- COMP2000: Software Engineering 2 - Introduction to Java (Part II)
  - Lecture Overview
  - Table of Contents
  - Encapsulation
  - Inheritance
  - Polymorphism
    - Method Overloading
    - Method Overriding
  - Abstraction
    - Interfaces
  - Lambda Expressions
  - I/O Operations
    - Scanner Class
    - BufferedReader Class
    - FileReader
    - Console Output
  - Debugging

COMP2000: Software Engineering 2 - Introduction to Java (Part II)

Lecture Overview

This lecture, delivered by Dr. Vivek Singh (Lecturer in Artificial Intelligence, University of Plymouth), covers advanced Java concepts including object-oriented programming principles, functional programming elements, I/O operations, and debugging techniques.

Table of Contents

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction
5. Lambda Expressions

6. I/O Operations
7. Debugging

Encapsulation

Encapsulation is the bundling of data (variables) and methods (functions) that work on the data into a single unit, typically a class. It's used to hide "sensitive" data from users.

Key points:

- Declare class variables/attributes as private
- Provide public get and set methods (getters and setters) to access and update private variables

Example:

```
public class EmployeeRecord {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String nm) {
        name = nm;
    }

    public static void main(String[] args) {
        EmployeeRecord emp = new EmployeeRecord();
        emp.setName("Helen");
        String e = emp.getName();
        System.out.println(e);
    }
}
```

Inheritance

Inheritance allows a class to inherit attributes and methods from another class.

Key concepts:

- Subclass (child): the class that inherits from another class

- Superclass (parent): the class being inherited from
- Use the `extends` keyword to inherit from a class

Example:

```
class Vehicle {
    protected String brand = "Ford";
    public void honk() {
        System.out.println("Tuut, tuut!");
    }
}

class Car extends Vehicle {
    private String modelName = "Mustang";
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.honk();
        System.out.println(myCar.brand + " " + myCar.modelName);
    }
}
```

Benefits of Inheritance:

- Code reusability
- Code organization
- Maintainability
- Easier upgrades

The `final` keyword:

- When applied to a class, it prevents the class from being extended
- Useful for creating immutable or secure classes

Polymorphism

Polymorphism allows us to perform a single action in different ways. It occurs when we have many classes related to each other by inheritance.

Example:

```
class Animal {
    public void animalSound() {
        System.out.println("The animal makes a sound");
```

```
    }
}

class Pig extends Animal {
    public void animalSound() {
        System.out.println("The pig says: wee wee");
    }
}

class Dog extends Animal {
    public void animalSound() {
        System.out.println("The dog says: bow wow");
    }
}
```

Method Overloading

- Multiple methods in a class with the same name but different parameters
- Increases readability and flexibility
- Can be achieved by changing the number of arguments or data type

Method Overriding

- Provides specific implementation of a method already provided by its superclass
- Used for runtime polymorphism
- Rules:
    1. Same method name as in the parent class
    2. Same parameters as in the parent class
    3. Must have an IS-A relationship (inheritance)

Abstraction

Abstraction is the process of hiding certain details and showing only essential information to the user.

Key points:

- Abstract classes cannot be used to create objects
- Abstract methods can only be used in an abstract class and do not have a body
- An abstract class can have both abstract and regular methods

Example:

```
abstract class Animal {
    public abstract void animalSound();
    public void sleep() {
        System.out.println("Zzz");
    }
}
```

Benefits of Abstraction:

- Simplifies code by providing a clean interface
- Encourages reuse
- Enforces a contract for subclasses

Interfaces

- Contains abstract methods and constants
- Used to achieve full abstraction
- All methods are public and abstract by default
- Can contain default and static methods with concrete implementations
- Supports multiple inheritance

Example:

```
interface Animal {
    public void animalSound();
    public void sleep();
}

class Pig implements Animal {
    public void animalSound() {
        System.out.println("The pig says: wee wee");
    }
    public void sleep() {
        System.out.println("Zzz");
    }
}
```

Lambda Expressions

Lambda expressions are a way to express anonymous functions concisely.

Syntax:

- `(parameters) -> expression`
- `(parameters) -> { statements }`

Examples:

```
// Filtering data
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
List<String> filteredNames = names.stream()
    .filter(name -> name.startsWith("A"))
    .collect(Collectors.toList());

// Map operations
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> squaredNumbers = numbers.stream()
    .map(n -> n * n)
    .collect(Collectors.toList());
```

I/O Operations

Scanner Class

Used for enabling user input.

Example:

```
import java.util.Scanner;

public class MyScannerClass {
    public static void main(String[] args) {
        Scanner myObj = new Scanner(System.in);
        System.out.println("Type a number:");
        int x = myObj.nextInt();
        System.out.println("You entered: " + x);
    }
}
```

BufferedReader Class

Used to read text from an input stream efficiently.

Example:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class BufferedReaderExample {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        System.out.println("Enter a line of text:");
        String line = br.readLine();
        System.out.println("You entered: " + line);
    }
}
```

FileReader

Used to read character files.

Example:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class ReadFile {
    public static void main(String[] args) throws IOException {
        FileReader fr = new FileReader("./src/readme");
        BufferedReader br = new BufferedReader(fr);
        String line;
        while ((line = br.readLine()) != null) {
            System.out.println(line);
        }
        br.close();
    }
}
```

Console Output

- `System.out.print()`: Prints without a newline
- `System.out.println()`: Prints with a newline

- `System.out.printf()`: Formats and prints text

Debugging

Debugging is the process of finding and correcting errors in a program.

Steps for effective debugging:

1. Prevent mistakes through good design and best practices
2. Find mistakes early through testing and tools
3. Reproduce the error
4. Generate hypotheses about the cause
5. Collect information (using print statements or debuggers)
6. Examine data and fix the error or generate new hypotheses

Techniques:

- Use pseudocode for high-level design
- Test important inputs and edge cases
- Use assertions to verify code behavior
- Create minimal test cases
- Use debugging tools like IntelliJ's debugger

Remember: "Don't introduce errors in the first place" is the best debugging strategy.