BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

CS F213 – Object Oriented Programming

Lab 2: Reading user input, Designing Simple Classes

Time:	2 hours
Objective:	Teach students how to handle user input in Java applications and design basic classes with proper encapsulation, constructors, and methods through practical examples involving business scenarios.
Concepts Covered:	Java input handling (BufferedReader and Scanner classes), class design fundamentals (attributes, methods, constructors, visibility), object-oriented concepts (encapsulation, instance fields), and basic business logic implementation.
Prerequisites:	Basic Java syntax knowledge, understanding of object-oriented programming concepts, familiarity with data types and control structures, and basic understanding of class and object relationships.

1. Reading user inputs

There are two ways how you can read user inputs from the keyboard (referred by `stdin` in C and `System.in` in Java).First method is by using the instance of `BufferedReader` class and the second method is by using the instance of `Scanner` class.

1.1. Using BufferedReader for text-based user input from the keyboard

Copy and paste the following code in a file `Example1.java` and run the code.

```
import java.io.*; // java.io package is imported for using BufferedReaderclass

class Example1 {
	public static void main(String args[]) throws IOException {
		/*
		 * instantiate InputStreamReader class and pass System.in to its constructor
		 */
		InputStreamReader isr = new InputStreamReader(System.in);
		/*
		 * instantiate BufferedReader class and pass the reference variable isr which is
		 * of type InputStreamReader created in the previous line to the constructor of
		 * BufferedReader
		 */
		BufferedReader br = new BufferedReader(isr);
		System.out.println("Enter Your First Name: ");
		/*
		 * call readLine method on br reference variable which is of type BufferedReader
		 */
		String name = br.readLine();
```

```
            System.out.println("Your name is: " + name);
      } // End of main
}// End of class Example1
```

Let's understand some important pieces of code here:

`java.io.*`: Imports all classes from Java's input/output package, giving us tools to handle reading and writing data in our program.

`throws IOException`: Tells Java this code might have input/output errors (like if reading keyboard input fails) and needs error handling.

`InputStreamReader`: Converts raw input data (bytes) from keyboard into readable characters that Java can understand.

`BufferedReader`: A more efficient way to read text input that stores data in a temporary memory space (buffer) before processing.

These components work together like a pipeline: keyboard input → InputStreamReader converts bytes to characters → BufferedReader stores them efficiently → your program can read them with methods like readLine().

Exercise 1: Write a program in java to take 10 integer numbers as user input using the `BufferedReader` and print the sum of these numbers [name the file as `Exercise1.java`].

Driver Code:

```
import java.io.*;

class Exercise1 {
      public static void main(String args[]) throws IOException {
            // Step 1: Create input stream reader and buffered reader objects
            // Step 2: Initialize sum variable

            // TODO: Write code to:
            // 1. Use a loop to read 10 numbers
            // 2. For each iteration:
            // - Print a message asking for the number
            // - Read the string using br.readLine()
            // - Convert string to integer using Integer.parseInt()
            // - Add the number to sum

            // WRITE YOUR CODE HERE

            // Step 3: Print the final sum
      }
}
```

[CAREFUL: when reading input with BufferedReader, the input is always read as `String`, therefore you are required to parse the input to its correct type. In this Exercise use `Integer.parseInt()` method]

1.2. Using the Scanner class for text-based user input from the keyboard

Copy and paste the following code in a file `Example2.java` and run the code.

```
import java.util.Scanner;

class Example2 {
	public static void main(String args[]) {
		int num1;
		double double1;
		String numStr1, numStr2;
		/*
		 * instantiate scanner class by passing System.in to its constructor
		 */
		Scanner in = new Scanner(System.in);
		System.out.println("Enter an integer: ");
		num1 = in.nextInt(); // reads an int from keyboard
		System.out.println("You entered: " + num1);
		System.out.println("Enter a double: ");
		double1 = in.nextDouble();// reads a double from keyboard
		System.out.println("You entered: " + double1);
		System.out.println("Enter your first name ");
		numStr1 = in.next(); // reads string from keyboard
		System.out.println("Your name is " + numStr1);
		System.out.println("Enter your surname: ");
		numStr2 = in.next();
		System.out.println("Your surname is " + numStr2);
		in.close(); // always remember to free the resources by closing scanner
	}
}
```

Exercise 2: Write the program description given in Exercise 1 in Java using the `Scanner` class [name the file as `Exercise2.java`]

Exercise 3: Write a program that allows users to input the names of 5 movies using a `BufferedReader` or `Scanner` and assign a rating (1-5) to each movie. Store the movie names and ratings in an array and display the movies along with their average rating at the end.

---

2. Class Designing

Some guidelines before we dive into examples:

Without trying to be comprehensive or tedious, I want to end this chapter with some hints that will make your classes more acceptable in well-mannered OOP circles.

2.1.1. Always keep data private.

This is first and foremost; doing anything else violates encapsulation. You may need to write an accessor or mutator method occasionally, but you are still better off keeping the instance fields private. Bitter experience shows that the data representation may change, but how this data is used will change much less frequently. When data are kept private, changes in their representation will not affect the users of the class, and bugs are easier to detect.

2.1.2. Always initialize data.

Java won't initialize local variables for you, but it will initialize instance fields of objects. Don't rely on the defaults, but initialize all variables explicitly, either by supplying a default or by setting defaults in all constructors.

2.1.3. Don't use too many basic types in a class.

The idea is to replace multiple related uses of basic types with other classes. This keeps your classes easier to understand and to change. For example, replace the following instance fields in a `Customer` class:

```
private String street;
private String city;
private String state;
private int zip;
```

with a new class called Address. This way, you can easily cope with changes to addresses, such as the need to deal with international addresses.

2.1.4. Not all fields need individual field accessors and mutators.

You may need to get and set an employee's salary. You certainly won't need to change the hiring date once the object is constructed. And, quite often, objects have instance fields that you don't want others to get or set, such as an array of state abbreviations in an Address class.

2.1.5. Break up classes that have too many responsibilities.

This hint is, of course, vague: "too many" is obviously in the eye of the beholder. However, if there is an obvious way to break one complicated class into two classes that are conceptually simpler, seize the opportunity.

(On the other hand, don't go overboard; ten classes, each with only one method, are usually an overkill.)

Here is an example of a bad design:

```
public class CardDeck { // bad design

      private int[] value;
      private int[] suit;

      public CardDeck() {}
```

```
      public void shuffle() {}
      public int getTopValue() {}
      public int getTopSuit() {}
      public void draw() {}
}
```

This class really implements two separate concepts: a deck of cards, with its shuffle and draw methods, and a card, with the methods to inspect its value and suit. It makes sense to introduce a Card class that represents an individual card. Now you have two classes, each with its own responsibilities:

```
public class CardDeck {
      private Card[] cards;

      public CardDeck() {}
      public void shuffle() {}
      public Card getTop() {}
      public void draw() {}
}

public class Card
{
      private int value;
      private int suit;

      public Card(int aValue, int aSuit) {}
      public int getValue() {}
      public int getSuit() {}
}
```

2.1.6. Make the names of your classes and methods reflect their responsibilities.

Just as variables should have meaningful names that reflect what they represent, so should classes. (The standard library certainly contains some dubious examples, such as the Date class that describes time.)

A good convention is that a class name should be a noun (Order), or a noun preceded by an adjective (`RushOrder`) or a gerund (an "-ing" word, as in Billing Address). As for methods, follow the standard convention that accessor methods begin with a lowercase get (`getSalary`) and mutator methods use a lowercase set (`setSalary`).

2.1.7. Prefer immutable classes.

The `LocalDate` class, and other classes from the `java.time` package, are immutable—no method can modify the state of an object. Instead of mutating objects, methods such as `plusDays` return new objects with the modified state. The problem with mutation is that it can happen concurrently when multiple threads try to update an object at the same time. The results are unpredictable. When classes are immutable, it is safe to share their objects among multiple threads.

Therefore, it is a good idea to make classes immutable when you can. This is particularly easy with classes that represent values, such as a string or a point in time. Computations can simply yield new values instead of updating existing ones.

Of course, not all classes should be immutable. It would be strange to have the `raiseSalary` method return a new `Employee` object when an employee gets a raise.

2.2. Exercises

2.2.1. Write Java implementation for a class named '`Item`' which encapsulates the details of items to be purchased by the customer of the XYZ shop. The class Item is described as follows:

Attributes description:

1. **`itemName: String`** [Name of the ordered Item of the Customer]
2. **`itemidNo: String`** [unique identification number of the ordered Item of the Customer]
3. **`itemQuantity: int`** [quantity of the ordered Item of the Customer]
4. **`itemPrice: double`** [price of the ordered Item of the Customer]

Methods description:

The class supplies the methods(s) as per the following specification:

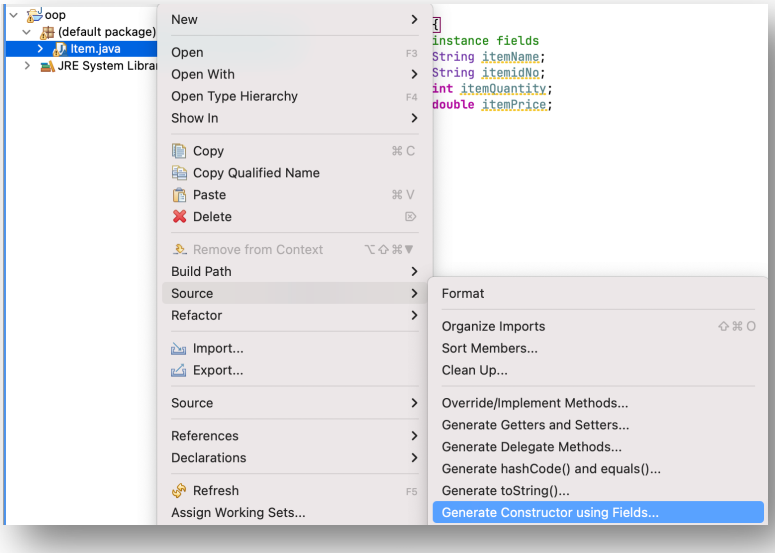
a) Any Item instance can be created either by supplying the value for all the instance fields in the order of **`itemName`**, **`itemidNo`**, **`itemQuantity`** and **`itemPrice`** OR by supplying the value for **`itemName`**, **`itemidNo`** and **`itemQuantity`** fields only OR by supplying the value for **`itemName`**, and **`itemidNo`** fields only. If an Item instance is created by providing the value for **`itemName`**, **`itemidNo`** and **`itemQuantity`** fields only then value for **`itemPrice`** is by default initialized to 500. If an Item instance is created by providing the value for **`itemName`** and **`itemidNo`** fields only then value for **`itemPrice`** is by default initialized to 500 and value for **`itemQuantity`** is by default initialized to 1.
b) Accessor and mutator methods are provided for every instance field.
c) All instance field(s) have a private visibility and all methods have a public visibility.

Driver Code:

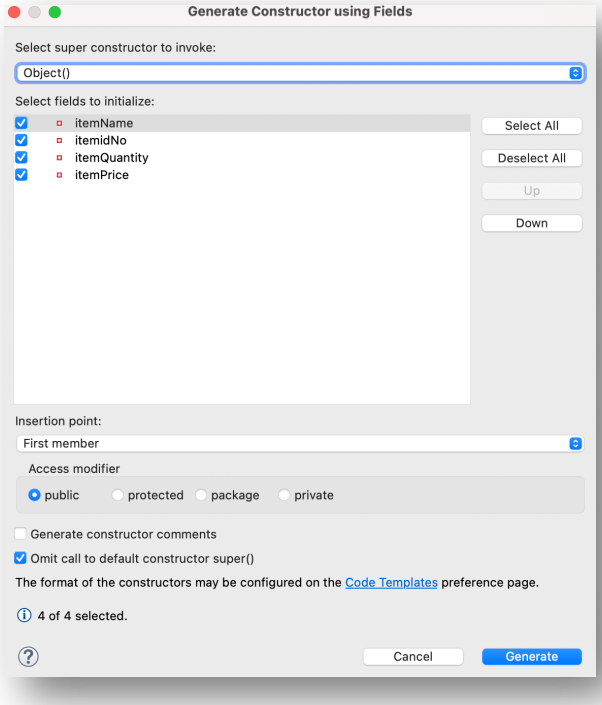
```
// Item class

class Item {
// Private instance fields
   private String itemName;
   private String itemidNo;
   private int itemQuantity;
   private double itemPrice;
}
```

Tips:

You can generate constructors, and getter and setter methods using Eclipse IDE. To do so, right click Item.java from Package Explorer pane → Source → Generate Constructor using Fields...



In the pop-up, select the fields you want to be in the constructor, and check the box next to 'Omit call to default constructor super()'.



Similarly, you can generate getter and setter methods. To do so, right click Item.java from Package Explorer pane → Source → Generate Getter and Setter Methods...

2.2.2. Write the java implementation for a class named '`TaxOnSalary`' to calculate tax on salary. The class `TaxOnSalary` is described as follows:

Attributes:

(i) `salary: double` // salary to calculate tax

(ii) `isPANsubmitted: boolean` // PAN submission status

Methods:

The class supplies the operation(s) as per the following specification:

(i) A `TaxOnSalary` instance can be created either by supplying the value for the instance field `isPANsubmitted` OR without supplying value for any field. If `TaxOnSalary` instance is created by providing the value for `isPANsubmitted` then the value for salary is initialized with 1000.00 however it can be reinitialized through the method `inputSalary()` *[which is described below]*. If `TaxOnSalary` instance is created without supplying value for any field, then value for *salary* and `isPANsubmitted` is by default initialized to `0.0` and `false` respectively

(ii) Accessor methods(s) are provided for every instance field.

(iii) A method for computing the tax based on salary [`calculateTax() : double`] is supplied. The tax is calculated as per the rules shown below:

a. if `salary` < 180000 and `isPANsubmitted` = true, then tax payable is zero
b. if `salary` < 180000 and `isPANsubmitted` = false, then tax payable is 5% of the salary
c. if 180000 < `salary` < 500000, then tax payable is 10% of the salary
d. if 500000 < `salary` < 1000000, then tax payable is 20% of the salary
e. if 1000000 < `salary`, then tax payable is 30% of the salary

A method named `inputSalary()` is supplied to read the value for the *salary* as an input from the user [consider reading this value from keyboard] and to assign the value to the corresponding instance variable *salary*.

Also, write a Test class named `TestTax.java` which

a) Creates two instances of `tax1` and `tax2` of the class `TaxOnSalary` with different initializations [see point (i) in the description of Methods].
b) Takes `salary` as an input from the user [using keyboard] for both the instances `tax1` and `tax2`.
c) Calculate and display tax for both the instance `tax1` and `tax2`.

```
// TaxOnSalary.java
// TODO: Add necessary imports

class TaxOnSalary {
      // TODO: Declare private instance variables for salary and isPANsubmitted

      // TODO: Implement constructor with no parameters
```

```
	// Initialize salary to 0.0 and isPANsubmitted to false

	// TODO: Implement constructor with isPANsubmitted parameter
	// Initialize salary to 1000.00

	// TODO: Implement accessor methods for salary and isPANsubmitted

	// TODO: Implement inputSalary() method
	// Use BufferedReader to read salary from keyboard

	// TODO: Implement calculateTax() method
	// Add tax calculation logic based on rules:
	// - salary < 180000 & PAN: 0%
	// - salary < 180000 & no PAN: 5%
	// - 180000-500000: 10%
	// - 500000-1000000: 20%
	// - >1000000: 30%
}
```

```
// TestTax.java
import java.io.*;

class TestTax {
	public static void main(String[] args) throws IOException {
		// TODO: Create two TaxOnSalary instances
		// - tax1 using no-argument constructor
		// - tax2 with isPANsubmitted value

		// TODO: Input salaries for both instances using inputSalary()

		// TODO: Calculate and display tax details for both instances
		// Output should show:
		// - Salary
		// - PAN submission status
		// - Calculated tax
	}
}
```

2.2.3. Write Java implementation for a class named BankAccount.java and complete the class as per specification below.

You are tasked with designing a program to simulate the basic operations of a bank account. The program should allow users to create accounts, deposit money, withdraw money, and view their account balances. This simulation will be implemented using the following steps:

Attributes

(i) `accountNumber`: String // Unique account number of the bank account.
(ii) `accountHolderName`: String // Name of the account holder.
(iii) `balance`: double // Balance amount in the bank account.

Methods

The `BankAccount` class supplies the operations as per the following specifications:

(i) A `BankAccount` instance can be created using either of the following ways:
1. By supplying the values for all instance fields (`accountNumber, accountHolderName`, and `balance`).
2. By supplying the values for only `accountNumber` and `accountHolderName` (in this case, the balance is initialized to 0.0 by default).

(ii) Accessor and mutator methods are provided for every instance field.

(iii) A method for depositing money (`deposit(double amount)`) is supplied:
- Adds the specified amount to the balance if the amount is positive.
- Prints a success message with the updated balance.
- If the amount is negative, prints an error message indicating an invalid deposit.

(iv) A method for withdrawing money (`withdraw(double amount)`) is supplied:
- Deducts the specified amount from the balance if sufficient funds are available.
- Prints a success message with the updated balance.
- If the amount exceeds the balance, prints an error message indicating insufficient funds.
- If the amount is negative, prints an error message indicating an invalid withdrawal.

(v) A method named `displayDetails()` is supplied to display the account number, account holder's name, and the current balance.

```
// Driver program
import java.util.Scanner;

class BankAccount {
	private String accountNumber;
	private String accountHolderName;
	private double balance;

	// Constructor to initialize all fields



	// Constructor to initialize accountNumber and accountHolderName only

	// Accessor methods
	public String getAccountNumber() {
		// TODO: Return accountNumber
	}
```

```
	public String getAccountHolderName() {
		// TODO: Return accountHolderName
	}

	public double getBalance() {
		// TODO: Return balance
	}

	// Deposit method
	public void deposit(double amount) {
		// TODO: Add the amount to balance if it is positive, otherwise display an error
	}

	// Withdraw method
	public void withdraw(double amount) {
		// TODO: Deduct the amount from balance if sufficient funds are available,
		// otherwise display an error
	}

	// Display account details
	public void displayDetails() {
		// TODO: Print account number, holder name, and balance
	}
}

public class TestBankAccount {
	public static void main(String[] args) {
		Scanner scanner = new Scanner(System.in);
		// TODO: Prompt the user to enter details for Account 1
		// Create an object of BankAccount for Account 1

		// TODO: Prompt the user to enter details for Account 2
		// Create an object of BankAccount for Account 2

		// Menu-driven interface

	}
}
```