CPS 2231 2025 Spring

Lab 6: Advanced Object-Oriented Programming in Java

Topics: Inheritance, Polymorphism, Interfaces & Encapsulation

Instructor: Dr. Y. Tiffany Tang[1] and Dr. Pinata Winoto[2]

Teaching Assistants: Songlin Shang[3], Cong Ye[4] and Zike Deng[5]

Important Dates

Distribution Date:	April 15, 2025
Due Date:	April 30, 2025, 11:59 PM
Submission:	Submit through Canvas and GradeScope

[1]Tiffany Ya Tang, Associate Professor, Department of Computer Science, Wenzhou-Kean University; Email: yatang@kean.edu
[2]Pinata Winoto, Assistant Professor, Department of Computer Science, Wenzhou-Kean University; Email: pwinoto@kean.edu
[3]Songlin Shang, Department of Computer Science, Wenzhou-Kean University; Email: shangs@kean.edu
[4]Cong Ye, Department of Computer Science, Wenzhou-Kean University; Email: yecon@kean.edu
[5]Zike Deng, Department of Computer Science, School of Computing and Data Science, University of Hong Kong; Email: baylordeng@connect.hku.hk

1 Getting Started

1.1 File Setup

Please download the following files from Canvas:

- Lab6_1.java
- Lab6_2.java
- Lab6_3.java
- check.py (verification script)

Before you start the lab, create a project on Eclipse called Lab6. We recommend downloading the `check.py` file from Canvas and placing it in the same folder as your lab files. Your directory structure should look like:

Figure 1: Required directory structure for Lab6

1.2 Environment Setup

Before you start coding, navigate your terminal to the folder containing your lab files using the following command:

```
(base) matsumatsu@songprodeMacBook-Pro ~ % cd path  (path is the way
    to the folder of check.py)
```

Listing 1: Navigating to project directory

For example:

```
(base) matsumatsu@songprodeMacBook-Pro ~ % cd /Users/matsumatsu/
    Desktop/auto/Lab6/Lab6
(base) matsumatsu@songprodeMacBook-Pro Lab6 %
```

Listing 2: Example terminal navigation

Now you can proceed with your programming tasks after completing these preparation steps.

2 Programming Tasks

2.1 Lab6_1: Inheritance and Polymorphism

Write a Java program to create a base class `Animal` with methods `move()` and `makeSound()`. Create two subclasses `Bird` and `Panthera`. Override the `move()` method in each subclass to describe how each animal moves. Also, override the `makeSound()` method in each subclass to make a specific sound for each animal.

2.1.1 Requirements

1. Create a base class `Animal` with:
   - A method `move()` that prints: "Animal moves"
   - A method `makeSound()` that prints: "Animal makes a sound"
2. Create a subclass `Bird` that:
   - Inherits from `Animal`
   - Overrides `move()` to print: "Bird flies through the air"
   - Overrides `makeSound()` to print: "Bird chirps"
3. Create a subclass `Panthera` that:
   - Inherits from `Animal`
   - Overrides `move()` to print: "Panthera stalks through the grass"
   - Overrides `makeSound()` to print: "Panthera roars"

2.1.2 Test Cases

Your implementation will be tested with the following test cases:

2.1.3 Expected Output for Test Cases 1-3

Below are the expected outputs for the first three test cases:

```
Checking class definition...
Bird is a class: OK
Bird extends Animal: OK
All checks passed!
```

Listing 3: Test Case 1: Bird Inheritance - Expected Output

```
Checking method overriding...
Bird overrides move(): OK
Bird overrides makeSound(): OK
All checks passed!
```

Listing 4: Test Case 2: Bird Method Overriding - Expected Output

```
Testing Bird object output...
Bird flies through the air
Bird chirps
Output matches expected: OK
```

Listing 5: Test Case 3: Bird Output - Expected Output

Test Case	**Description**
1. Bird Inheritance	Verify that Bird class is defined and correctly inherits from Animal.
2. Bird Method Overriding	Check that Bird class overrides move() and makeSound() methods.
3. Bird Output	Test that calling move() and makeSound() on a Bird object produces the correct output.
4. Panthera Inheritance	Verify that Panthera class is defined and correctly inherits from Animal.
5. Panthera Method Overriding	Check that Panthera class overrides move() and makeSound() methods.
6. Panthera Output	Test that calling move() and makeSound() on a Panthera object produces the correct output.
7. Animal Output	Test that calling move() and makeSound() on an Animal object produces the correct output.
8. Polymorphism with Bird	Assign a Bird object to an Animal reference and test method calls.
9. Polymorphism with Panthera	Assign a Panthera object to an Animal reference and test method calls.
10. Array of Animals	Store Bird and Panthera objects in an Animal array and test method calls.

Table 1: Test cases for Lab6_1

2.2 Lab6_2: Sorting Algorithms Implementation

Write a Java program to create an interface `Sortable` with a method `sort()` that sorts an array of integers in ascending order. Create two classes `BubbleSort` and `SelectionSort` that implement the `Sortable` interface and provide their own implementations of the `sort()` method.

2.2.1 Sorting Algorithms Overview

Bubble Sort Bubble Sort is a simple comparison-based sorting algorithm. It works by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

Bubble Sort Algorithm:

1. Start from the first element, compare adjacent elements and swap them if they are in wrong order.
2. After the first pass, the largest element will be at the end of the array.
3. Repeat the process for the remaining elements, ignoring the last sorted element in each pass.
4. Continue until no more swaps are needed.

Example: Sorting the array [5, 3, 8, 4, 2]

Pass	Array State	Explanation
Initial	[5, 3, 8, 4, 2]	Unsorted array
Pass 1	[3, 5, 4, 2, 8]	Compare and swap: 5 & 3, 8 & 4, 4 & 2
Pass 2	[3, 4, 2, 5, 8]	Compare and swap: 5 & 4, 4 & 2
Pass 3	[3, 2, 4, 5, 8]	Compare and swap: 3 & 2
Pass 4	[2, 3, 4, 5, 8]	Final sorted array

Selection Sort Selection Sort is another comparison-based sorting algorithm. It divides the input list into two parts: a sorted sublist and an unsorted sublist. The algorithm repeatedly finds the minimum element from the unsorted sublist and moves it to the beginning of the unsorted sublist.

Selection Sort Algorithm:

1. Find the minimum element in the unsorted part of the array.
2. Swap it with the first element of the unsorted part.
3. Move the boundary between the sorted and unsorted parts one element to the right.
4. Repeat until the entire array is sorted.

Example: Sorting the array [5, 3, 8, 4, 2]

Pass	Array State	Min	Explanation
Initial	[5, 3, 8, 4, 2]	-	Unsorted array
Pass 1	[2, 3, 8, 4, 5]	2	2 swapped with 5
Pass 2	[2, 3, 8, 4, 5]	3	3 already in position
Pass 3	[2, 3, 4, 8, 5]	4	4 swapped with 8
Pass 4	[2, 3, 4, 5, 8]	5	5 swapped with 8
Final	[2, 3, 4, 5, 8]	-	Sorted array

2.2.2 Requirements

1. Create an interface `Sortable` with:
   - A method `sort(...)` that sorts an array of integers in ascending order
2. Create a class `BubbleSort` that:
   - Implements the `Sortable` interface
   - Implements the `sort()` method using the Bubble Sort algorithm
3. Create a class `SelectionSort` that:
   - Implements the `Sortable` interface
   - Implements the `sort()` method using the Selection Sort algorithm

2.2.3 Test Cases

Your implementation will be tested with the following test cases:

Test Case	Description
1. Interface Implementation	Verify that Sortable interface is correctly defined with the sort method.
2. BubbleSort Implementation	Test that BubbleSort class implements the Sortable interface correctly.
3. SelectionSort Implementation	Test that SelectionSort class implements the Sortable interface correctly.
4. Sorting Random Array	Test both sorting algorithms with a randomly generated array.
5. Sorting Already Sorted Array	Test performance with an already sorted array.
6. Sorting Reverse Sorted Array	Test with an array in descending order.
7. Sorting Array with Duplicates	Test with an array containing duplicate values.
8. Empty Array Handling	Test behavior when provided with an empty array.
9. Single Element Array	Test with an array containing only one element.
10. Large Array Performance	Test with a large array to evaluate performance.

Table 2: Test cases for Lab6_2

2.2.4 Expected Output for Test Cases 1-3

Below are the expected outputs for the first three test cases:

```
Compilation successful
Checking interface definition...
Sortable is an interface: OK
Checking method signatures...
sort method exists: OK
sort method accepts int[] parameter: OK
sort method has void return type: OK
All checks passed!
```

Listing 6: Test Case 1: Interface Implementation - Expected Output

```
Checking class definition...
BubbleSort is a class: OK
BubbleSort implements Sortable: OK
BubbleSort has sort method: OK

Testing Bubble Sort algorithm...
Original array: [5, 3, 8, 4, 2]
Pass 1: [3, 5, 4, 2, 8]
Pass 2: [3, 4, 2, 5, 8]
Pass 3: [3, 2, 4, 5, 8]
Pass 4: [2, 3, 4, 5, 8]
After BubbleSort: [2, 3, 4, 5, 8]
Sort validation: PASSED
```

Listing 7: Test Case 2: BubbleSort Implementation - Expected Output

```
Checking class definition...
SelectionSort is a class: OK
SelectionSort implements Sortable: OK
SelectionSort has sort method: OK

Testing Selection Sort algorithm...
Original array: [5, 3, 8, 4, 2]
Pass 1: [2, 3, 8, 4, 5]
Pass 2: [2, 3, 8, 4, 5]
Pass 3: [2, 3, 4, 8, 5]
Pass 4: [2, 3, 4, 5, 8]
After SelectionSort: [2, 3, 4, 5, 8]
Sort validation: PASSED
```

Listing 8: Test Case 3: SelectionSort Implementation - Expected Output

2.3 Lab6_3: Encapsulation with the Account Class

Write a Java program to create a class called `Account` with private instance variables `accountNumber`, `accountHolder`, and `balance`. Provide public getter and setter methods to access and modify these variables. Add a method called `deposit()` that takes an amount and increases the balance by that amount, and a method called `withdraw()` that takes an amount and decreases the balance by that amount.

2.3.1 Object-Oriented Programming Concepts

Encapsulation Encapsulation is one of the four fundamental OOP concepts and refers to the bundling of data and methods that operate on that data within a single unit (a class). It involves restricting direct access to some of an object's components, which is often done by making variables private and providing public methods to access and modify them.

Key Benefits of Encapsulation:

- **Data Hiding:** Private variables cannot be accessed directly from outside the class
- **Controlled Access:** Access to data is controlled through methods (getters and setters)
- **Data Validation:** Input validation can be performed within setter methods
- **Flexibility:** Implementation details can be changed without affecting code that uses the class
- **Maintainability:** Encapsulated code is easier to maintain and modify

2.3.2 Requirements

1. Create a class `Account` with:
   - Private instance variables:
     - `accountNumber` (...)
     - `accountHolder` (...)
     - `balance` (...)
   - Constructor(s) to initialize these variables
   - Public getter and setter methods:
     - `getAccountNumber()`, `setAccountNumber()`
     - `getAccountHolder()`, `setAccountHolder()`
     - `getBalance()`, `setBalance()`
   - Transaction methods:
     - `deposit(...)`: Increases the balance by the specified amount
     - `withdraw(...)`: Decreases the balance by the specified amount

2.3.3 Implementation Guidelines

- The `withdraw()` method should check if there is sufficient balance before allowing withdrawal
- Add appropriate validation in setter methods (e.g., balance cannot be negative)
- Format currency values appropriately when displaying account information
- Include a `toString()` method to display account details in a readable format

2.3.4 Test Cases

Your implementation will be tested with the following test cases:

Test Case	Description
1. Class Structure	Verify that Account class is properly defined with required private variables and public methods.
2. Constructor Implementation	Test that constructor(s) correctly initialize the account properties.
3. Getter/Setter Methods	Verify that all getter and setter methods work as expected.
4. Deposit Method	Test that the deposit method correctly increases the account balance.
5. Withdraw Method	Test that the withdraw method correctly decreases the account balance.
6. Insufficient Funds	Test that the withdraw method handles insufficient funds appropriately.
7. Multiple Transactions	Test a sequence of deposits and withdrawals on the same account.
8. Negative Values	Test validation of negative values for deposits and withdrawals.
9. Account Information	Test that account information is displayed correctly.
10. Multiple Accounts	Test operations on multiple account objects simultaneously.

Table 3: Test cases for Lab6_3

2.3.5 Expected Output for Test Cases 1-3

Below are the expected outputs for the first three test cases:

```
Checking class definition...
Account is a class: OK
Private fields check:
- accountNumber is private: OK
- accountHolder is private: OK
- balance is private: OK
Public methods check:
- getAccountNumber(): OK
- setAccountNumber(): OK
- getAccountHolder(): OK
- setAccountHolder(): OK
- getBalance(): OK
- setBalance(): OK
- deposit(): OK
- withdraw(): OK
All checks passed!
```

Listing 9: Test Case 1: Class Structure - Expected Output

```
Testing constructor...
Created account: Account #A1001, Holder: John Doe, Balance: $1000.00
```

```
Fields initialized correctly: OK
Created account with custom values: OK
All constructor tests passed!
```

Listing 10: Test Case 2: Constructor Implementation - Expected Output

```
Testing getter/setter methods...
Initial values:
Account #A1001, Holder: John Doe, Balance: $1000.00

Setting new values...
New account number: B2002
New account holder: Jane Smith
New balance: $2500.00

Getting values after update:
Account #B2002, Holder: Jane Smith, Balance: $2500.00
All getter/setter tests passed!
```

Listing 11: Test Case 3: Getter/Setter Methods - Expected Output

3 Implementation Instructions

3.1 General Guidelines

Before starting the implementation, please note the following:

- All code should be written in Java
- Each task has a template file provided in the Lab6_template folder
- Look for TODO comments in the template files - these indicate where you need to write your code
- Do not modify the existing test methods or main methods
- Ensure your code follows the exact output format specified in the test cases

3.2 Lab6_1: Implementation Details

In the Lab6_1.java template file, you need to implement three classes:

1. **Animal Class**
   - Look for the comment: "// TODO: Create the Animal class here"
   - Implement two methods:
     - `move()` method
     - `makeSound()` method
2. **Bird Class**
   - Look for the comment: "// TODO: Create the Bird class here"
   - Must extend the Animal class
   - Override both methods:
     - `move()` method
     - `makeSound()` method
3. **Panthera Class**
   - Look for the comment: "// TODO: Create the Panthera class here"
   - Must extend the Animal class
   - Override both methods:
     - `move()` method
     - `makeSound()` method

3.3 Lab6_2: Implementation Details

In the Lab6_2.java template file, you need to implement:

1. **Sortable Interface**
   - Look for the comment: "// TODO: Create the Sortable interface here"
   - Define the `sort()` method
2. **BubbleSort Class**
   - Look for the comment: "// TODO: Create the BubbleSort class here"
   - Must implement the Sortable interface
   - Implement the `sort()` method using bubble sort algorithm
   - Follow the algorithm steps described in the theory section
3. **SelectionSort Class**

- Look for the comment: "// TODO: Create the SelectionSort class here"
- Must implement the Sortable interface
- Implement the `sort()` method using selection sort algorithm
- Follow the algorithm steps described in the theory section

3.4 Lab6_3: Implementation Details

In the Lab6_3.java template file, you need to implement the Account class:

1. **Account Class**
   - Look for the comment: "// TODO: Create the Account class here"
   - Implement private fields:
     - `accountNumber` (String)
     - `accountHolder` (String)
     - `balance` (double)
   - Implement constructor for initializing all fields
   - Implement getter/setter methods:
     - `getAccountNumber()`, `setAccountNumber()`
     - `getAccountHolder()`, `setAccountHolder()`
     - `getBalance()`, `setBalance()`
   - Implement transaction methods:
     - `deposit()` method
     - `withdraw()` method
   - Implement `toString()` method

3.5 Important Notes

- All output must exactly match the format shown in the test cases
- Do not modify any existing code in the template files
- Test your implementation using the provided check.py script
- Make sure to handle edge cases (e.g., insufficient funds in withdraw)
- Use proper encapsulation principles (private fields, public methods)
- Follow Java naming conventions and code style guidelines

4 Submission Instructions

Congratulations on completing all labs in Lab6! Follow these steps to finalize your submission:

4.1 Generating Submission Package

Run the following command to verify your work and generate a submission package:

```
python check.py --uid 123456
```
(replace 123456 with your student number)

Terminal execution example:

```
(base) matsumatsu@songprodeMacBook-Pro ~ % python check.py --uid
    123456
```

Listing 12: Running the verification script

4.2 Verification Process

The script will check your lab files and create a zip package. If everything is correct, you will see:

```
(base) matsumatsu@songprodeMacBook-Pro ~ % python check.py --uid 123456
Your Student ID is 123456. Is this correct? (y/n):
```

Type `y` and press `Enter`. You will then see the packaging progress:

```
(base) matsumatsu@songprodeMacBook-Pro Lab6 %
********** Zipping Files **********
Zipping Lab6_1.java... Done.
Zipping Lab6_2.java... Done.
Zipping Lab6_3.java... Done.

Successfully created submission package: Lab6_123456_0415_1507.zip
```

After successful packaging, you will see a new zip file in your directory:

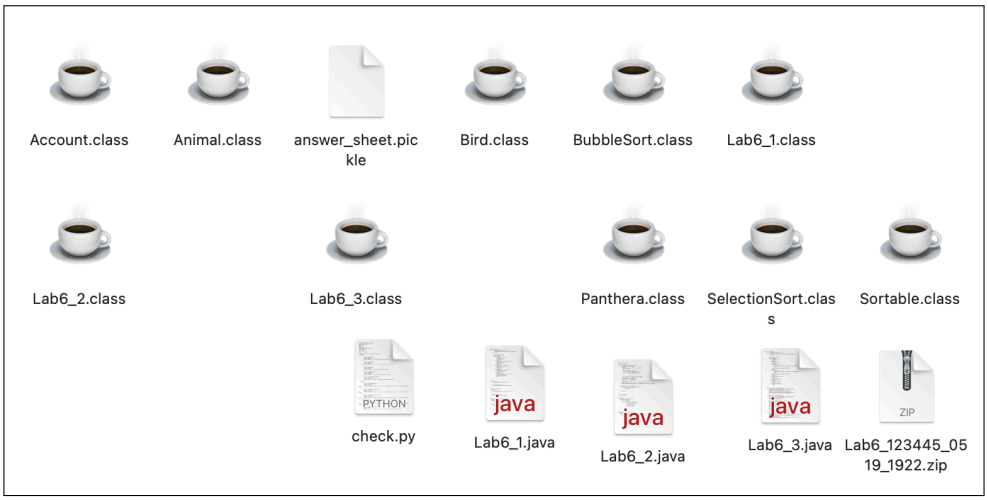

Figure 2: Directory with generated submission package

Submit this zip file to Canvas to complete your lab assignment.

5 Grading Criteria

Your submission will be graded based on the following criteria:

5.1 Test Cases

Each problem in Lab6 contains 10 test cases, and each test case carries equal points. Your total score for each problem will be calculated as:

$$\text{Problem Score} = \text{Number of Passed Test Cases} \times \text{Points per Test Case} \times x \tag{1}$$

Where x is a coefficient determined by your code implementation:

- $x = 1.0$ if you only use basic packages provided in the standard library
- $x = 0.6$ if you use any non-basic packages or external libraries

Important Note: For each task, only 5 out of the 10 test cases are visible in this document. The remaining 5 test cases are hidden and will be used for final grading. These hidden test cases are designed to thoroughly test your implementation's correctness and robustness. Therefore, it is crucial to:

- Carefully read and understand all requirements
- Test your code with various inputs beyond the visible test cases
- Handle edge cases and potential errors appropriately
- Follow all implementation guidelines strictly

5.2 Scoring Breakdown

Component	Details	Weight
Lab6_1.java	10 test cases	30%
Lab6_2.java	10 test cases	30%
Lab6_3.java	10 test cases	30%
Code style and documentation	Proper naming, comments, indentation	10%

Table 4: Grading distribution for Lab6

5.3 Penalties

The following penalties may be applied to your submission:

- Using non-basic packages: 40% reduction in score (coefficient $x = 0.6$)
- Late submission: 10% deduction per day
- High AI-generated code probability: Up to 100% deduction based on review
- Plagiarism: Zero score and possible disciplinary action

Note on Basic Packages: Basic packages refer to standard Java libraries that are part of the core Java Development Kit (JDK). Any external libraries or packages not included in the standard JDK will be considered non-basic packages.