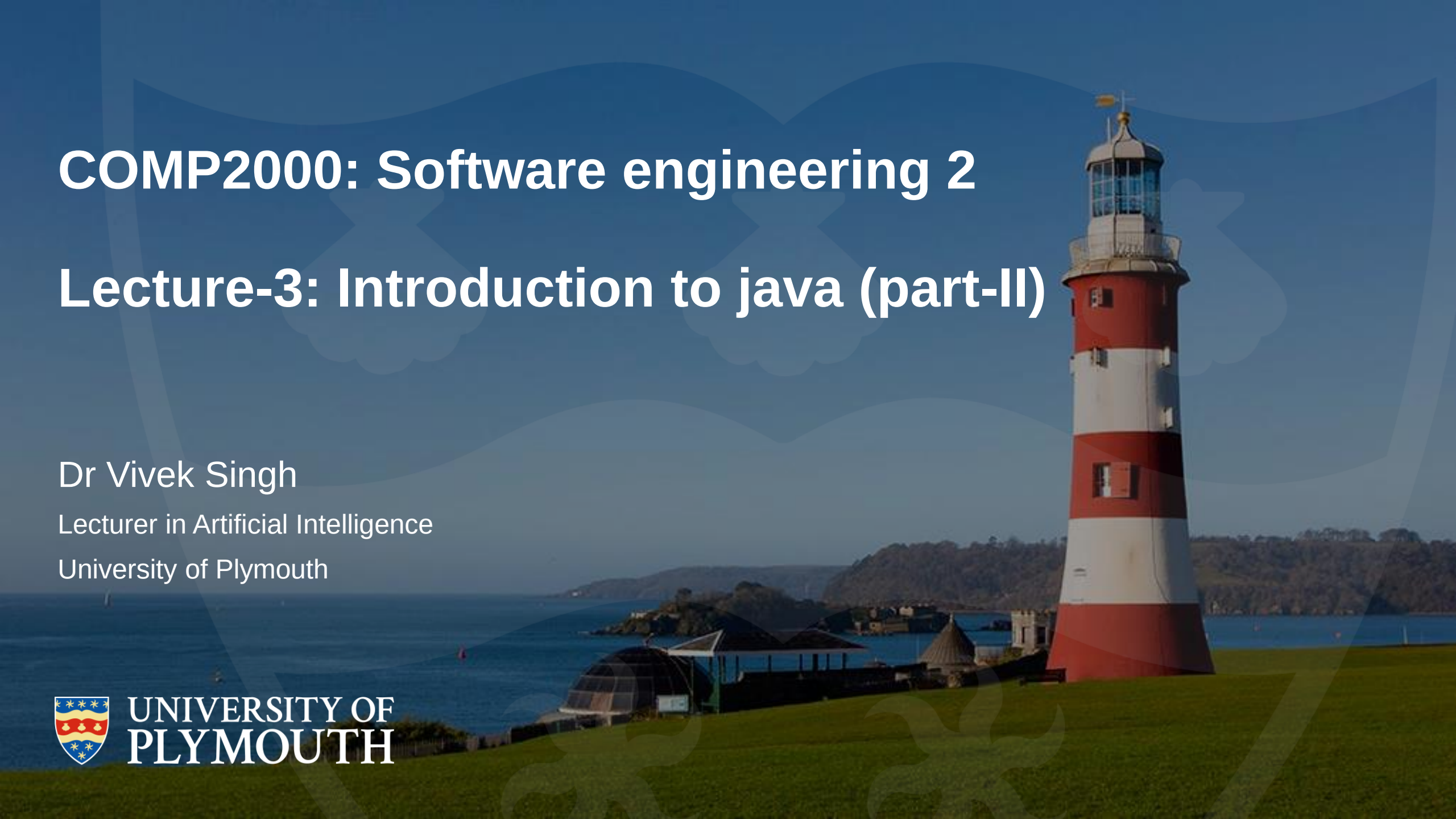
COMP2000: Software engineering 2

Lecture-3: Introduction to java (part-II)

Dr Vivek Singh

Lecturer in Artificial Intelligence

University of Plymouth



UNIVERSITY OF PLYMOUTH

Outline

- Encapsulation
- Inheritance
- Polymorphism
- Abstraction
- Lambda expression
- I/O operations
- Debugging

UNIVERSITY OF PLYMOUTH

Encapsulation

- The **Encapsulation**, is bundling the data (variables) and methods (functions) that work on the data into a single unit, typically a class.
- It also used to make sure that "sensitive" data is hidden from users.

To achieve this, you must:

- declare class variables/attributes as private
- provide public **get** and **set** methods (called getters and setters) to access and update the value of a private variable

Example

- The Getter method:

```
public String getName(){
    return name;
}
```

- The setter method:

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

```
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);

    }
}
```

```java
public class Employee {
    // Step 1: Private fields
    private String name;
    private int age;
    private double salary;

    // Step 2: Public getter and setter for 'name'
    public String getName() {
        return name;
    }

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

    // Public getter and setter for 'age'
    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        if (age > 0) {  // Validating age before setting
            this.age = age;
        } else {
            System.out.println("Age must be positive.");
        }
    }

    // Public getter and setter for 'salary'
    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        if (salary > 0) {  // Validating salary before setting
            this.salary = salary;
        } else {
            System.out.println("Salary must be positive.");
        }
    }
```

UNIVERSITY OF PLYMOUTH

Inheritance

Java Inheritance (Subclass and Superclass):
In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

subclass (child) - the class that inherits from another class
superclass (parent) - the class being inherited from

To inherit from a class, use the extends keyword.
In the example below, the Car class (subclass) inherits the attributes and methods from the Vehicle class (superclass):

```java
// Superclass
public class Vehicle {
    String brand = "Toyota";

    public void startEngine() {
        System.out.println("Engine started.");
    }
}

// Subclass
public class Car extends Vehicle {
    int numberOfDoors = 4;

    public void displayCarInfo() {
        System.out.println("Brand: " + brand);  // inherited from Vehicle
        System.out.println("Number of Doors: " + numberOfDoors);
    }
}
```

```java
// Main class to test inheritance
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.startEngine();  // method from Vehicle class
        myCar.displayCarInfo();  // method from Car class
    }
}
```

Example of Inheritance

```java
// Sedan class extending Car
public class Sedan extends Car {
    String sedanModel;

    public Sedan(String brand, int numberOfDoors, String sedanModel) {
        super(brand, numberOfDoors);  // Call to the Car constructor
        this.sedanModel = sedanModel;
    }

    public void displaySedanDetails() {
        System.out.println("Brand: " + brand);  // inherited from Vehicle
        System.out.println("Sedan Model: " + sedanModel);
        System.out.println("Number of Doors: " + numberOfDoors);  // inherited from Car
    }
}
```

Extending further

UNIVERSITY OF PLYMOUTH

```java
public class Main {
    public static void main(String[] args) {
        // Creating a Sedan object
        Sedan mySedan = new Sedan("Toyota", 4, "Camry");

        // Using methods
        mySedan.startEngine();          // from Vehicle
        mySedan.displayCarInfo();       // from Car
        mySedan.displaySedanDetails();  // from Sedan
```

The **brand** attribute in **Vehicle** was set to a default access modifier. If it was set to private, the Car class would not be able to access it.

Why And When To Use "Inheritance"?

- Code reusability: reuse attributes and methods of an existing class in new class.
- Code Organization: organize your code into more logical hierarchies.
- Maintainability: to fox bugs, you only have to make changes in one place, making the code easier to maintain
- Easier Upgrades: You can add new functionality in subclasses without changing the existing classes

The final Keyword

```
final class Vehicle {
...

 }
 class Car extends Vehicle {
...
}
```

- If class is declared as final, it cannot be extended by any other class
- useful when you want to create immutable or secure classes

Polymorphism

Polymorphism means "**many forms**", and it occurs when we have many classes that are related to each other by inheritance.

Polymorphism uses those methods to perform different tasks.

This allows us to perform a single action in different ways.

For example, think of a superclass called Animal that has a method called animalSound().

Subclasses of Animals could be Pigs, Cats, Dogs, Birds.

And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.).

Example

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

// Subclass 1: Pig
class Pig extends Animal {
    public void animalSound() {
        System.out.println("The pig says: oink oink");
    }
}

// Subclass 2: Cat
class Cat extends Animal {
    public void animalSound() {
        System.out.println("The cat says: meow meow");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();  // Create an Animal object
        Animal myPig = new Pig();  // Create a Pig object
        Animal myCat = new Cat();  // Create a Cat object

        myAnimal.animalSound();  // Output: The animal makes a sound
        myPig.animalSound();     // Output: The pig says: oink oink
        myCat.animalSound();     // Output: The cat says: meow meow
    }
}
```

UNIVERSITY OF PLYMOUTH

Methods Overloading

- allows multiple methods in a class to have the same name but different parameters
- increases the readability and flexibility

- There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

```java
public class Calculator {

    // Method 1: Sum with two int parameters
    public int sum(int a, int b) {
        return a + b;
    }

    // Method 2: Sum with three int parameters
    public int sum(int a, int b, int c) {
        return a + b + c;
    }

    // Method 3: Sum with two double parameters (different data type)
    public double sum(double a, double b) {
        return a + b;
    }
```

```java
    public static void main(String[] args) {
        Calculator calc = new Calculator();

        // Calling the different overloaded methods
        System.out.println(calc.sum(5, 10));          // Calls method 1
        System.out.println(calc.sum(5, 10, 15));      // Calls method 2
        System.out.println(calc.sum(5.5, 7.3));       // Calls method 3
    }
```

UNIVERSITY OF PLYMOUTH

Method overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules:

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

```java
// Superclass: Vehicle
class Vehicle {
    // This method will be overridden in subclasses
    public void fuelType() {
        System.out.println("The vehicle uses generic fuel.");
    }

    // Shared method across all vehicles
    public void startEngine() {
        System.out.println("Engine is starting...");
    }
}

// Subclass 1: Car
class Car extends Vehicle {
    @Override
    public void fuelType() {
        System.out.println("The car uses gasoline.");
    }
}
```

```java
// Subclass 2: ElectricCar
class ElectricCar extends Vehicle {
    @Override
    public void fuelType() {
        System.out.println("The electric car uses electricity.");
    }
}

// Subclass 3: Motorcycle
class Motorcycle extends Vehicle {
    @Override
    public void fuelType() {
        System.out.println("The motorcycle uses petrol.");
    }
}
```

```java
// Main class to test method overriding
public class Main {
    public static void main(String[] args) {
        // Creating instances of the subclasses
        Vehicle myCar = new Car();
        Vehicle myElectricCar = new ElectricCar();
        Vehicle myMotorcycle = new Motorcycle();

        // All vehicles start the engine in the same way
        myCar.startEngine();            // Output: Engine is starting...
        myElectricCar.startEngine();    // Output: Engine is starting...
        myMotorcycle.startEngine();     // Output: Engine is starting...

        // Method overriding: different behavior for each subclass
        myCar.fuelType();               // Output: The car uses gasoline.
        myElectricCar.fuelType();       // Output: The electric car uses electricity.
        myMotorcycle.fuelType();        // Output: The motorcycle uses petrol.
    }
}
```

Abstraction

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.

Abstraction can be achieved with either **abstract classes** or **interfaces** (which you will learn more about in the next slides).
The abstract keyword is a non-access modifier, used for classes and methods:

Abstract class: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
Abstract method: can only be used in an abstract class, and it does not have a body.
The body is provided by the subclass (inherited from).

An abstract class can have both abstract and regular methods:

UNIVERSITY OF PLYMOUTH

- A class which contains the **abstract** keyword in its declaration is known as abstract class.
- Abstract classes may or may not contain *abstract methods*, i.e., methods without body (**public void get();**)
- But, if a class has at least one abstract method, then the class **must** be declared abstract.
- If a class is declared abstract, it cannot be instantiated.
- To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.
- If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

Example

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

Try to generate an object.

```
Animal myObj = new Animal(); // will generate an error
```

UNIVERSITY OF PLYMOUTH

```java
// Abstract Class
abstract class Vehicle {
    // Abstract method (doesn't have a body)
    public abstract void startEngine();

    // Regular method
    public void stopEngine() {
        System.out.println("Engine stopped.");
    }
}

// Subclass that inherits Vehicle
class Car extends Vehicle {
    // Implementing the abstract method
    public void startEngine() {
        System.out.println("Car engine started.");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        // Cannot instantiate an abstract class
        // Vehicle myVehicle = new Vehicle(); // Not allowed

        // Create a Car object (subclass of Vehicle)
        Vehicle myCar = new Car();
        myCar.startEngine();  // Output: Car engine started.
        myCar.stopEngine();   // Output: Engine stopped.
    }
}
```

- Abstract class example

Why use Abstraction?

- Simplifies code by providing a clean interface and hiding unnecessary complexity
- Encourages reuse by allowing subclasses to provide specific implementations for abstract methods.
- Enforces a contract that subclasses must adhere to, ensuring that important methods are implemented.

Interface

- blueprint of a class that contains abstract methods and constants
- Interfaces are used to achieve full abstraction in Java
- Unlike abstract classes, interfaces cannot contain any concrete methods (except in special cases like default and static methods)
- The class that implements an interface must provide concrete implementations for all of the interface's methods

- Java does not support multiple inheritance but it does support multiple inheritance with interfaces.
- By default, all methods in an interface are public and abstract.
- Interface can contain default and static methods with concrete implementations.

Example of Interface

```java
interface Animal {
    void makeSound();  // Abstract method, must be implemented by subclasses
}

// Implement the interface in a class
class Dog implements Animal {
    // Implementing the method from the Animal interface
    @Override
    public void makeSound() {
        System.out.println("Dog barks: Woof Woof");
    }
}

// Implement the interface in another class
class Cat implements Animal {
    // Implementing the method from the Animal interface
    @Override
    public void makeSound() {
        System.out.println("Cat meows: Meow Meow");
    }
}
```

Multiple inheritance using interfaces

```
// Interface 1: Vehicle
interface Vehicle {
    // Abstract method representing starting the vehicle
    void startEngine();

    // Abstract method representing stopping the vehicle
    void stopEngine();
}

// Interface 2: Electric
interface Electric {
    // Abstract method for charging the vehicle
    void chargeBattery();

    // Abstract method for showing battery status
    void showBatteryStatus();
}
```

```java
// Class: ElectricCar implementing both Vehicle and Electric interfaces
class ElectricCar implements Vehicle, Electric {
    private int batteryLevel = 100;

    // Implementing startEngine() from Vehicle interface
    @Override
    public void startEngine() {
        if (batteryLevel > 0) {
            System.out.println("Electric car engine started.");
        } else {
            System.out.println("Battery is empty! Cannot start engine.");
        }
    }
```

```java
    // Implementing chargeBattery() from Electric interface
    @Override
    public void chargeBattery() {
        batteryLevel = 100;
        System.out.println("Battery is fully charged.");
    }

    // Implementing showBatteryStatus() from Electric interface
    @Override
    public void showBatteryStatus() {
        System.out.println("Battery level is: " + batteryLevel + "%");
    }
```

UNIVERSITY OF PLYMOUTH

Lambda expression

- It is a way to express an anonymous function (a function without a name) in a more concise and readable way.
- Used when you need to pass small pieces of functionality as parameters.

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

Filtering Data with lambda expression

```java
import java.util.Arrays;
import java.util.List;

public class StreamExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        // Using lambda to filter and print even numbers
        numbers.stream()
                .filter(n -> n % 2 == 0)
                .forEach(n -> System.out.println(n));
    }
}
```

Map Operations with lambda expression

```java
import java.util.HashMap;
import java.util.Map;

public class LambdaMap {
    public static void main(String[] args) {
        Map<String, Integer> itemPrices = new HashMap<>();
        itemPrices.put("Apple", 50);
        itemPrices.put("Banana", 20);
        itemPrices.put("Orange", 30);

        // Using lambda to print each item and its price
        itemPrices.forEach((item, price) -> System.out.println
            (item + ": " + price + " units"));
    }
}
```

- Custom interfaces with lambda expression

```java
// Define a custom functional interface
@FunctionalInterface
interface MathOperation {
    int operate(int a, int b);
}

public class Main {
    public static void main(String[] args) {
        // Lambda expression to add two numbers
        MathOperation addition = (a, b) -> a + b;

        // Lambda expression to subtract two numbers
        MathOperation subtraction = (a, b) -> a - b;

        // Using the lambda functions
        int sum = addition.operate(10, 5);  // 10 + 5 = 15
        int difference = subtraction.operate(10, 5);  // 10 - 5 = 5

        System.out.println("Sum: " + sum);
        System.out.println("Difference: " + difference);
    }
}
```

I/O operations (Scanner class)

- Scanner class is used to enable user's input.
- The Scanner class can be found in the java.util package.

- Example:
- `Scanner myObj = new Scanner(System.in); // Create a Scanner object`
- The following example is a java program to add two numbers.

Exampl
e

```java
import java.util.Scanner;

public class MyScannerClass {
    public static void main(String[] args) {
        int x, y, sum;
        Scanner myObj = new Scanner(System.in); // Create a Scanner object
        System.out.println("Type a number:");
        x = myObj.nextInt(); // Read user input

        System.out.println("Type another number:");
        y = myObj.nextInt(); // Read user input

        sum = x + y;  // Calculate the sum of x + y
        System.out.println("Sum is: " + sum); // Print the sum
    }
}
```

Scanner with multiple types of data

```java
import java.util.Scanner;

public class ScannerExample {
    public static void main(String[] args) {
        // Create a Scanner object
        Scanner scanner = new Scanner(System.in);

        // Reading an integer
        System.out.println("Enter an integer:");
        int intValue = scanner.nextInt();

        // Reading a floating-point number
        System.out.println("Enter a floating-point number:");
        float floatValue = scanner.nextFloat();

        // Consume the newline character left by nextFloat()
        scanner.nextLine(); // This is important after reading numerical input

        // Reading a string
        System.out.println("Enter a string:");
        String stringValue = scanner.nextLine();
```

BufferedReader Class

- is used to read text from an input stream efficiently, character by character or by lines

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

public class Main {
    public static void main(String[] args) throws IOException {
        // Creating BufferedReader using InputStreamReader to read from console
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

        System.out.println("Enter your name:");
        String name = reader.readLine();  // Read a line of text

        System.out.println("Enter your age:");
        int age = Integer.parseInt(reader.readLine());  // Read and parse integer
```

UNIVERSITY OF PLYMOUTH

The Full Picture

10010101010101000101 …

InputStream
`System.in`

‘O’ ‘k’ ‘a’ ‘y’ ‘ ‘ ‘a’ ‘w’ ‘e’ …

InputStreamReader

“Okay awesome, cool\n” …

BufferedReader

InputStream

- InputStream is a stream of bytes
  - Read one byte after another using `read()`
- A byte is just a number
  - Data on your hard drive is stored in bytes
  - Bytes can be interpreted as characters, numbers..

```
InputStream stream = System.in;
```

InputStreamReader

- Reader is a class for character streams
  - Read one character after another using `read()`
- InputStreamReader takes an InputStream and converts bytes to characters
- Still inconvenient
  - Can only read a character at a time

```
new InputStreamReader(stream)
```

BufferedReader

- BufferedReader buffers a character stream so you can read line by line
  - `String readLine()`

```
new BufferedReader(
    new InputStreamReader(System.in));
```

Checked exceptions

- Java compiler checks if the code handles these exceptions properly
- You must handle checked exceptions either by catching them with a try-catch block or by declaring them using the throws keyword.
- Checked exceptions are subclasses of Exception.

Example

```java
public class Main {
    public static void main(String[] args) {
        try {
            BufferedReader reader = new BufferedReader(new FileReader("file.txt"));
            String line = reader.readLine();
        } catch (IOException e) {
            System.out.println("An I/O error occurred: " + e.getMessage());
        }
```

```java
public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(new FileReader("file.txt"));
        String line = reader.readLine();  // This can throw IOException
    }
}
```

UNIVERSITY OF PLYMOUTH

FileReader

- FileReader takes a text file
  - converts it into a character stream
  - FileReader(“PATH TO FILE”);

- Use this + BufferedReader to read files!

```
FileReader fr = new FileReader(“readme.txt”);
BufferedReader br = new BufferedReader(fr);
```

UNIVERSITY OF PLYMOUTH

FileReader Code

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

public class ReadFile {

  public static void main(String[] args)  throws IOException{
    // Path names are relative to project directory (Eclipse Quirk )
    FileReader fr = new FileReader("./src/readme");
    BufferedReader br = new BufferedReader(fr);
    String line = null;
    while ((line = br.readLine()) != null) {
      System.out.println(line);
    }
    br.close();

  }
}
```

UNIVERSITY OF PLYMOUTH

Console Output with System.out

- System.out.print(): Prints text to the console without a newline at the end.
- System.out.println(): Prints text to the console followed by a newline.
- System.out.printf(): Formats and prints text using a specified format.

```
public class OutputExample {
    public static void main(String[] args) {
        System.out.print("Hello, ");
        System.out.println("World!");  // Moves to the next line after printing
        System.out.printf("Hello, %s! You have %d new messages.%n", "Alice", 5);
    }
}
```

Buffered Output with BufferedWriter

```java
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class BufferedOutputExample {
    public static void main(String[] args) {
        try {
            BufferedWriter writer = new BufferedWriter(new
                FileWriter("buffered_output.txt"));
            writer.write("Writing with BufferedWriter.");
            writer.newLine();
            writer.write("This is the second line.");

            writer.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```java
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class ObjectOutputExample {
    public static void main(String[] args) {
        try {
            FileOutputStream fileOut = new FileOutputStream("object_output.ser");
            ObjectOutputStream objOut = new ObjectOutputStream(fileOut);

            Person person = new Person("John Doe", 30);
            objOut.writeObject(person);

            objOut.close();
        } catch (IOException e) {
            e.printStackTrace();
```

```java
class Person implements Serializable {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
```

UNIVERSITY OF PLYMOUTH

ObjectOutputStream: to write objects to an output stream, which can be saved to a file.

Debugging

The process of finding and correcting an error in a program

A fundamental skill in programming

You will be required to debug at least once in your project

UNIVERSITY OF PLYMOUTH

Step 1: Don’t Make Mistakes

Don’t introduce errors in the first place

- Reuse: find existing code that does what you want
- Design: think before you code
- Best Practices: Recommended procedures/ techniques to avoid common problems

Design: Pseudocode

A high-level, understandable description of what a program is supposed to do

Don’t worry about the details, worry about the structure

Pseudocode: Interval Testing

Example:

Is a number within the interval [*x*, *y*)?

If number =< x return false

else If number > y return false

else return true

Step 2: Find Mistakes Early

Easier to fix errors the earlier you find them

- Test your design
- Tools: detect potential errors
- Test your implementation
- Check your work: assertions

Testing: Important Inputs

Want to check all “paths” through the program.

Think about one example for each “path”

Example:

Is a number within the interval [*x*, *y*)?

Intervals: Important Cases

Below the lower bound

Equal to the lower bound

Within the interval

Equal to the upper bound

Above the upper bound

Intervals: Important Cases

What if lower bound > upper bound?

What if lower bound == upper bound?

(hard to get right!)

UNIVERSITY OF PLYMOUTH

Tools: IntelliJ Warnings

- Warnings: may not be a mistake, but it likely is.
- Suggestion: Always pay attention to warnings. Even if the program works as expected
  - developers should try to fix all warnings
- Extra checks: FindBugs and related tools
- Unit testing: Unit testing ensures that individual pieces of code work as expected

UNIVERSITY OF PLYMOUTH

Assertions

Verify that code does what you expect

If true: nothing happens
If false: program crashes with error

Disabled by default (enable with `-ea flag`)

```
assert difference >= 0;
```

```java
public class Main {
    public static void main(String[] args) {
        int difference = -5; // This will cause the assertion to fail

        // This checks if the condition holds, otherwise it throws an AssertionError
        assert difference >= 0 : "Difference cannot be negative";

        System.out.println("Difference is: " + difference);
    }
}
```

UNIVERSITY OF PLYMOUTH

Step 3: Reproduce the Error

- Figure out how to repeat the error
- Create a minimal test case

Go back to a working version, and introduce changes one at a time until the error comes back

Eliminate extra stuff that isn’t used

Step 4: Generate Hypothesis

What is going wrong?

What might be causing the error?

Question your assumptions: “x can’t be possible:” What if it is, due to something else?

Step 5: Collect Information

If x is the problem, how can you verify?
Need information about what is going on inside the program

System.out.println() is very powerful

Eclipse debugger can help

Step 6: Examine Data

Examine your data

Is your hypothesis correct?

Fix the error, or generate a new hypothesis

Would you like to share your thoughts and suggestions on COMP2000?

- Menti.com

Thank you

UNIVERSITY OF PLYMOUTH