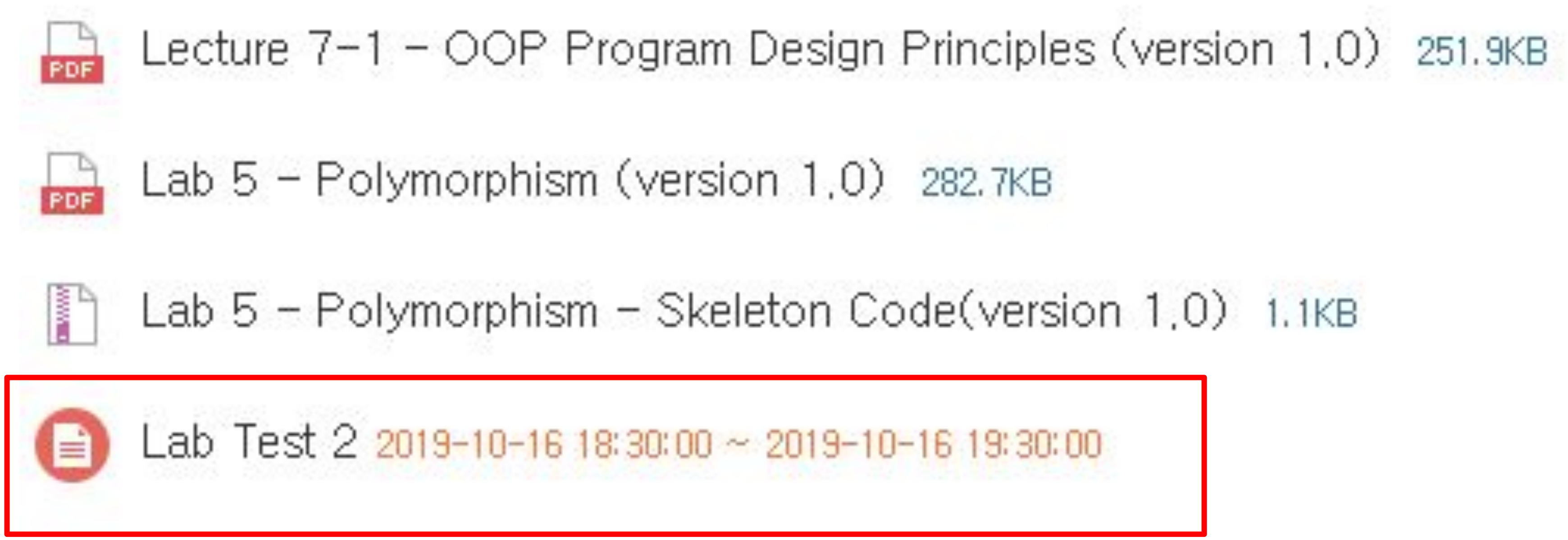
Lab Test 2

Lab Test

- It will be a closed-book, individually-done test.
- Duration is <u>60 minutes</u>.
- There will be a total of 3 questions in the lab test.
- We provide a skeleton code (`.java` file) for each question in the test.
- After finishing up your test, please zip all three `.java` files (not `.class` files) into one file and name it as your student number.

Lab Test

- For the Lab Test, download ‘LabTest02_skeleton.zip’ from ‘Lab Test 2’, not ‘Lab 5 skeleton code’.



Lecture 7-1 - OOP Program Design Principles (version 1.0) 251.9KB

Lab 5 - Polymorphism (version 1.0) 282.7KB

Lab 5 - Polymorphism - Skeleton Code(version 1.0) 1.1KB

Lab Test 2 2019-10-16 18:30:00 ~ 2019-10-16 19:30:00

Lab Test

1. Download the lab test zip file (LabTest02_skeleton.zip) from eTL.
2. Unzip the .zip file.
3. Open IntelliJ IDEA and make a new project.
4. Drag and drop the three `.java` skeleton codes to the src folder in the IDEA project.
5. Fill out the codes.
6. Zip the three completed .java files into one .zip file and name it as your student number (201X-XXXXX.zip).
7. Upload the .zip file to eTL.

Problem 1

Problem Description :

Write a **class *TextBook*** which inherits the class *Book*. The Behavior of the method *read* should be different. It should first print the title as the class *Book*, and in second line print the "Unwilling" instead of "Reading".

```
class Book{
 protected String title;
 Book(String s){   this.title = s;   }
 void read(){
   System.out.println("Title : "+ title);
   System.out.println("Reading");
 }
}
```

class *Book* code

Problem 1

Restrictions :

- The length of input string N satisfies N > 1.
- A newline can either be printed at the end of “Unwilling” or not.
- “Reading” should be substituted to “Unwilling”, and nothing else will be allowed.
- Code for the class *Book* and *main*() should not be modified.

```
Scanner scanner = new Scanner(System.in);
String title = scanner.nextLine();
Book book = new TextBook(title);
if(book instanceof Book){
  System.out.println("Class : " + book.getClass());
}
book.read();
```

main() method

Problem 1

Input/Output Example :

input	Introduction to Algorithms
output	Class : class TextBook Title : Introduction to Algorithms Unwilling
input	Information Theory
output	Class : class TextBook Title : Information Theory Unwilling

Problem 1

Input/Output Example :

input	Jane Eyre

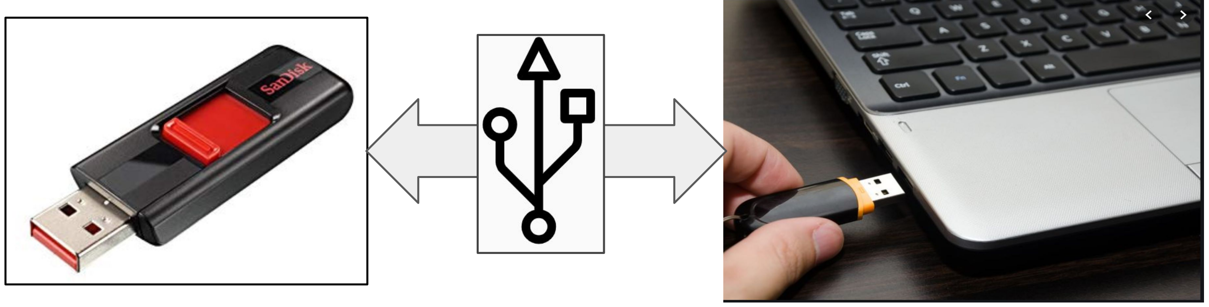
output

```
Class : class TextBook
Title : Jane Eyre
Unwilling
```

Input/Output Description :

A single entire line of the input is considered as the title of a textbook. If input is “Introduction to Algorithms”, the title of the textbook is “Introduction to Algorithms”. The class name output should be “class TextBook”. Subsequently, it will print the title of the textbook and “Unwilling” in the next line.

Problem 2

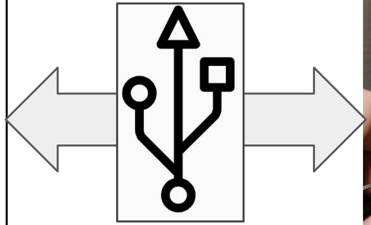


Problem Description :

Suppose we have the USB flash drive (class *FlashDrive*) which stores data with Object types in its storage.

Device driver is the module that helps the interaction between the hardware devices like USB flash drive and the computer (Operating Systems).

Problem 2



Problem Description :

In this problem, we are going to implement the USB flash drive driver class “***USBDriver***” **only** by implementing the interface ***Driver***. The spec of the interface Driver is as below.

You should implement the Driver utilizing the class instance and methods of the class *FlashDrive*, not making your own storage. Read the definition of class *FlashDrive* carefully and implement the *USBDriver* utilizing it.

```
interface Driver{
    public Object access(int index);
    public void save(Object o, int index);
}
```

Problem 2

Problem Description :

the method *access* :

- Inputs the index of the storage to access/lookup, and outputs the object that was stored at the exact index of the object.

the method *save* :

- Inputs the object to save in the storage and the index of the storage specifying where to store it, and outputs nothing. (void)
- The object is stored at the given index of the storage of the *FlashDrive* class instance.

Problem 2

Restrictions :

- For the method access, the test input index will always be lower than the capacity of the flash drive, and will always be nonnegative.
- For the method save, the test input index will always be lower than the capacity of the flash drive, and will always be nonnegative.
- You need to define <u>the class constructor</u> which inputs one and only argument of type *FlashDrive*.
- You should not modify the code of class *FlashDrive*, interface *Driver,* and *test()* method.

Problem 2

class *FlashDrive* code

```java
class FlashDrive{
  private Object[] storage;
  private int capacity;
  FlashDrive(int capacity){
    if(capacity > 0){
      this.capacity = capacity;
      storage = new Object[this.capacity];
    }
    else{
      throw new IllegalArgumentException(
      "USB Flash drive storage cannot be smaller than 0");
      }
  }
  public Object load(int index){
    driveStateCheck(index);
    return storage[index];
  }
```

Problem 2

```
  public void store(Object o, int index){
    driveStateCheck(index);
    storage[index] = o;
  }
  private void driveStateCheck(int index){
    if(!stateValid(index)){
      throw new IllegalArgumentException("Failed");
    }
  }
  private boolean stateValid(int index){
    return  storage != null && index < capacity && index >= 0;
  }
}
```

Problem 2

- *sample()* method and *test()* method in Problem2 class are provided to help you test your implementation.
  - *test()* method deals with the streaming input, and a similar version will be used for the actual evaluation of your submission.
  - These are **already implemented** in the skeleton code.

- The main purpose of this problem is to test your skills in understanding the code and its logics. So read the code of the class *FlashDrive* carefully, and you will find it much easier than it seems.

Problem 2

Sample() method as main()

```java
FlashDrive flash = new FlashDrive(10);
Driver usbDriver = new USBDriver(flash);
// Store
usbDriver.save("Document",1);
usbDriver.save(77.0,3);
usbDriver.save(true,4);
// Access
System.out.println(usbDriver.access(3));
System.out.println(usbDriver.access(1));
System.out.println(usbDriver.access(4));
// Overwrite
usbDriver.save("Site",1);
System.out.println(usbDriver.access(1));
// Not stored
System.out.println(usbDriver.access(5));
// Delete and attempt access
usbDriver.save(null,4);
System.out.println(usbDriver.access(4));
```

output

```
77.0
Document
true
Site
null
null
```

Problem 2

Input/Output description of *test()* method

- The first line of the input is two integers : the capacity of the usb flash drive storage, and the total number of following input commands.
- Following lines are driver input commands. About the first word, if it is ‘s’, then it is save instruction. If it is ‘a’, then it is the access instruction.
    - On the save instruction, the second word is a String data to save, and the third word is the index of the storage to save.
    - On the access instruction, the second word is an index of the storage to access and load.

Problem 2

- Input/Output Example of *test()* method

Input

```
10 9
s Doc1 1
s Doc2 3
s Doc3 4
a 3
a 1
a 4
s Pic 1
a 1
a 2
```

Output

```
Doc2
Doc1
Doc3
Pic
null
```

Input

```
10 7
s a 1
a 1
s b 2
a 2
s c 3
a 3
a 9
```

Output

```
a
b
c
null
```

Input

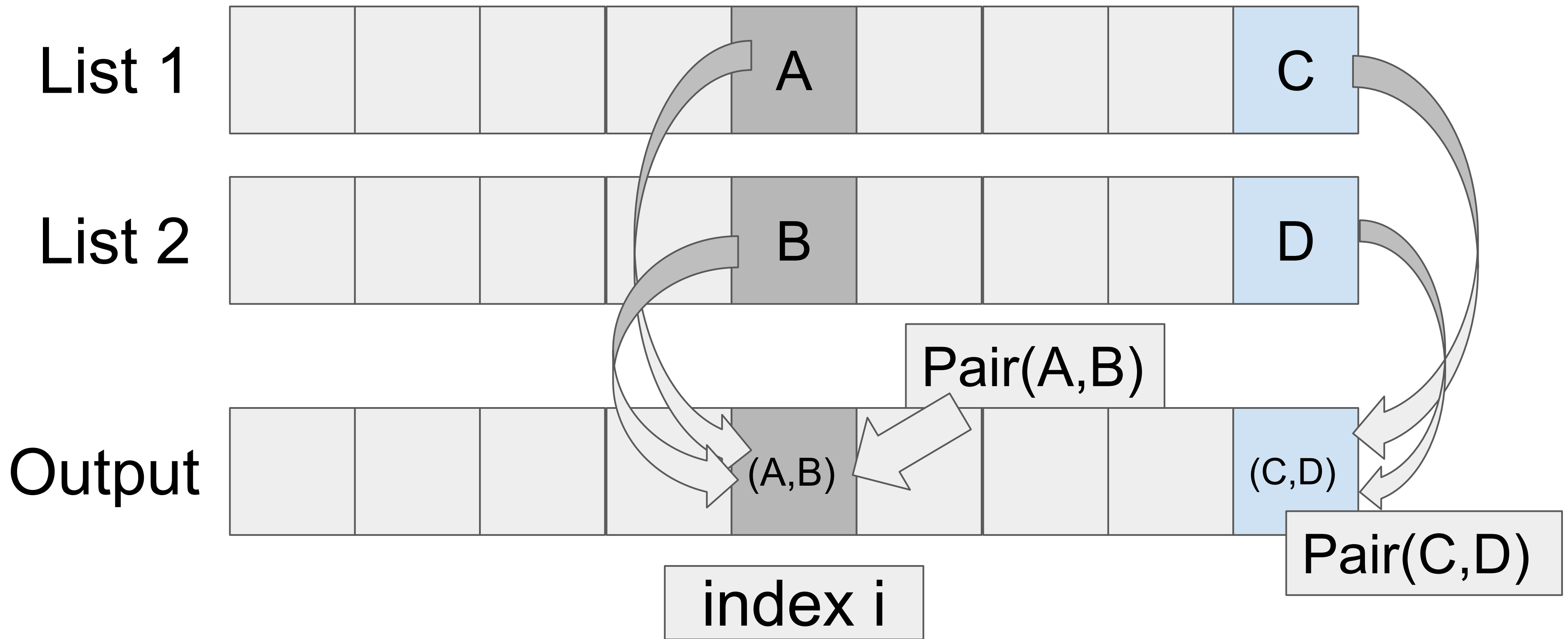
```
4 8
s Just 1
a 1
s The 1
a 1
s Way 1
a 1
s You 1
a 1
```

Output

```
Just
The
Way
You
```

Problem 3

Problem Description : Write **a method '*ziplist*'** which inputs two arbitrarily typed *Lists*, and outputs a single *ArrayList* which each element of the List is a *Pair* of two elements,from the same index element of both two input lists.



Problem 3

- Pair is defined as following.

```
class Pair<K,V>{
 K key; V value;
 Pair(K key, V value){ this.key = key; this.value = value; }
 public String toString() {   return "(" + key + ", " + value + ")";  }
}
```

Hint (i) the signature of the method should be as below.

```
static <T1,T2> ArrayList<Pair<T1,T2>> zipList(List<T1> a,List<T2> b )
```

(ii) List<E> interface spec

add(E *elem*) : Append *elem* at the end of the List<E>

get(int *index*) : Get the *index*-th element of the List<E>

Problem 3

Restriction :

- *zipList* method should be defined inside *Problem3* class.
- Codes of the *Pair* class should not be modified.
- We are going to evaluate it in a separate main class by *Problem3.zipList* call, so define the method as static, as given in the hint.
- Two input lists to be zipped are all going to be **same-length** lists, so no need to explicitly handle the different-length lists.
- Empty lists will not be an input.

Problem 3

Sample main() method

```
List<Integer> list1 = new ArrayList<Integer>();
List<Integer> list2 = new ArrayList<Integer>();
for(int i = 0; i < 10; i++){
 list1.add((int)(Math.random() * 10));
 list2.add((int)(Math.random() * 10));
}
System.out.println(list1);
System.out.println(list2);
System.out.println(zipList(list1,list2));
```

Example output

```
[1, 0, 9, 3, 0, 3, 2, 4, 7, 4]
[7, 1, 1, 3, 0, 2, 9, 2, 0, 9]
[(1, 7), (0, 1), (9, 1), (3, 3), (0, 0),
(3, 2), (2, 9), (4, 2), (7, 0), (4, 9)]
```

Problem 3

Sample main() method

```
List<String> list1 = new ArrayList<String>();
List<Integer> list2 = new ArrayList<Integer>();
list1.add("All"); list1.add("Computer");
list1.add("Programming"); list1.add("is not just");
list1.add("coding");
for(int i = 0; i < 5; i++){
 list2.add((int)(Math.random() * 5));
}
System.out.println(list1);
System.out.println(list2);
System.out.println(zipList(list1,list2));
```

output

```
[All, Computer,
Programming, is
not just, coding]
[1, 1, 1, 0, 3]
[(All, 1),
(Computer, 1),
(Programming, 1),
(is not just, 0),
(coding, 3)]
```