i] and a[j] have been interchanged.
    */
    private static void interchange(int i, int j, double[] a)
    {
        double temp;
        temp = a[i];
        a[i] = a[j];
        a[j] = temp; //original value of a[i]
    }
}
```

SelectionSort Class (Part 4 of 4)

```java
public class SelectionSortDemo
{
  public static void main(String[] args)
  {
    double[] b = {7.7, 5.5, 11, 3, 16, 4.4, 20, 14, 13, 42};

    System.out.println("Array contents before sorting:");
    int i;
    for (i = 0; i < b.length; i++)
      System.out.print(b[i] + " ");
    System.out.println();

    SelectionSort.sort(b, b.length);

    System.out.println("Sorted array values:");
    for (i = 0; i < b.length; i++)
     System.out.print(b[i] + " ");
    System.out.println();
  }
}
```

Sample Dialogue

```
Array contents before sorting:
7.7 5.5 11.0 3.0 16.0 4.4 20.0 14.0 13.0 42.0
Sorted array values:
3.0 4.4 5.5 7.7 11.0 13.0 14.0 16.0 20.0 42.0
```

Multidimensional Arrays

Chapter 6.4

Multidimensional Arrays

- **Multidimensional arrays are declared and created in basically the same way as one-dimensional arrays.**
  - You simply use as many square brackets as there are indices.
  - Each index must be enclosed in its own brackets.

```
double[][] table = new double[100][10];
int[][][] figure = new int[10][20][30];
```

- **Multidimensional arrays may have any number of indices, but perhaps the most common number is two.**
  - Two-dimensional array can be visualized as a two-dimensional display with the first index giving the row, and the second index giving the column.

```
char[][] a = new char[5][12];
```

  - Note that, like a one-dimensional array, each element of a multidimensional array is just a variable of the base type (in this case, `char`).

Multidimensional Arrays

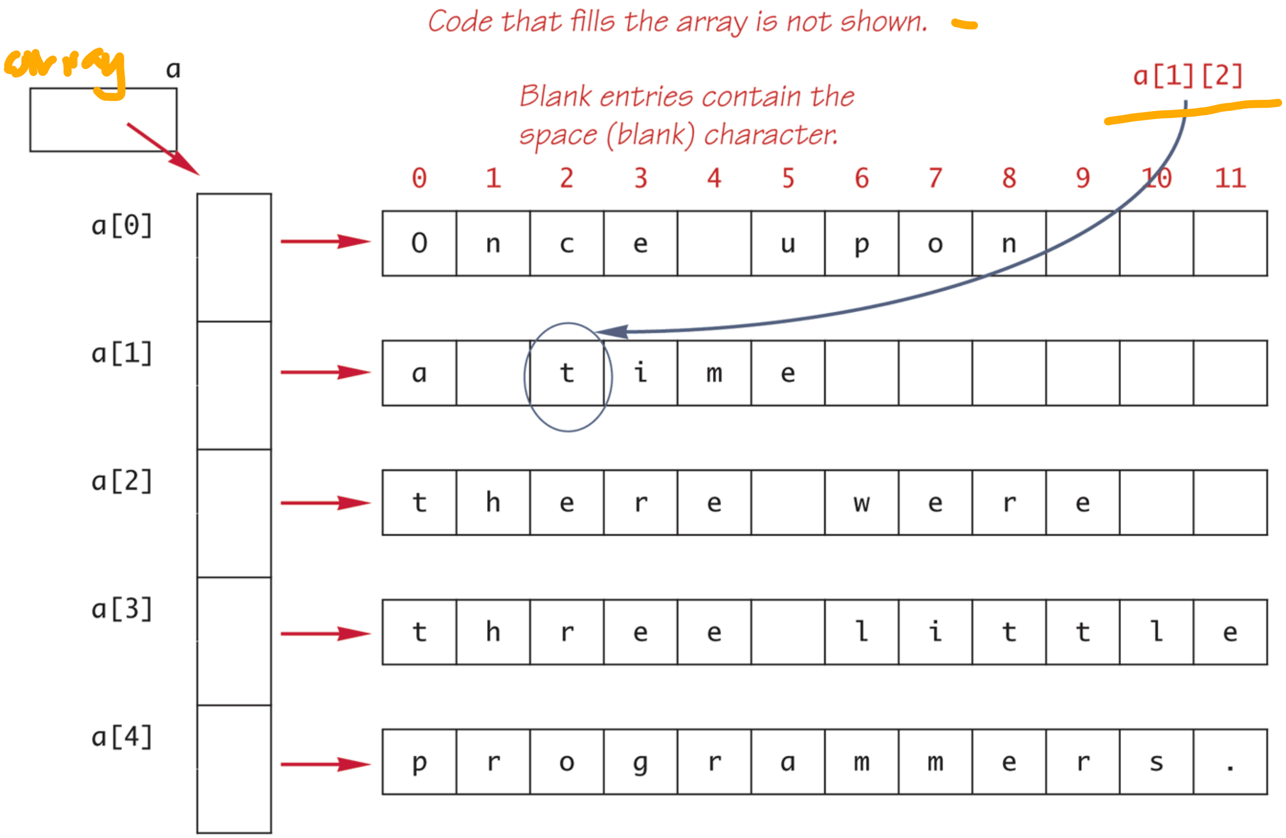
- **In Java, a two-dimensional array, such as `a`, is actually an array of arrays.**
  - The array `a` contains a reference to a one-dimensional array of size 5 with a base type of `char[]`.
  - Each indexed variable (`a[0]`, `a[1]`, etc.) contains a reference to a one-dimensional array of size 12, also with a base type of `char[]`.

- **A three-dimensional array is an array of arrays of arrays, and so forth for higher dimensions.**

Two-Dimensional Array as an Array of Arrays (Part 1 of 2)

Display 6.17 **Two-Dimensional Array as an Array of Arrays**

```
char[][] a = new char[5][12];
```

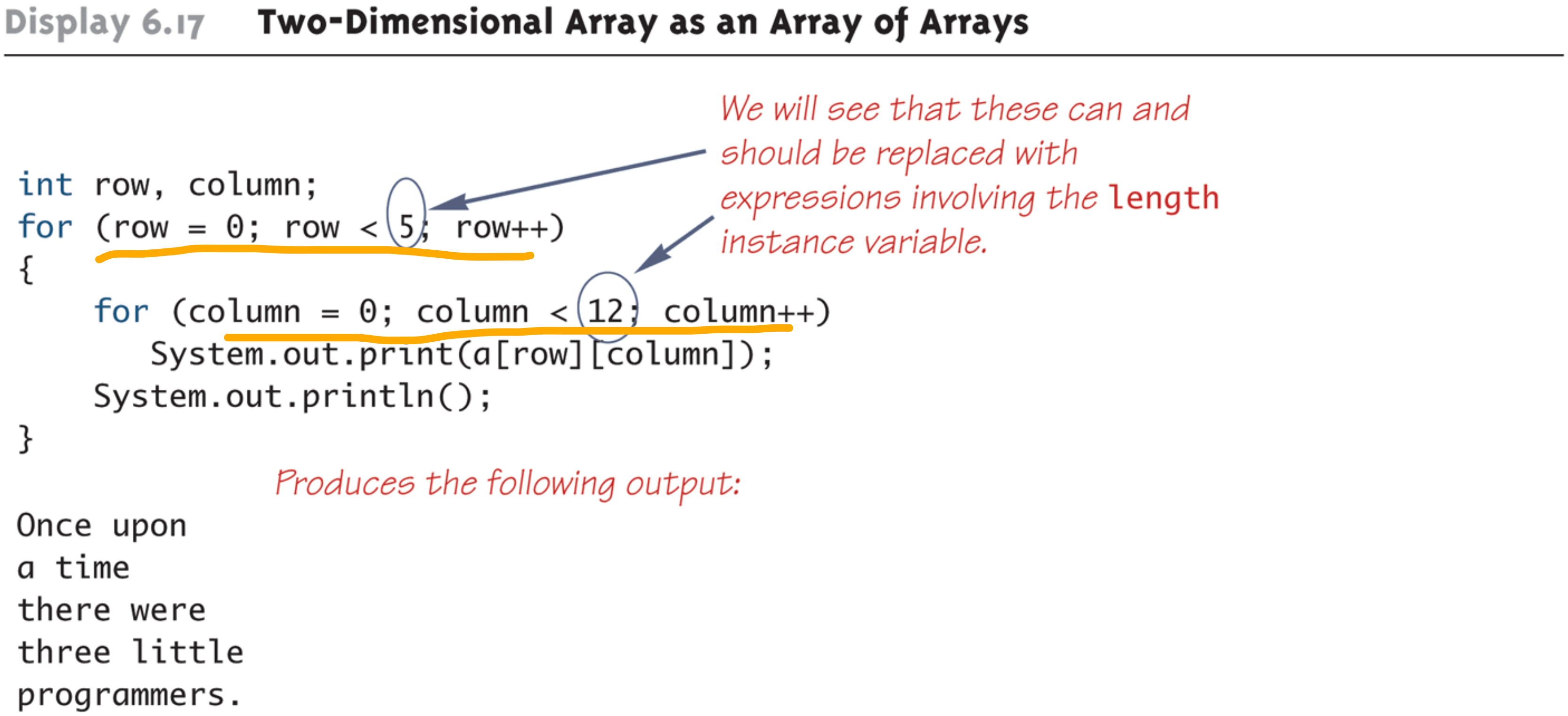


Code that fills the array is not shown.
array
a
a[1][2]
Blank entries contain the space (blank) character.

	0	1	2	3	4	5	6	7	8	9	10	11
a[0]	O	n	c	e		u	p	o	n			
a[1]	a		t	i	m	e						
a[2]	t	h	e	r	e		w	e	r	e		
a[3]	t	h	r	e	e		l	i	t	t	l	e
a[4]	p	r	o	g	r	a	m	m	e	r	s	.

(continued)

Two-Dimensional Array as an Array of Arrays (Part 2 of 2)



Display 6.17 **Two-Dimensional Array as an Array of Arrays**

We will see that these can and should be replaced with expressions involving the `length` *instance variable.*

```
int row, column;
for (row = 0; row < 5; row++)
{
    for (column = 0; column < 12; column++)
       System.out.print(a[row][column]);
    System.out.println();
}
```

Produces the following output:

```
Once upon
a time
there were
three little
programmers.
```

Using the `length` Instance Variable

- **The instance variable `length` does not give the total number of indexed variables in a two-dimensional array.**

  ```
  char[][] page = new char[30][100];
  ```

  - Because a two-dimensional array is actually an array of arrays, the instance variable `length` gives the number of first indices (or "rows") in the array.
    - `page.length` is equal to 30.
  - For the same reason, the number of second indices (or "columns") for a given "row" is given by referencing `length` for that *"row" variable.*
    - `page[0].length` is equal to 100.

- **The following program demonstrates how a nested `for` loop can be used to process a two-dimensional array.**
  - Note how each `length` instance variable is used.

  ```
  int row, column;
  for (row = 0; row < page.length; row++)
    for (column = 0; column < page[row].length; column++)
      page[row][column] = 'Z';
  ```

Multidimensional Array Parameters and Returned Values

- **Methods may have multidimensional array parameters.**
  - They are specified in a way similar to one-dimensional arrays.
  - They use the same number of sets of square brackets as they have dimensions.

```
public void myMethod(int[][] a)
{ . . . }
```

  - The parameter `a` is a two-dimensional array.

- **Methods may have a multidimensional array type as their return type.**
  - They use the same kind of type specification as for a multidimensional array parameter.

```
public double[][] aMethod()
{ . . . }
```

  - The method `aMethod` returns an array of `double`.

Conclusion

Chapter 6. Arrays

Summary: Arrays

- **6.3 Programming with Arrays**
  - Privacy Leaks with Array Instance Variables
  - Example: Sorting an Array

- **6.4 Multidimensional Arrays**
  - Multidimensional Array Basics
  - Using the length Instance Variable
  - Multidimensional Array Parameters and Returned Values

OBJECT-ORIENTED SYSTEMS DESIGN
[Exercise]: Defining Classes II

2022/04/13

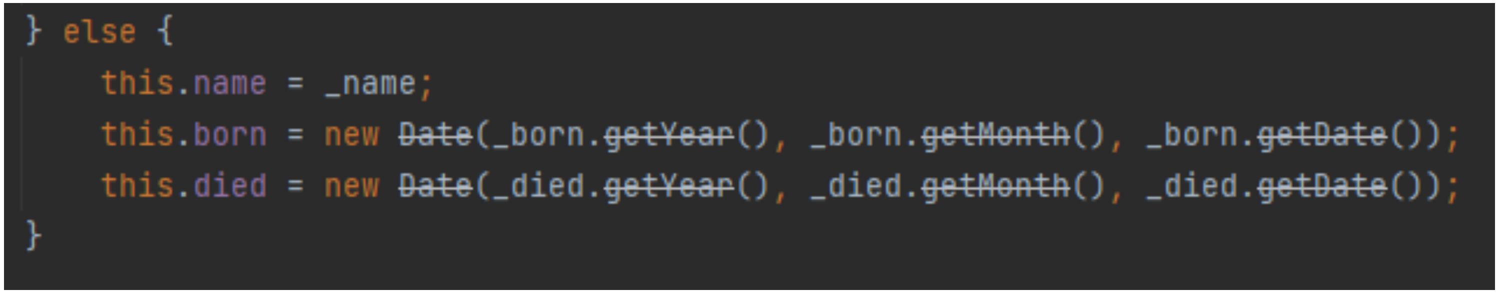
Department of Computer Science,
Hanyang University

Lecturer: Taeuk Kim
TA: Taejun Yoon, Seong Hoon Lim, Young Hyun Yoo

Before We Start,

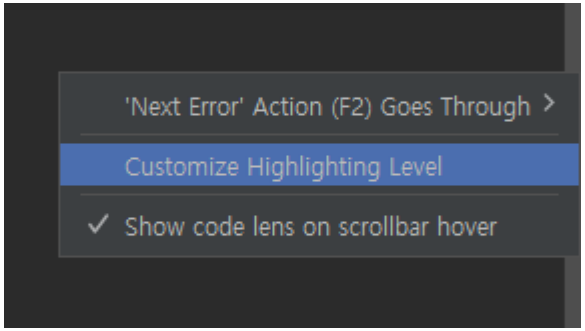
- **The `Date` class pre-defined in Java has many problems.**

- **Many of the methods are deprecated, annotated as below.**

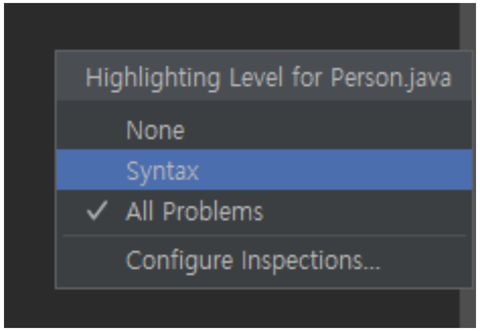


```
} else {
    this.name = _name;
    this.born = new ~~Date~~(_born.~~getYear~~(), _born.~~getMonth~~(), _born.~~getDate~~());
    this.died = new ~~Date~~(_died.~~getYear~~(), _died.~~getMonth~~(), _died.~~getDate~~());
}
```

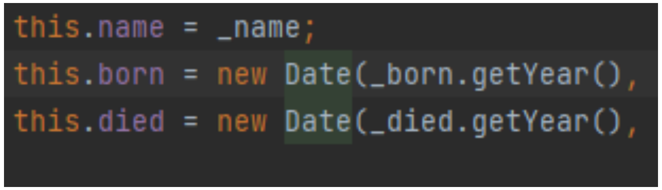
- **But it doesn't matter for our exercise session.**


'Next Error' Action (F2) Goes Through >
Customize Highlighting Level
✓ Show code lens on scrollbar hover

Right-click on the rightmost scroll bar.



Click "Customize Highlighting level" and Set to "Syntax".



```
this.name = _name;
this.born = new Date(_born.getYear(),
this.died = new Date(_died.getYear(),
```

Then, Strikethrough will disappear.

Construct a `Person` Class

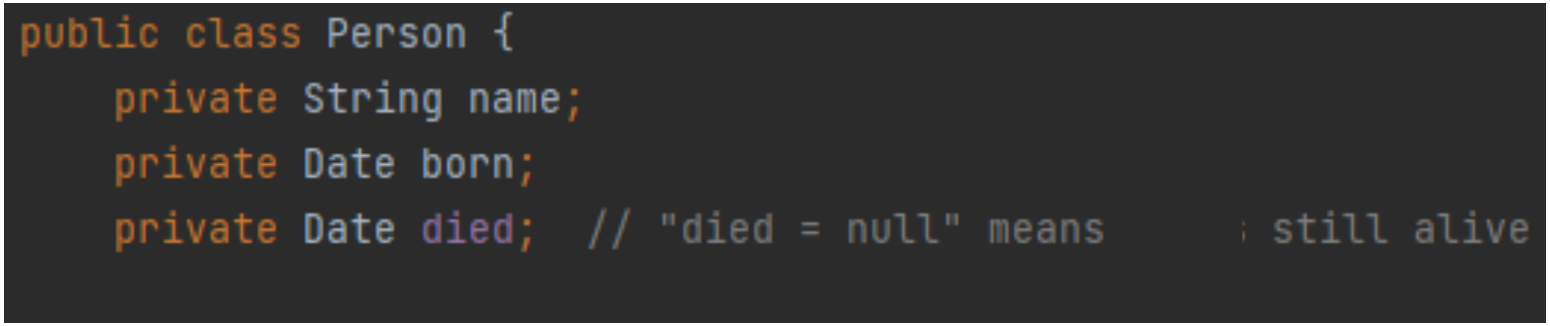
A class definition example

Constructing a `Person` class

- **The `Person` class contains 3 instance variables.**
  - `String name, Date born, Date died.`
  - These all variables should be private : accessors & mutators are required.
  - If `died` is `null`, it means he/she is still alive.
  - The date of birth(`born`) cannot be earlier than the date of death(`died`).

- **The `Person` class has 4 methods.**
  - `boolean consistent(Date born, Date died)`
    - Should be private and static: this function is used only for constructors.
    - Returns `true` if the birth and death dates are valid (i.e., birth ≤ death)
  - `String toString()`
    - Returns in the `String` format some information corresponding to the calling object.
  - `boolean equals(Person other)`
    - Returns `true` if the name and dates of birth and death of the calling object are equal to those of `other`.
  - `boolean datesMatch(Date date1, Date date2)`
    - Should be private and static : `datesMatch` is used only for `equals`.
    - Returns `true` if the two given dates are equal.

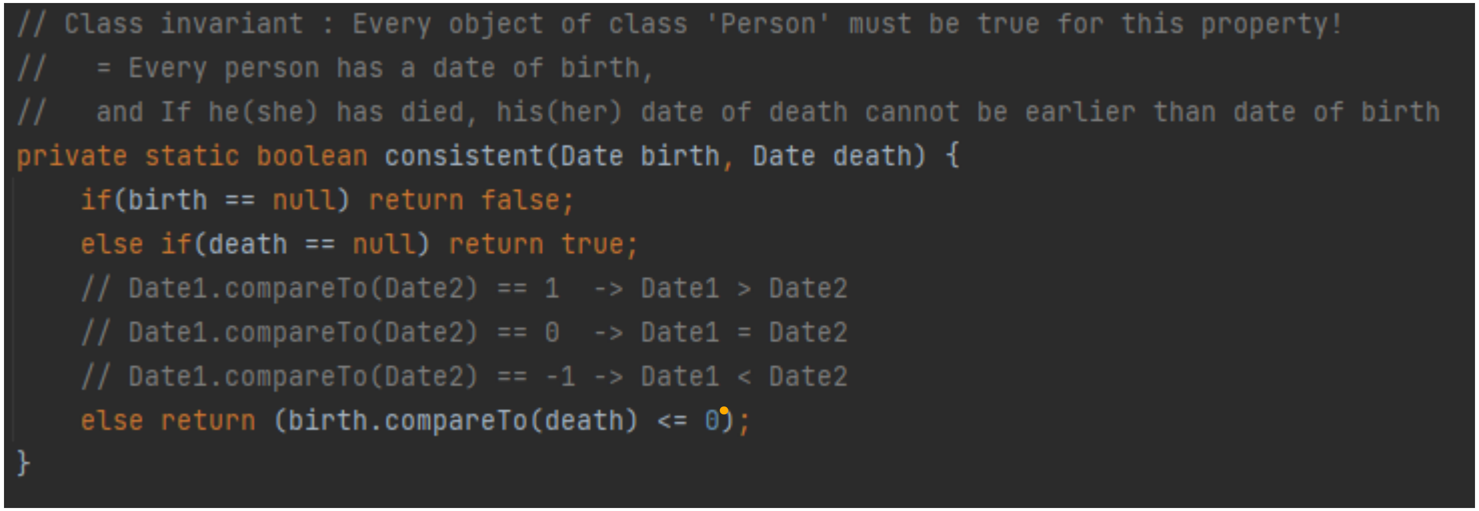
Constructing a `Person` class



```
public class Person {
    private String name;
    private Date born;
    private Date died;  // "died = null" means       still alive
```

- **Instance variable declaration**
  - **Encapsulation:** All instance variables are private.
  - The objects of the **`Person`** class type are only allowed to have access to their instance variables.
  - Other objects or methods should request access to or modification of these instance variables through accessor or mutator methods.

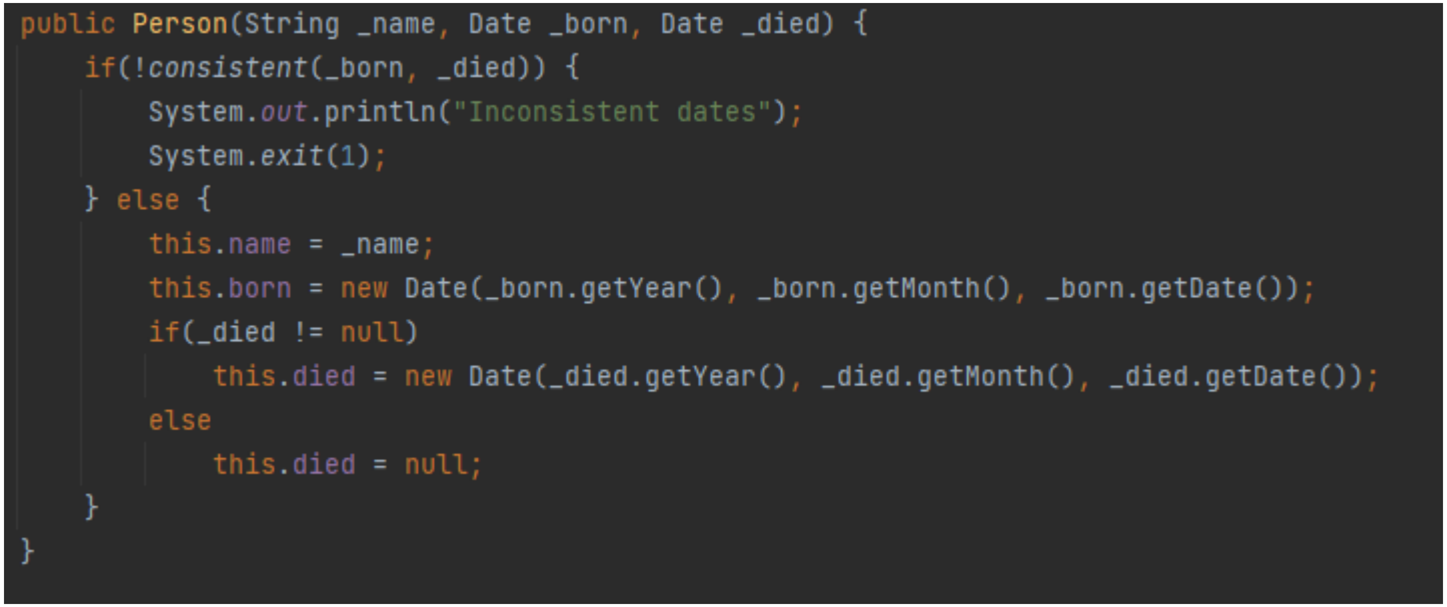
Constructing a `Person` class



```java
// Class invariant : Every object of class 'Person' must be true for this property!
//   = Every person has a date of birth,
//   and If he(she) has died, his(her) date of death cannot be earlier than date of birth
private static boolean consistent(Date birth, Date death) {
    if(birth == null) return false;
    else if(death == null) return true;
    // Date1.compareTo(Date2) == 1  -> Date1 > Date2
    // Date1.compareTo(Date2) == 0  -> Date1 = Date2
    // Date1.compareTo(Date2) == -1 -> Date1 < Date2
    else return (birth.compareTo(death) <= 0);
}
```

- **Class invariant**
  - Every object of the `Person` class must satisfy the properties specified in the function `consistent`.
    - Every person has his/her own date of birth.
    - The date of death cannot be earlier than the corresponding date of birth.

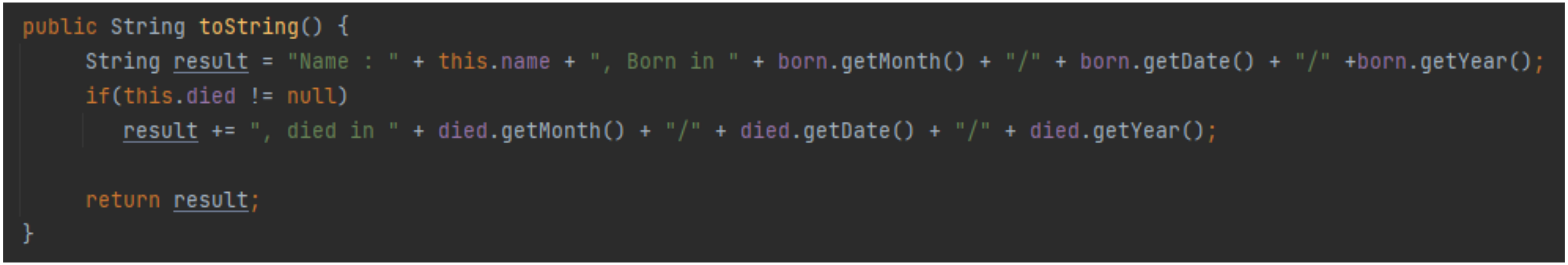
Constructing a `Person` class



```
public Person(String _name, Date _born, Date _died) {
    if(!consistent(_born, _died)) {
        System.out.println("Inconsistent dates");
        System.exit(1);
    } else {
        this.name = _name;
        this.born = new Date(_born.getYear(), _born.getMonth(), _born.getDate());
        if(_died != null)
            this.died = new Date(_died.getYear(), _died.getMonth(), _died.getDate());
        else
            this.died = null;
    }
}
```

- **Constructor**
  - Check whether a new object to be made satisfies the class invariant properties.
  - If it satisfies the class invariant, fill in the instance variables.
  - Note that the instance variables of the `Date` class type should be initialized with `new Date()`.

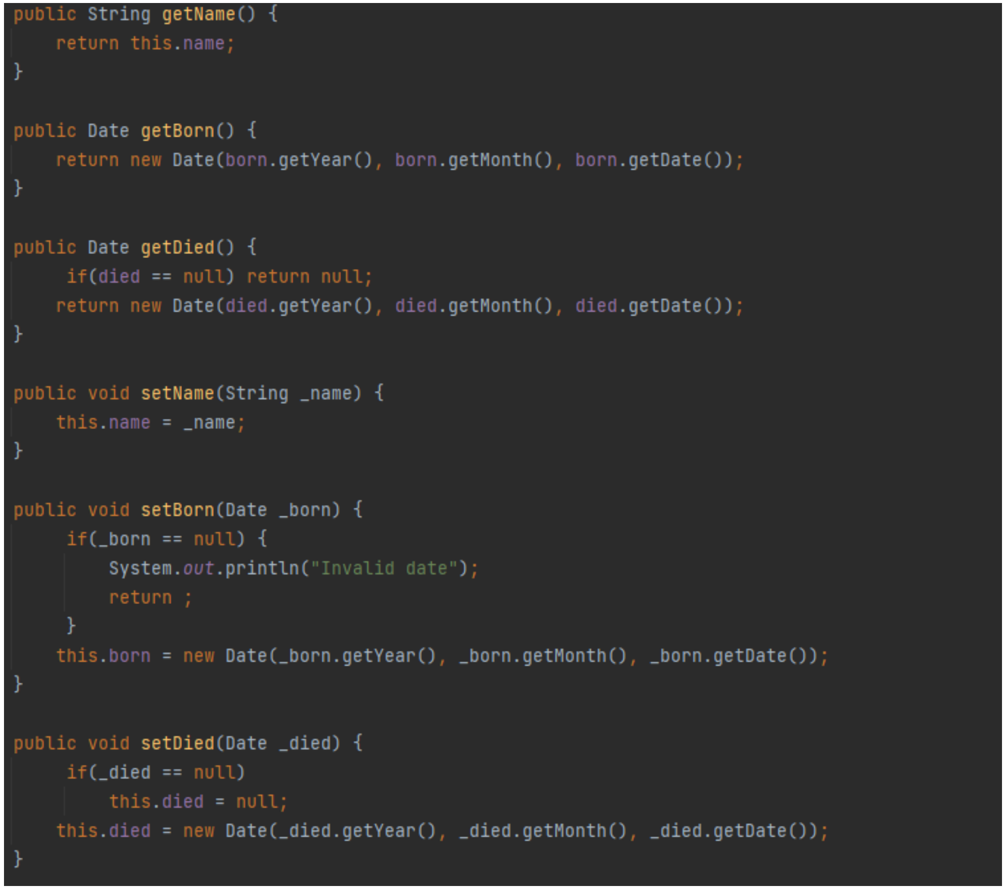
Constructing a `Person` class



```
public String toString() {
    String result = "Name : " + this.name + ", Born in " + born.getMonth() + "/" + born.getDate() + "/" +born.getYear();
    if(this.died != null)
        result += ", died in " + died.getMonth() + "/" + died.getDate() + "/" + died.getYear();

    return result;
}
```

- **The `toString()` method**
  - Returns in the `String` format some information corresponding to the calling object.

Constructing a `Person` class



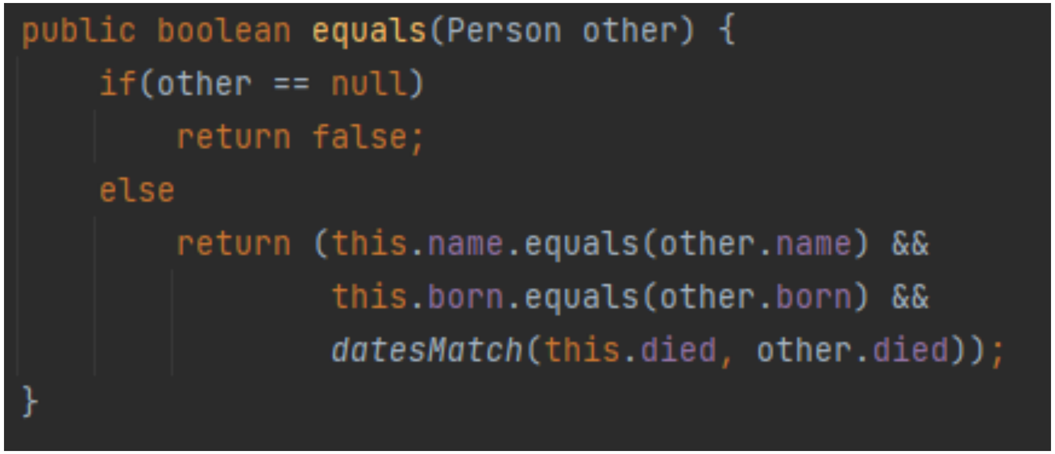
```java
public String getName() {
    return this.name;
}

public Date getBorn() {
    return new Date(born.getYear(), born.getMonth(), born.getDate());
}

public Date getDied() {
     if(died == null) return null;
    return new Date(died.getYear(), died.getMonth(), died.getDate());
}

public void setName(String _name) {
    this.name = _name;
}

public void setBorn(Date _born) {
     if(_born == null) {
         System.out.println("Invalid date");
         return ;
     }
    this.born = new Date(_born.getYear(), _born.getMonth(), _born.getDate());
}

public void setDied(Date _died) {
     if(_died == null)
         this.died = null;
    this.died = new Date(_died.getYear(), _died.getMonth(), _died.getDate());
}
```

`String` is immutable.
= Shallow copy is fine.

`Date` is mutable.
= Create an independent copy with the `new` command. (deep copy)

- **Accessors & Mutators**
  - Because our instance variables are private, other objects should request access to or modification of these instance variables through accessor or mutator methods.

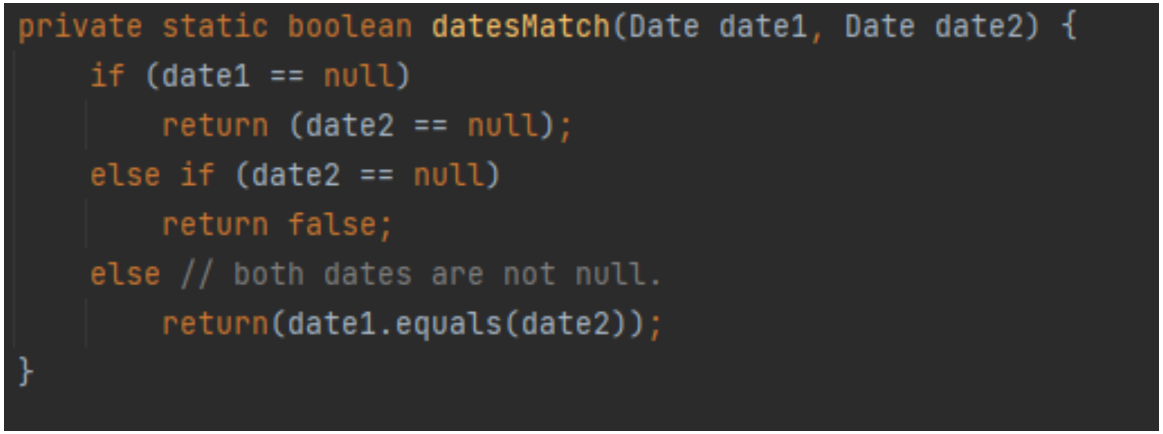
Constructing a `Person` class



```
public boolean equals(Person other) {
    if(other == null)
        return false;
    else
        return (this.name.equals(other.name) &&
                this.born.equals(other.born) &&
                datesMatch(this.died, other.died));
}
```

- **The `equals()` method**
  - Returns **`true`** if the name and dates of birth and death of the calling object are equal to those of **`other`**.
  - As the variable **`died`** can be assigned as **`null`**, we define a separate method called **`datesMatch`** for comparison.

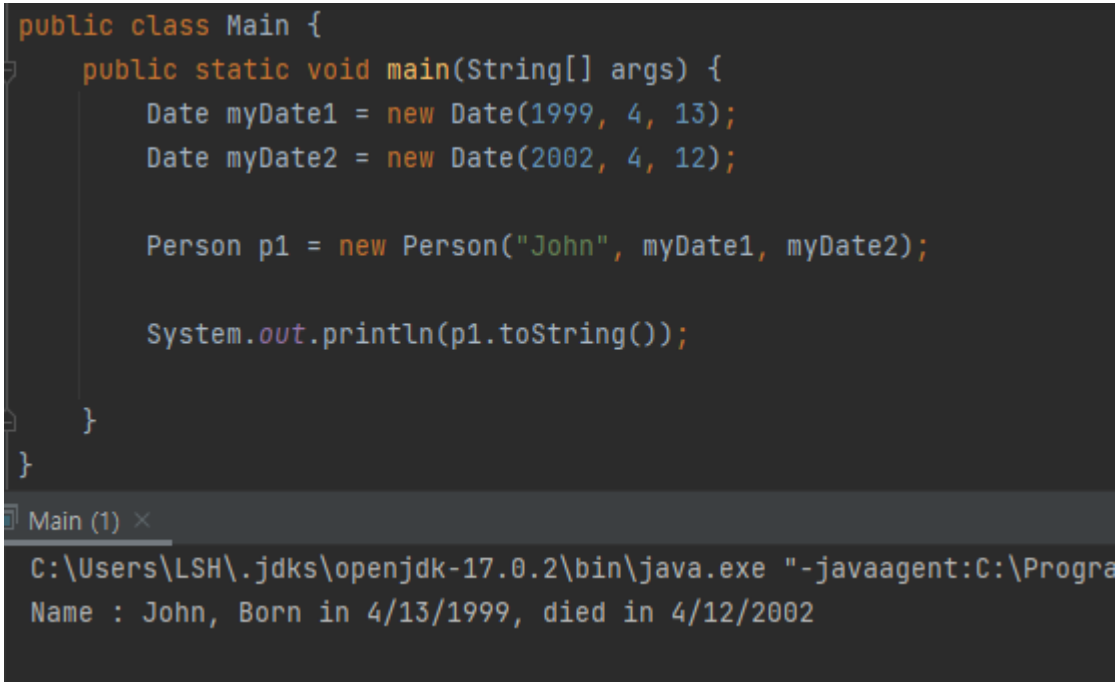
Constructing a `Person` class



```java
private static boolean datesMatch(Date date1, Date date2) {
    if (date1 == null)
        return (date2 == null);
    else if (date2 == null)
        return false;
    else // both dates are not null.
        return(date1.equals(date2));
}
```

- **The `datesMatch()` method**
  - We first deal with the cases where either `date1` or `date2` is `null`.
  - And then, we compare the two.

Constructing a `Person` class



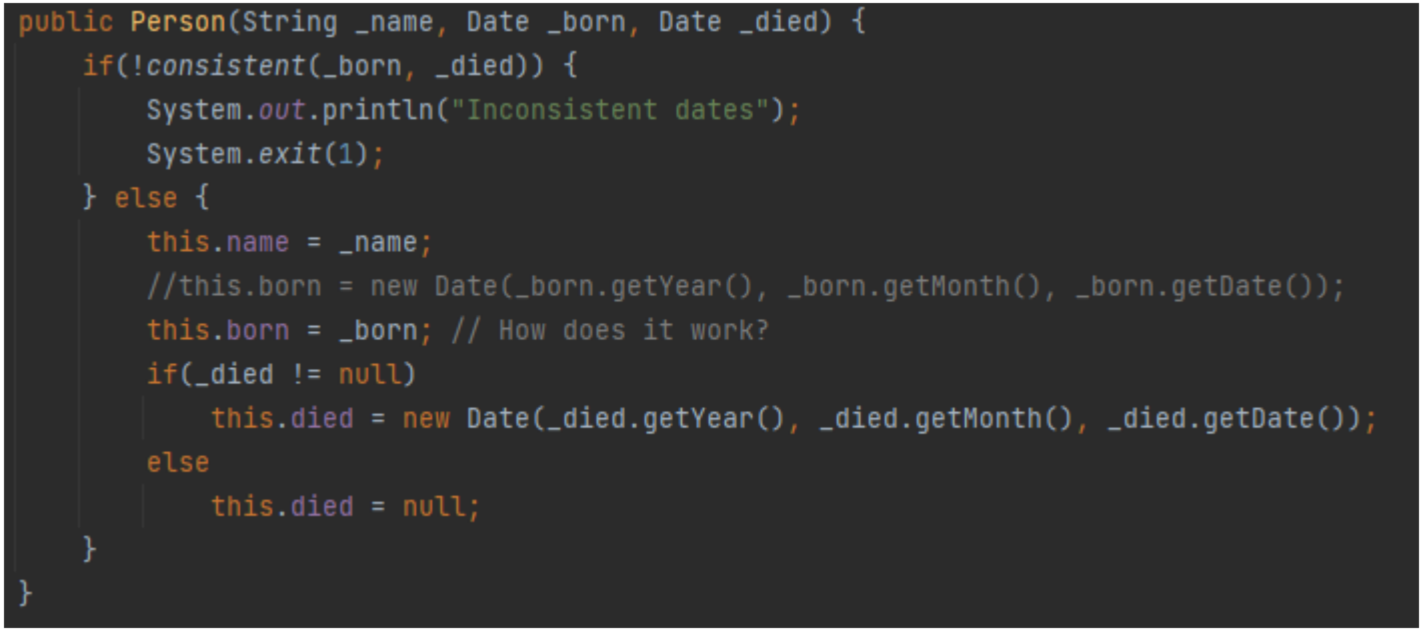
```
public class Main {
    public static void main(String[] args) {
        Date myDate1 = new Date(1999, 4, 13);
        Date myDate2 = new Date(2002, 4, 12);

        Person p1 = new Person("John", myDate1, myDate2);

        System.out.println(p1.toString());

    }
}
```

```
C:\Users\LSH\.jdks\openjdk-17.0.2\bin\java.exe "-javaagent:C:\Progra
Name : John, Born in 4/13/1999, died in 4/12/2002
```

An example of declaring a variable of the **`Person`** class type and printing its attributes.

Pitfalls : Deep copy vs. Shallow copy

- **If we change the constructor to…**



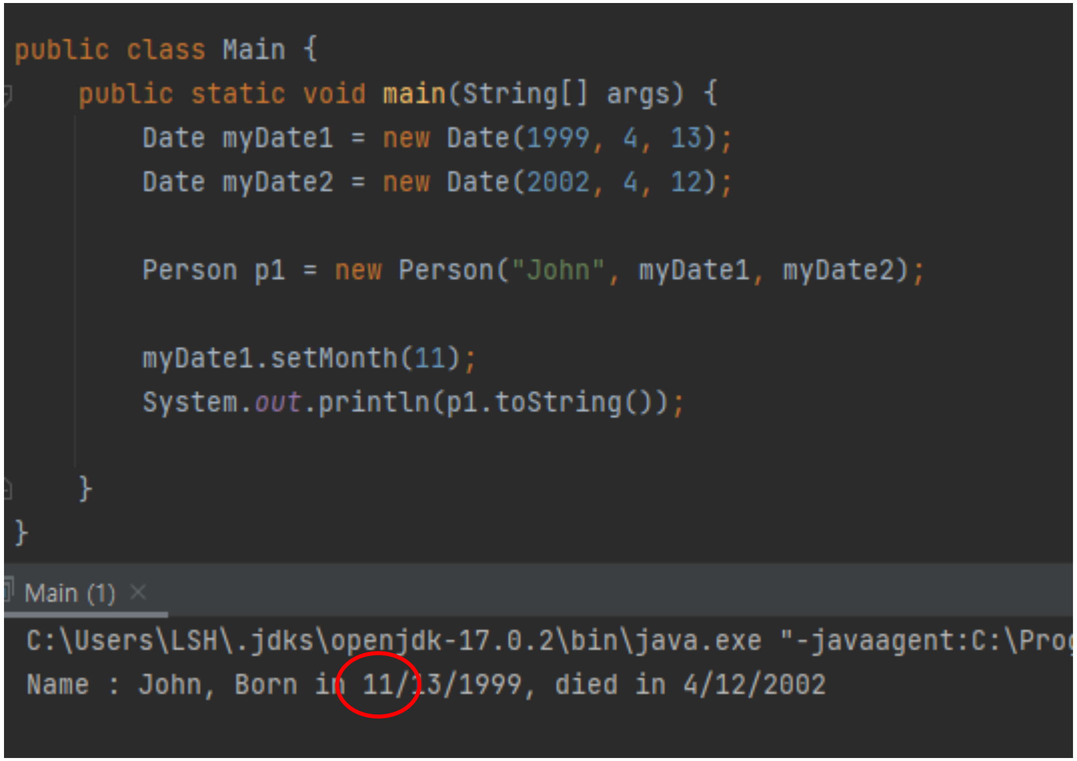
```
public Person(String _name, Date _born, Date _died) {
    if(!consistent(_born, _died)) {
        System.out.println("Inconsistent dates");
        System.exit(1);
    } else {
        this.name = _name;
        //this.born = new Date(_born.getYear(), _born.getMonth(), _born.getDate());
        this.born = _born; // How does it work?
        if(_died != null)
            this.died = new Date(_died.getYear(), _died.getMonth(), _died.getDate());
        else
            this.died = null;
    }
}
```

In this example, the assignment statement for the variable `born` has been changed to `this.born = _born`.

Would it work correctly?

Pitfalls : Deep copy vs. Shallow copy

- **Unexpected data modification (or privacy leak) might happen!**



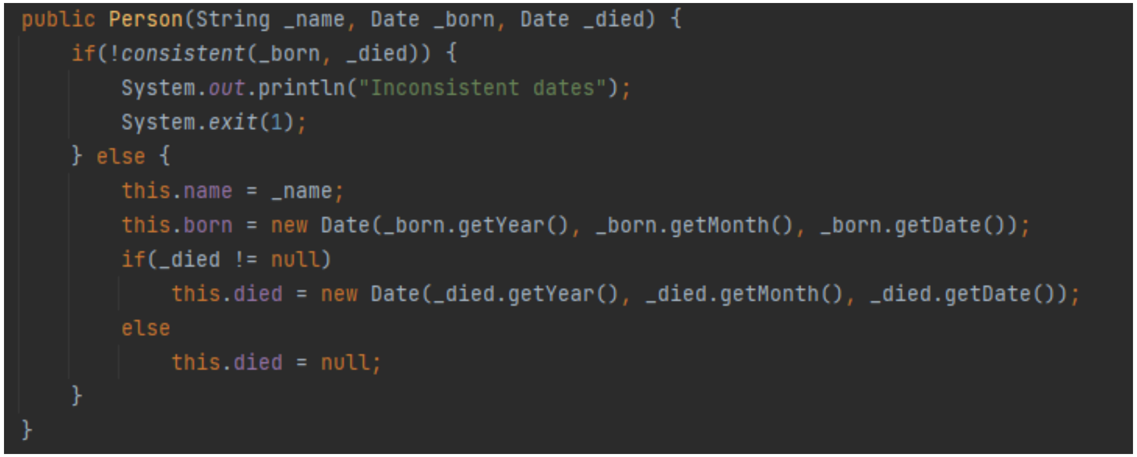
```java
public class Main {
    public static void main(String[] args) {
        Date myDate1 = new Date(1999, 4, 13);
        Date myDate2 = new Date(2002, 4, 12);

        Person p1 = new Person("John", myDate1, myDate2);

        myDate1.setMonth(11);
        System.out.println(p1.toString());

    }
}
```

```
C:\Users\LSH\.jdks\openjdk-17.0.2\bin\java.exe "-javaagent:C:\Prog
Name : John, Born in 11/13/1999, died in 4/12/2002
```

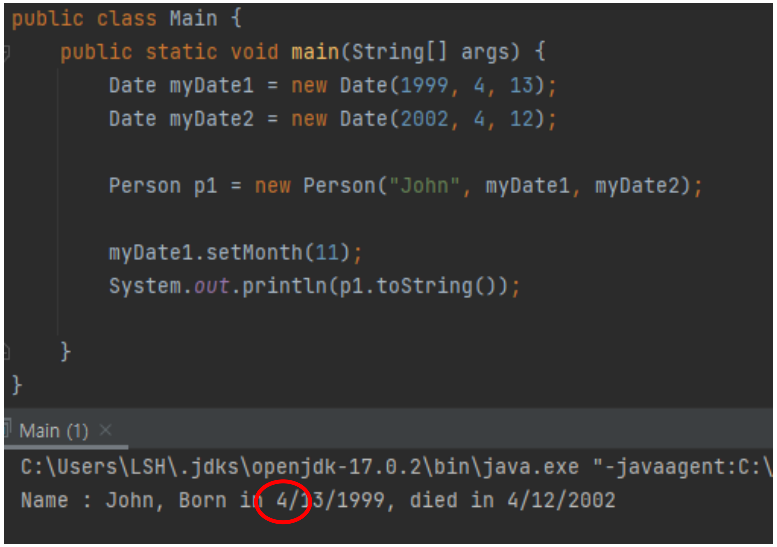
When we change the value of `myDate1`, which is an argument for defining `p1`, the birth date of `p1` is also changed! (= shallow copy)

Pitfalls : Deep copy vs. Shallow copy

- **The original correct form and its result**



```java
public Person(String _name, Date _born, Date _died) {
    if(!consistent(_born, _died)) {
        System.out.println("Inconsistent dates");
        System.exit(1);
    } else {
        this.name = _name;
        this.born = new Date(_born.getYear(), _born.getMonth(), _born.getDate());
        if(_died != null)
            this.died = new Date(_died.getYear(), _died.getMonth(), _died.getDate());
        else
            this.died = null;
    }
}
```



```java
public class Main {
    public static void main(String[] args) {
        Date myDate1 = new Date(1999, 4, 13);
        Date myDate2 = new Date(2002, 4, 12);

        Person p1 = new Person("John", myDate1, myDate2);

        myDate1.setMonth(11);
        System.out.println(p1.toString());

    }
}
```

```
C:\Users\LSH\.jdks\openjdk-17.0.2\bin\java.exe "-javaagent:C:\
Name : John, Born in 4/13/1999, died in 4/12/2002
```

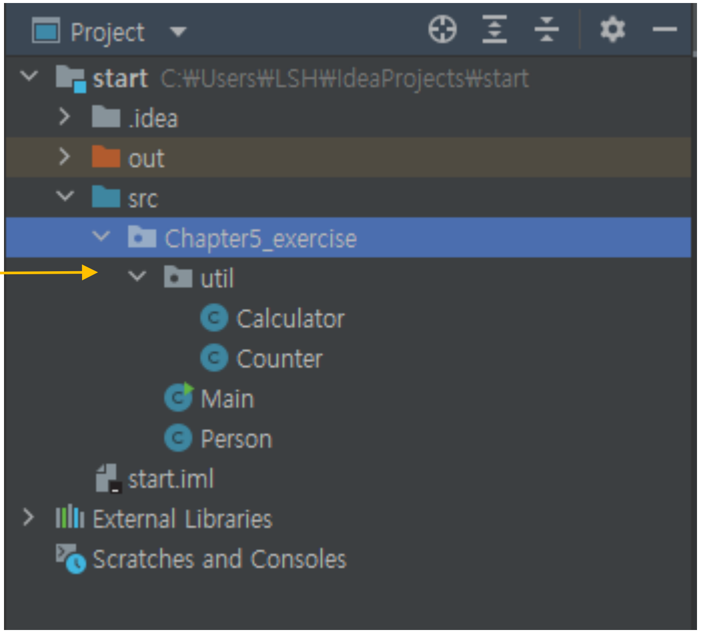
When we change the value of `myDate1`, which is an argument for defining `p1`, the birth date of `p1` **doesn't** change (=Deep copy).

Packages

Defining a Package

- **Define a package containing two utility classes.**
  - We attempt to construct a package called `util` inside the current directory (where our `Main` class is located in).
  - The `util` package provides two utility classes.
    - `Calculator`: a class for addition, subtraction, multiplication and division.
    - `Counter`: a class for counting numbers.

The `util` package is a subdirectory of `Chapter5_exercise` (or any folder that contains your `Main` class).



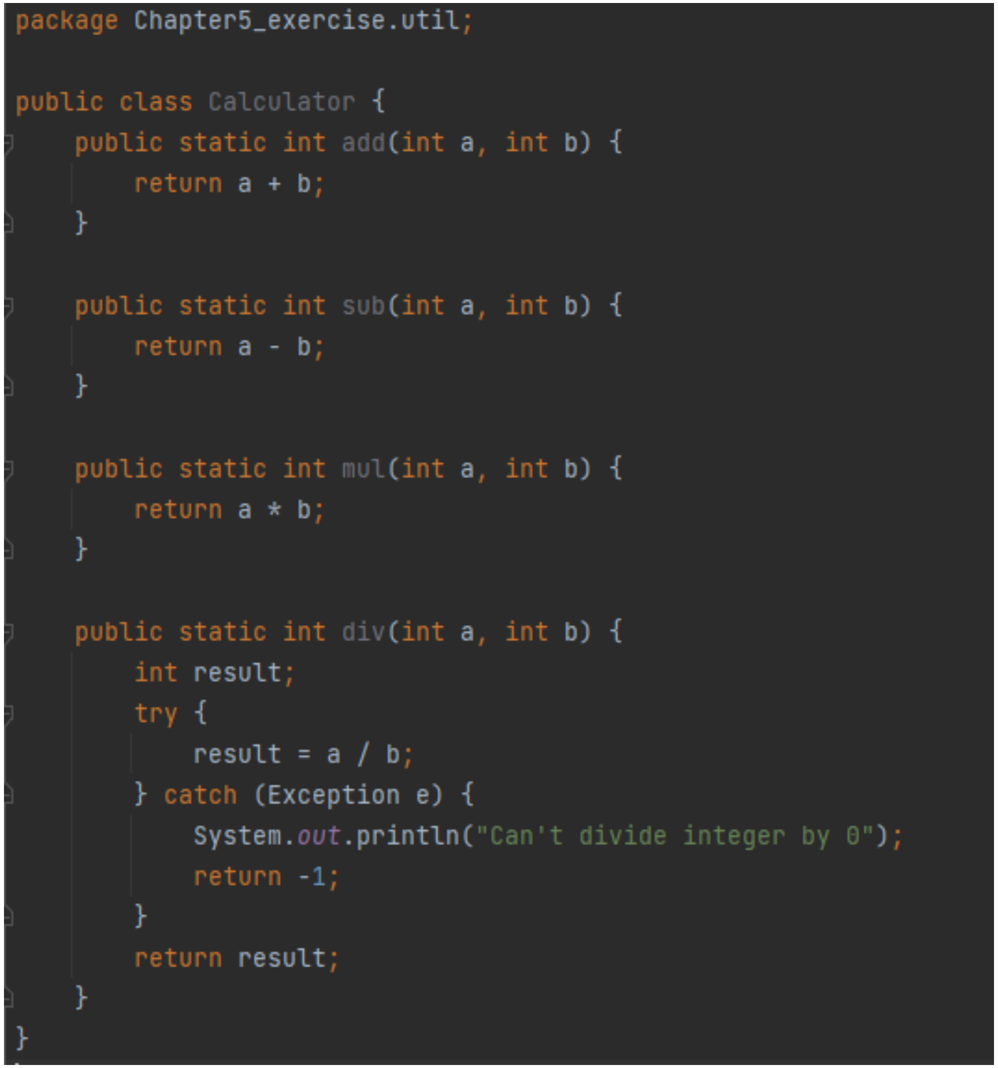
```
Project
start C:₩Users₩LSH₩IdeaProjects₩start
  .idea
  out
  src
    Chapter5_exercise
      util
        Calculator
        Counter
      Main
      Person
  start.iml
External Libraries
Scratches and Consoles
```

Specification of `Calculator` and `Counter`

- **Calculator class has 4 methods.**
  - `public static int add(int a, int b)`
    - `return a + b;`
  - `public static int sub(int a, int b)`
    - `return a – b;`
  - `public static int mul(int a, int b)`
    - `returns a * b;`
  - `public static int div(int a, int b)`
    - `returns a / b;`
    - If `b == 0`, print an error message and `return -1`.

- **Counter has 1 instance variable & method.**
  - `private static int counter`
    - Starts from 0.
    - When the `getCounter()` method is invoked, this value increases by 1.

Calculator Class Overview



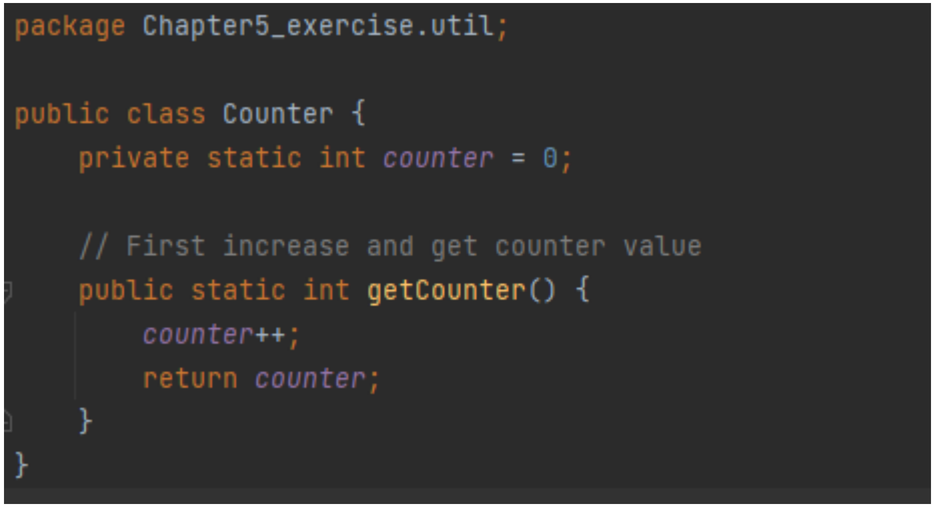
```
package Chapter5_exercise.util;

public class Calculator {
    public static int add(int a, int b) {
        return a + b;
    }

    public static int sub(int a, int b) {
        return a - b;
    }

    public static int mul(int a, int b) {
        return a * b;
    }

    public static int div(int a, int b) {
        int result;
        try {
            result = a / b;
        } catch (Exception e) {
            System.out.println("Can't divide integer by 0");
            return -1;
        }
        return result;
    }
}
```

Because all the methods are defined as **`static`**, we can use them without declaring an object of the **`Calculator`** class type.

Counter Class Overview

As the instance variable **counter** is defined as **static**, every **Counter** object shares the same variable.

In addition, as the method **getCounter()** is defined as **static**, we can utilize it without defining a separate an object.

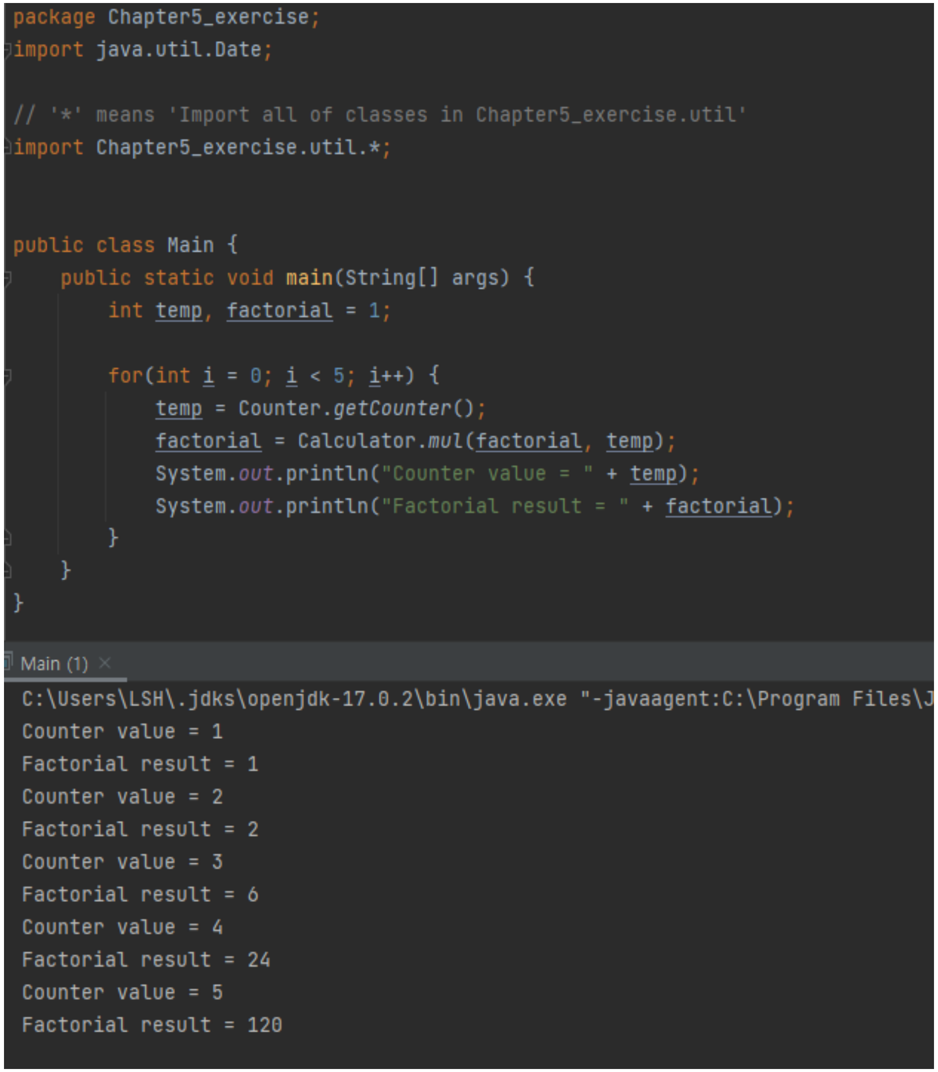


```java
package Chapter5_exercise.util;

public class Counter {
    private static int counter = 0;

    // First increase and get counter value
    public static int getCounter() {
        counter++;
        return counter;
    }
}
```

Use them in our package!

As the `util` package is imported here,

we can utilize the `Calculator` & `Counter` classes outside of the `util` package.



```
package Chapter5_exercise;
import java.util.Date;

// '*' means 'Import all of classes in Chapter5_exercise.util'
import Chapter5_exercise.util.*;


public class Main {
    public static void main(String[] args) {
        int temp, factorial = 1;

        for(int i = 0; i < 5; i++) {
            temp = Counter.getCounter();
            factorial = Calculator.mul(factorial, temp);
            System.out.println("Counter value = " + temp);
            System.out.println("Factorial result = " + factorial);
        }
    }
}
```

```
C:\Users\LSH\.jdks\openjdk-17.0.2\bin\java.exe "-javaagent:C:\Program Files\J
Counter value = 1
Factorial result = 1
Counter value = 2
Factorial result = 2
Counter value = 3
Factorial result = 6
Counter value = 4
Factorial result = 24
Counter value = 5
Factorial result = 120
```

Practice

Exercise/WeekN/Main.java,
/Person.java,
/util/AgeCalculator.java

Practice 1

- **Define your own `Person` class following the given code snippets.**

Practice 2

- **Define a `AgeCalculator` class.**
  - It should be located in the `util` package (directory).
  - Location: (your git_repo)/…/WeekN/**util/AgeCalculator.java**

- **It should support 2 static methods.**
  - `int calAge(Person p)`
    - Returns the age of the person `p`.
    - We follow the international rule (만 나이).
  - `int isOlder(Person p1, Person p2)`
    - Returns 1 when `p1`'s age > `p2`'s age.
    - Returns 0 when `p1`'s age == `p2`'s age.
      - e.g., **1999.06.18** and **1999.12.31** returns 0 (the same age 22).
      - e.g., **1999.02.18** and **1999.12.31** returns 1 (`p1`'s age : 23).
    - Returns -1 when `p1`'s age < `p2`'s age.

- **Finally,**
  - Declare two random people using the `Person` class in your main method.
  - Then, compute and compare the ages of the two using the `AgeCalculator` class.

Time for Practice

Get it started, and ask TAs if you are in a trouble.