SLIIT UNI
THE KNOWLEDGE UNIVERSITY

Sri Lanka Institute of Information Technology

Faculty of Computing

SE1020 - Object-Oriented Programming

Year 01 and Semester 02

Lecture 05

Advanced OOP Concepts

Recall. . .

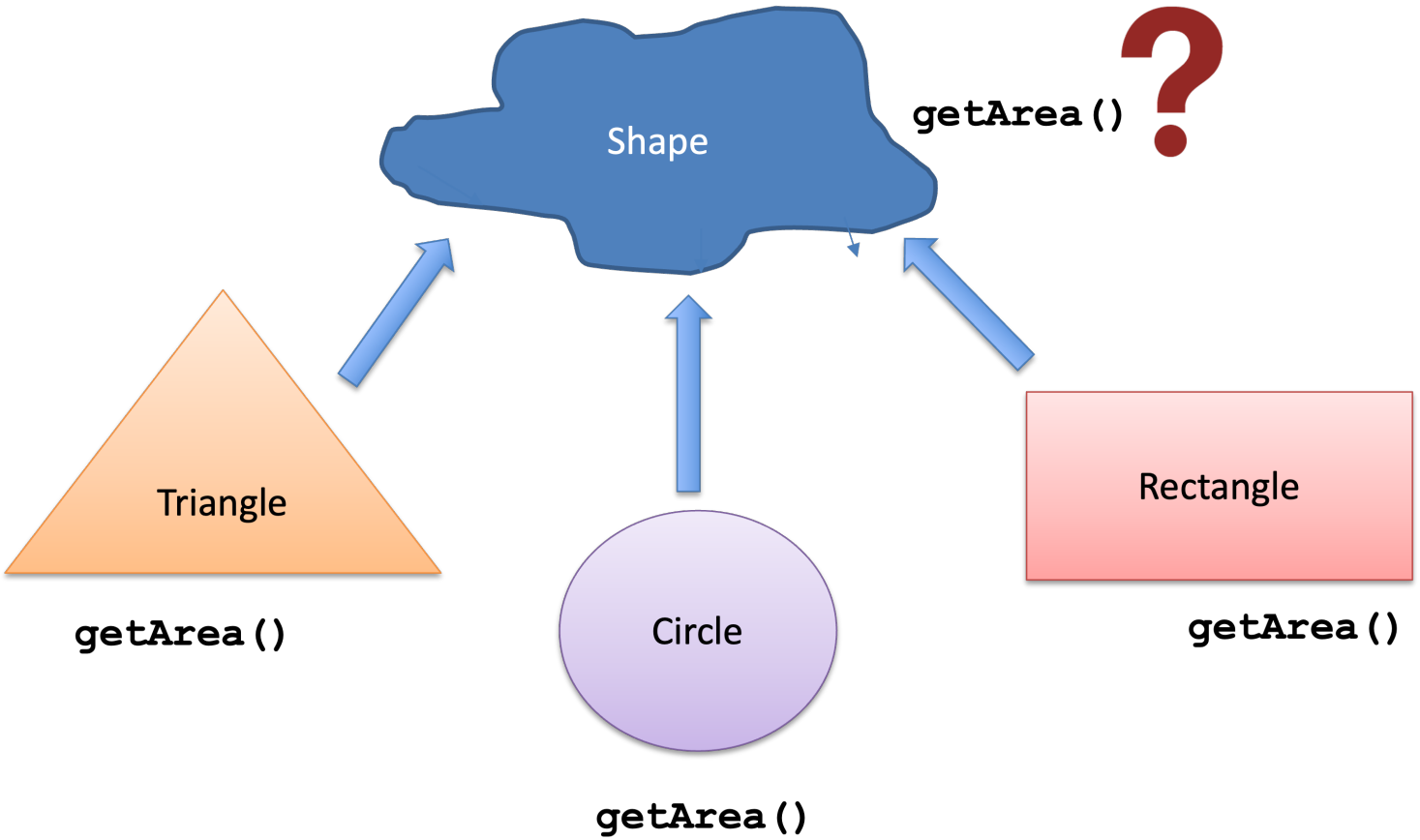
- Class relationships OOP include inheritance, composition, aggregation, association, and dependency, each describing how classes interact with one another.
- Inheritance enables a subclass to reuse and extend the behavior of a superclass.
- Composition represents a strong “whole-part” relationship where the part cannot exist independently, whereas aggregation shows a weaker link where parts can exist separately
- Association is a general ”has-a” or ”uses-a” relationship between two independent classes and can be one-way or two-way.
- Dependency is the weakest relationship type, where one class temporarily uses another within a method without storing it as an attribute.

Learning Objectives

- Understand and apply abstract classes to model common but incomplete behaviors.
- Use interfaces for designing flexible and extensible code.
- Implement and explain the purpose of the static modifier in Java programs.

Abstract Classes

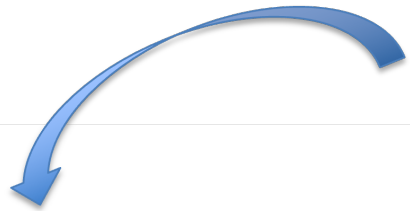
- Used in situations in which you will want to define a superclass where some methods don't have a complete implementation.
- Here we are expecting the sub classes will implement these abstract methods.
- One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method.
- e.g an area() method of a Shape class.


Shape
getArea()
Triangle
getArea()
Circle
getArea()
Rectangle
getArea()

Refer the following class called Shape.

```
class Shape{
    private String color;

    public double getArea(){
        //How to implement this method meaningfully?
        return 0;
    }
}
```

Any Shape has an area. So that class shape can contain a method called ***getArea().*** But how to calculate the area of a shape?



```
class Circle extends Shape{
    private double radius;

    public double getArea(){
        return 22 / 7 * this.radius * this.radius;
    }
}
```

If we are asked to calculate the area of a circle, then we know how to calculate the area of a circle.

- **getArea()** method will have different implementations depending on the child class.
- Any shape has an area. So that class Shape can contain a method called **getArea()**.
- But implementing **getArea()** method is possible only when we know the child class.
- The implementation of **getArea()** method is different from one child class to another.

```
class Rectangle extends Shape{
    private double length, width;

    public double getArea(){
        return this.length * this.width;
    }
}
```

- The methods which cannot be implemented **MUST** be defined as abstract.

```
abstract class Shape{
    private String color;

    public abstract double getArea();
}
```

- An abstract method does NOT have a method implementation (not even { }).
- We cannot invoke (call) an abstract method.
- The class which contain **at least one abstract method** MUST be defined as abstract.

- Every child class **MUST** override **all the abstract methods** of the parent class.
- If a child class did not override all the abstract method of the parent, then the child class also become abstract.
- An abstract class can contain;
  - Abstract methods (no implementation)
  - Concrete methods (with implementation)
- An abstract class is a class that **CANNOT** be instantiated.

- An abstract class can behave as a data type.

```
Shape s1;
```

Correct

- But cannot create instances (object) from an abstract class.

```
s1 = new Shape();
```

Wrong

- But super class variables can refer to child class objects.

```
s1 = new Circle();
Shape s2 = new Rectangle()
```

Correct

- Can call the getArea() method belongs to the child.

```
s1.getArea();
s2.getArea();
```

Correct
But the two method calls will perform two different actions

Abstract Classes

- Abstract classes are typically generic and represent incomplete concepts.
- In most object-oriented designs, the top of a class hierarchy is an abstract class.
- Classes that can be instantiated and provide full method implementations are called **concrete classes**.
- A class that contains at least one abstract method must be declared as abstract.
- Objects cannot be created from an abstract class.
- All child classes must override all the abstract methods of their parent abstract class, unless they are also declared abstract.
- Abstract classes forcing subclasses to provide specific method implementations.

Exercise 01:

Assume that you need to create a class **Animal** that has a two methods **eat()** and **makeSound()** and the subclasses **Dog** and **Cat**. Implement the three classes and necessary methods using the concept of the abstract classes. The implemented classes should be capable of creating the given main program and generating the given output.

Program

```
Animal dog = new Dog();
dog.eat();
dog.makeSound();

Animal cat = new Cat();
cat.eat();
cat.makeSound();
```

Output

```
I am eating
Woof woof
I am eating
Meaw meaw
```

Interfaces

- An interface is a contract which defines **what a class must do, not how**.
- All methods in an interface are public and abstract by default (interface is a fully abstract class).
- Interface attributes are by default public, static and final.
- A class can implement multiple interfaces.
- Although Java does not support multiple inheritance with classes, it **achieves similar functionality through interfaces**, since a class can **implement multiple interfaces**.

```java
interface Printable {
    int COPIES = 10; // by default public, static, final
    void print();    // by default public and abstract
}
```

```java
interface Scannable {
    int RESOLUTION = 300;
    void scan();
}
```

```java
class MultiFunctionPrinter implements Printable, Scannable {
    public void print() {
        System.out.println("Printing " + COPIES + " copy...");
    }

    public void scan() {
        System.out.println("Scanning at " + RESOLUTION + " ...");
    }
}
```

```java
public class OfficeApp {
    public static void main(String[] args) {
        MultiFunctionPrinter mfp = new MultiFunctionPrinter();
        mfp.print();
        mfp.scan();
    }
}
```

Key Concepts Demonstrated:

- Interfaces in Java cannot be instantiated (cannot create an object directly from an interface).
- This is because interfaces do not provide method implementations as they only define method signatures (declaration) without behavior.
- To use an interface, a class must implement it and provide concrete implementations for all its methods.
- it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

- Java do not allow multiple inheritance.

✗
```
class PartTimeStudent extends Employee, Student {
```

- Classes can extend exactly one class and implement any number of interfaces.

✓
```
class PartTimeStudent implements Employee, Student {
```

- Using interfaces we can achieve some thing similar to multiple inheritance.
- An interface can extends another interface.

Exercise 02:

- Create the `Printable` interface. Create two classes Employee and Book with suitable attributes. Have a constructor to assign attributes.
- Implement the printable interface in the Employee and Book class.
- Create MyMain class with a main method and create objects of the Book and Employee classes.
- Print the details of the book and the employee.

```
interface Printable {
  void print();
}
```

Abstract classes vs Interfaces

- Use an abstract class if
  - You want to share the code.
  - Expect to have common methods or properties
  - You want to have access modifiers other than public
  - You want to have properties which not static or not final
- Use an interface if
  - You want Unrelated classes to implement the interfaces.
  - Want to take advantages of multiple inheritances

Static Members

- In Java, class members (attributes and methods) can be defined as static.
- A static member does not belong to any specific object as it belongs to the class itself.
- All objects of that class share the same static variable or method.
- Static members are stored in static memory. It is a common memory area accessible to all objects of the same class.
- This is ideal for defining shared values (e.g., a common college name for all students, or a counter for object count).

Example Scenario: Managing Student Information in a Batch

Suppose we are developing a simple system to **store information about students in your batch**, such as their **student ID** and **name**.

```
class Student {
    private String name;
    private int studentId;
    private String batch;

    public Student(int pStudentId, String pName, String pBatch) {
        this.studentId = pStudentId;
        this.name = pName;
        this.batch = pBatch;
    }

    public void display() {
        System.out.print("ID: " + this.studentId );
        System.out.print(", Name: " + this.name);
        System.out.println(", Batch: " + this.batch);
    }

    public void setBatch(String pBatch) {
        this.batch = pBatch;
    }
}
```

```java
public class StudentApp {
    public static void main(String[] args) {
        Student s1 = new Student(101, "Sunil", "Y1.S2.WD.01");
        Student s2 = new Student(202, "Kamal", "Y1.S2.WD.01");

        s1.display();
        s2.display();

        System.out.println("\nChanging the batch...");
        s1.setBatch("Y1.S2.WD.15");
        s2.setBatch("Y1.S2.WD.15");

        s1.display();
        s2.display();
    }
}
```

The **same batch value must be passed** repeatedly during the creation of each student object. :

If the batch changes later, we need to **manually update it for every object** using a setter method.

Repeating and updating the batch for each student becomes **difficult and time-consuming** when there are **a large number of students**.

Output:

```
ID: 101, Name: Sunil, Batch: Y1.S2.WD.01
ID: 202, Name: Kamal, Batch: Y1.S2.WD.01

Changing the batch...
ID: 101, Name: Sunil, Batch: Y1.S2.WD.15
ID: 202, Name: Kamal, Batch: Y1.S2.WD.15
```

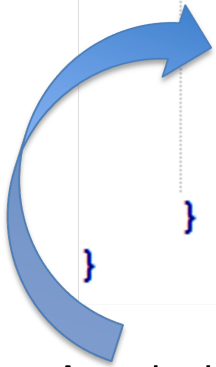
Solution Using static : Since all students in this system belong to the **same batch** (Y1.S2.WD.01), it is more efficient to store the batch name as a **static variable**.

By declaring the batch as a static variable;

- Store the value only once in memory (shared across all student objects).
- Avoid repeating the batch value in every object.
- Easily update the batch in a single place and the change reflects in all student objects automatically.

```
class Student {
    private String name;
    private int studentId;
    private static String batch;

    public Student(int pStudentId, String pName) {
        this.studentId = pStudentId;
        this.name = pName;
    }

    public void display() {
        System.out.print("ID: " + this.studentId );
        System.out.print(", Name: " + this.name);
        System.out.println(", Batch: " + this.batch);
    }

    public void setBatch(String pBatch) {
        this.batch = pBatch;
    }
}
```

```java
public class StudentApp {
    public static void main(String[] args) {
        Student s1 = new Student(101, "Sunil");
        Student s2 = new Student(202, "Kamal");
        s1.setBatch("Y1.S2.WD.01");

        s1.display();
        s2.display();

        System.out.println("\nChanging the batch...");
        s2.setBatch("Y1.S2.WD.15");

        s1.display();
        s2.display();
    }
}
```

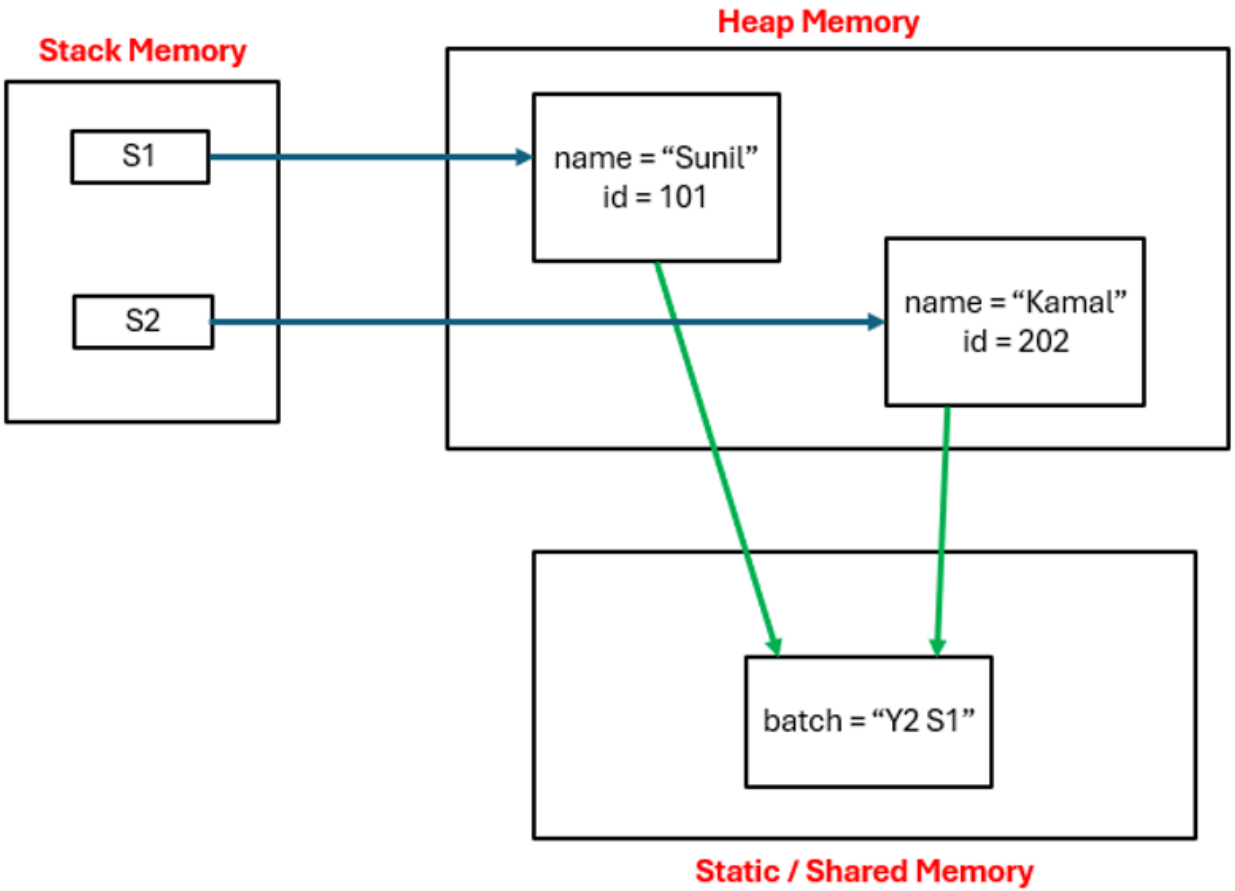


update the batch in a single place and the change reflects in **all student objects automatically**.

Output:

```
ID: 101, Name: Sunil, Batch: Y1.S2.WD.01
ID: 202, Name: Kamal, Batch: Y1.S2.WD.01

Changing the batch...
ID: 101, Name: Sunil, Batch: Y1.S2.WD.15
ID: 202, Name: Kamal, Batch: Y1.S2.WD.15
```

How memory works?



- Variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.
- There will be times when you will want to define a class member that will be used independently of any object of that class.
- When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object.
- You can declare both methods and variables to be **static**.
- The most common example of a **static** member is **main( )**. **main( )** is declared as static because it must be called before any objects exist.

Static Methods

```
class Student {
    private String name;
    private int studentId;
    private static String batch;

    public Student(int pStudentId, String pName) {
        this.studentId = pStudentId;
        this.name = pName;
    }

    public void display() {
        System.out.print("ID: " + this.studentId );
        System.out.print(", Name: " + this.name);
        System.out.println(", Batch: " + this.batch);
    }

    public void setBatch(String pBatch) {
        this.batch = pBatch;
    }

    public static void setBatch2(String pBatch) {
        batch = pBatch;
    }
}
```

setBatch2() is a static method

Static Methods can be called directly using the class name.

```
public class StudentApp {
    public static void main(String[] args) {
        Student.setBatch2("Y1.S2.WD.01");

        Student s1 = new Student(101, "Sunil");
        Student s2 = new Student(202, "Kamal");

        s1.display();
        s2.display();

        System.out.println("\nChanging the batch...");
        s2.setBatch("Y1.S2.WD.15");

        s1.display();
        s2.display();
    }
}
```

- Methods declared as **static** have several restrictions:
  - They can only directly call other **static** methods.
  - They can only directly access **static** data.
  - They cannot refer to **this** or **super** in any way.
- We can use static methods, for actions that are **common across all objects**.

Static Block in Java

- It runs only once, when the class is first loaded into memory, before any objects are created or any static methods are called.
- Executes automatically when the class is loaded by the JVM.
- A **static block** can be used to **initialize static variables**.

```java
class Student {
    private String name;
    private int studentId;
    private static String batch;

    public Student(int pStudentId, String pName) {
        this.studentId = pStudentId;
        this.name = pName;
    }

    public void display() {
        System.out.print("ID: " + this.studentId );
        System.out.print(", Name: " + this.name);
        System.out.println(", Batch: " + this.batch);
    }

    public void setBatch(String pBatch) {
        this.batch = pBatch;
    }

    public static void setBatch2(String pBatch) {
        batch = pBatch;
    }

    static{
        System.out.println("Hello.. I am comming from the static block \n");
    }
}
```

Output:

```
Hello.. I am comming from the static block

ID: 101, Name: Sunil, Batch: Y1.S2.WD.01
ID: 202, Name: Kamal, Batch: Y1.S2.WD.01

Changing the batch...
ID: 101, Name: Sunil, Batch: Y1.S2.WD.15
ID: 202, Name: Kamal, Batch: Y1.S2.WD.15
```

Note: There is no need to modify or explicitly call the static block in the main() method. The static block is executed automatically when the class is loaded into memory, before any objects are created or methods are called.

Thank You!