OOP using Java

Tessema Mengistu (Ph.D.)
Department of Computer Science
Virginia Tech
mengistu@vt.edu

Object Oriented Programming

- Objects and Classes
- Encapsulation
- Inheritance
- Polymorphism
- Exception Handling

Classes and Objects

- The underlying structure of all Java programs is class.
- Anything we wish to represent in Java must be encapsulated in a class
  - defines the “state” and “behavior” of the basic program components known as objects.
- A class essentially serves as a template for an object and behaves like a data type(such as int).

Classes and Objects

Rectangle
length width
perimeter() area()

```
public class Rectangle {

    public double length
    public double width; //


    //Methods to return perimeter and area
    public double perimeter() {
        return 2 *(length+width);
    }
    public double area() {
        return length * width;
    }
}
```

Objects

- An object is an instance of a class uniquely identified by:
  - Its name
  - Defined state
    - Represented by the values of its attributes in a particular time
- Problem solving in java is through the interaction of objects using the defined methods.

Objects

- Creating an object is a two step process:
  - Creating a reference variable

    Syntax:

    ```
    <class idn> <ref. idn>;
    ```

    - For Example : `Circle c1;`
  - Setting or assigning the reference with the newly created object.

    Syntax:

    ```
    <ref. idn> = new <class idn>(…);
    ```

    - For example: `r1 = new Rectangle();`
- The two steps can be done in a single statement
  - `Rectangle c2 = new Rectangle();`

Constructors

- **Constructors** are special methods used to construct an instance of a class.
- They have no return type.
- They have the same name as the class of the Object they are constructing.
- They initialize the state of the Object.
- Call the constructor by preceding it with the **new** keyword.

Default Constructor

- When you do not define a constructor in a class, it implicitly has a constructor with no arguments and an empty body.
  - Called default constructor.
- It initializes the state (fields) of an object to their default values.
- Result: every class has a constructor whether it is defined or not by the programmer.

Multiple Constructors

- A class can have multiple constructors.

```
public class Rectangle {
    public double length
    public double width;
        //Constructor 1
    public Rectangle(){
        length = 10.0;
        width = 10.0;}
      //Constructor 2
    public Rectangle(double l, double w){
          length = l;
             width = w;}
    //Methods to return perimeter and area
    public double perimeter() {
         return 2 *(length+width);
    }
    . . .
```

Using the **`this`** Reference

- Each object has a reference to itself
  - The **`this`** reference
    - Implicitly used to refer to instance variables and methods
- Inside methods
  - If parameter has same name as instance variable
    - Instance variable hidden
  - Use **`this.variableName`** to explicitly refer to the instance variable
  - Use **`variableName`** to refer to the parameter

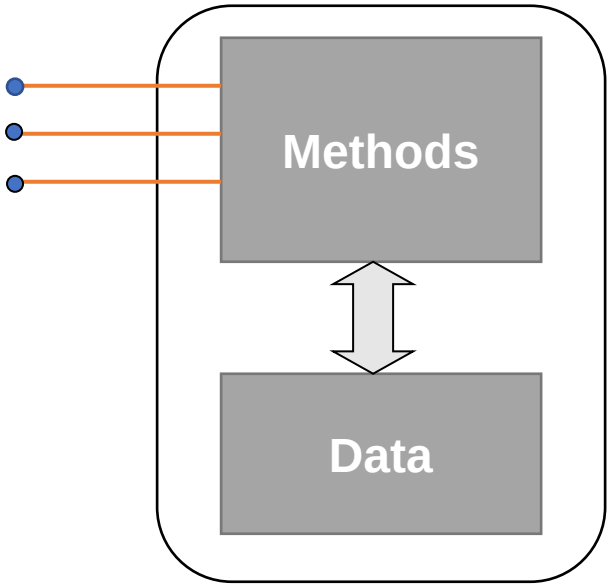
this Keyword

- An object can refer to itself with the **this** keyword

```
public class Rectangle {
    public double length
    public double width;
        //Constructor 1
    public Rectangle(){
        this.length = 10.0;
        this.width = 10.0;
      }
      //Constructor 2
    public Rectangle(double length, double width){
          this.length = length;
          this.width = width;
        }
        . . .
```

What is encapsulation?

- Hiding the data within the class
- Making it available only through the methods
- Each object protects and manages its own data. This is called self-governing.



Why encapsulation?

- To hide the internal implementation details of your class so they can be easily changed
- To protect your class against accidental or willful mistakes
- In general
  - Encapsulation separates the implementation details from the interface

Visibility Modifiers

- In Java we accomplish encapsulation using visibility modifiers.
  - public
    - **least restrictive**
    - Can be directly referenced from outside of an object.
    - visible to any class in the Java program
  - private
    - **most restrictive**
    - **Can't be accessed by anywhere outside the enclosing class**
    - Cannot be referenced externally.
    - Instance data should be defined private.
  - **Package - default**
    - **Intermediate b/n private and public**
    - access only to classes in the same package
  - protected
    - access to classes in the same package and to all subclasses

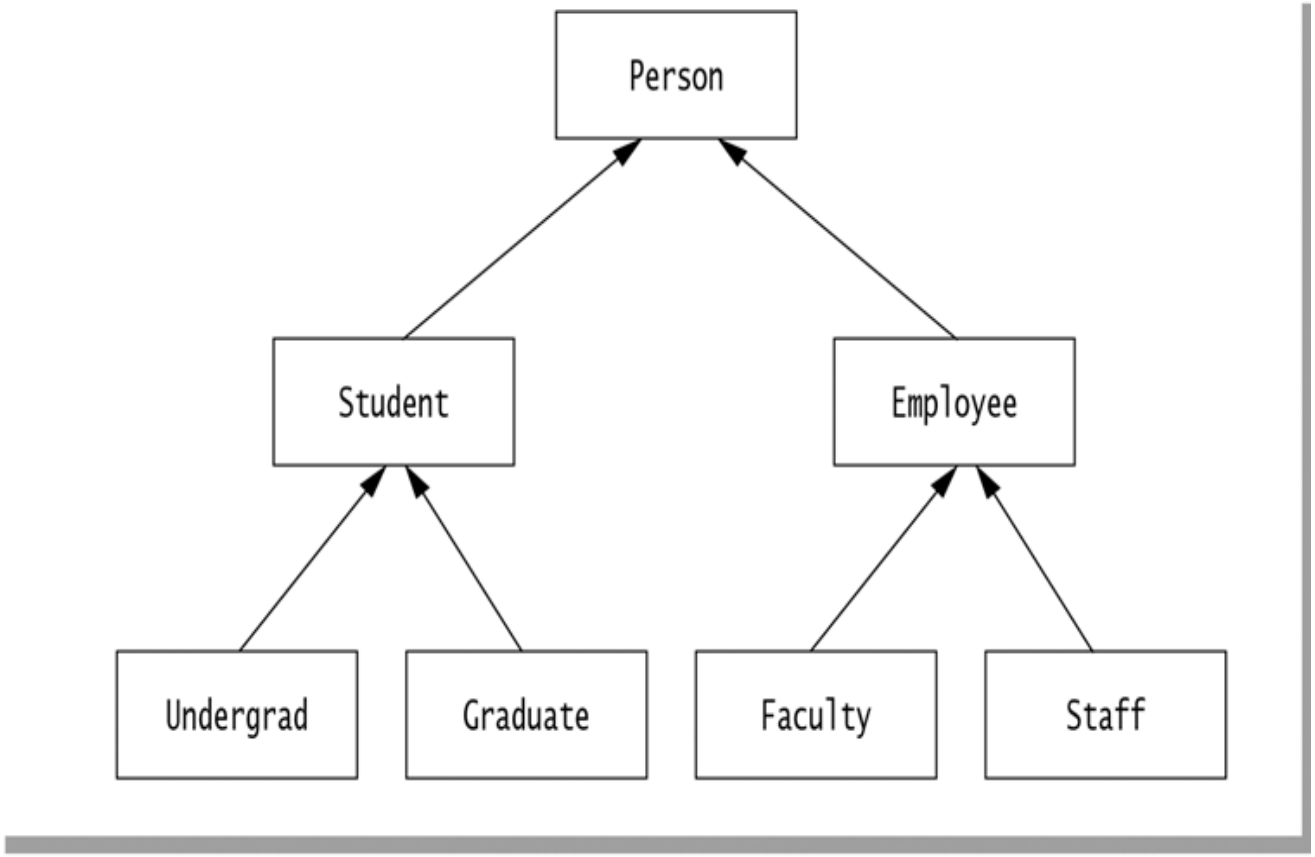
Inheritance

- Inheritance allows a software developer to derive a new class from an existing one
- For the purpose of
  - reuse
  - enhancement,
  - adaptation, etc
- The existing class is called the *parent class* or *superclass*
- The derived class is called the *child class* or *subclass*.
- Super classes are more generic and sub classes are specific version of the super classes

Inheritance

- Two ways of expressing class relationships
  - Generalization/Specialization
    - ‘is a” relationship
    - Circle is a shape
  - Whole-part
    - Part of or “has a” relationship
    - Employee class has a BirthDate class
    - Called aggregation
- Inheritance creates an is-a relationship

Inheritance



Inheritance

```
public Person {
          // Person class members
             }
public Student extends Person {
         // inherited person members
         // Student class members
  }
 public Undergrad extends Student {
         // inherited Person members
         // inherited Student members
         //Undergrad class members
  }
```

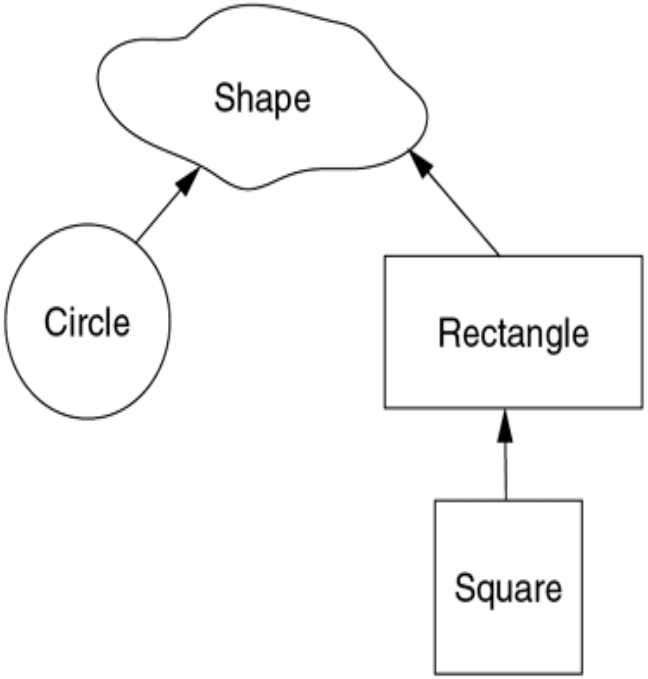
Multiple Inheritance

- Java supports *single inheritance*, meaning that a sub class can have only one super class
  - *Multiple inheritance* allows a class to be derived from two or more classes, inheriting the members of all parents
  - Collisions, such as the same variable name in two parents, have to be resolved
- In most cases, the use of interfaces gives us aspects of multiple inheritance without the overhead (will discuss later)

Polymorphism

- The term *polymorphism* literally means "having many forms”
- Polymorphism is an object-oriented concept that allows us to create versatile software designs
- In OOP, polymorphism promotes code reuse by calling the method in a generic way.
- For example we can calculate the area by calling the area() method on all shape objects. But depending on the type of shape(whether Circle or Rectangle) the correct version of area() will be called.

Polymorphism



Polymorphism

- Suppose we create the following reference variable:

```
Shape myShapes;
```

- Java allows this reference to point to an Shape object, or to any object of <u>any compatible type</u>
- This compatibility can be established using inheritance or using interfaces
- Careful use of polymorphic references can lead to elegant, robust software designs

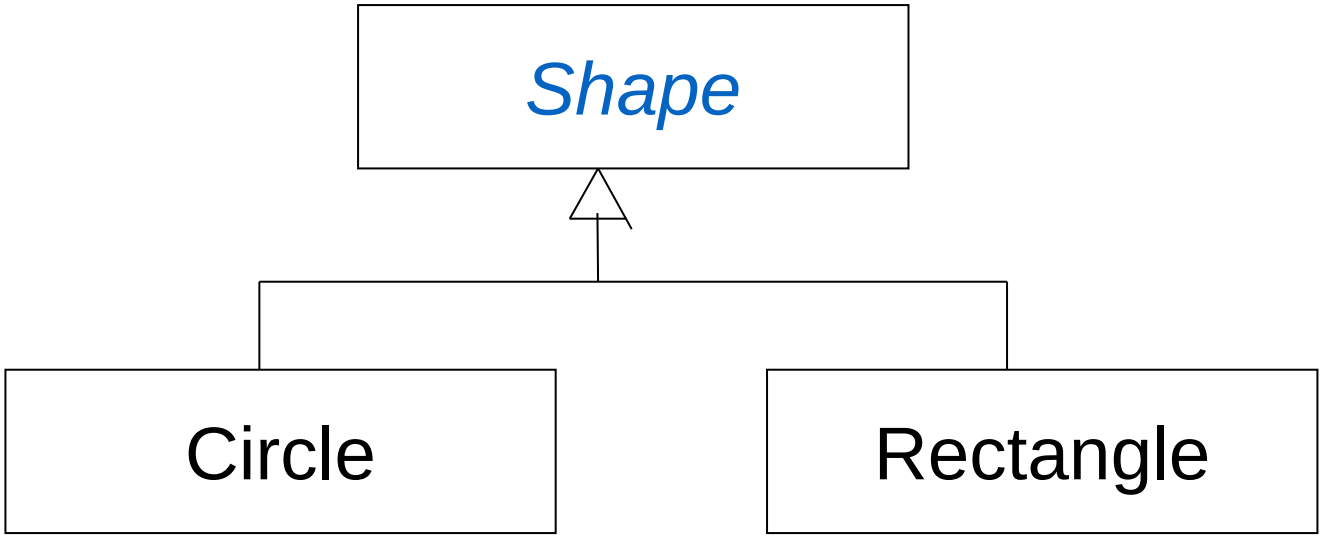
Abstract Class Syntax

```
abstract class Class_Name
        {
                ...
                abstract Type MethodName1();
                …
                Type Method2()
                        {
                                // method body
                        }
        }
```

- When a class contains one or more abstract methods, it should be declared as abstract class.
- The abstract methods of an abstract class must be defined in its subclass (called **concrete class**).
- We cannot declare abstract constructors or abstract static methods.

Abstract Class -Example

- Shape is a abstract class.



The Shape Abstract Class

```
public abstract class Shape {
      public abstract double area();
      public void move() { // non-abstract method
            // implementation
        }
}
```

- Is the following statement valid?
  - Shape s = new Shape();

- No. It is illegal because the Shape class is an abstract class, which cannot be instantiated to create its objects.

Abstract Classes

```
public Circle extends Shape {
    protected double r;
     protected static final double PI = 3.1415;
     public Circle() { r = 1.0; }
     public double area() {
             return PI * r * r; }
             …
           }
public Rectangle extends Shape {
     protected double w, h;
     public Rectangle() {
              w = 0.0; h=0.0; }
     public double area() {
                  return w * h; }
}
```

Abstract Classes Properties

- A class with one or more abstract methods is automatically abstract and it cannot be instantiated.
- A class declared abstract, even with no abstract methods can not be instantiated.
- A subclass of an abstract class can be instantiated if it overrides all abstract methods by implementing them.
- A subclass that does not implement all of the superclass abstract methods is itself abstract; and it cannot be instantiated.

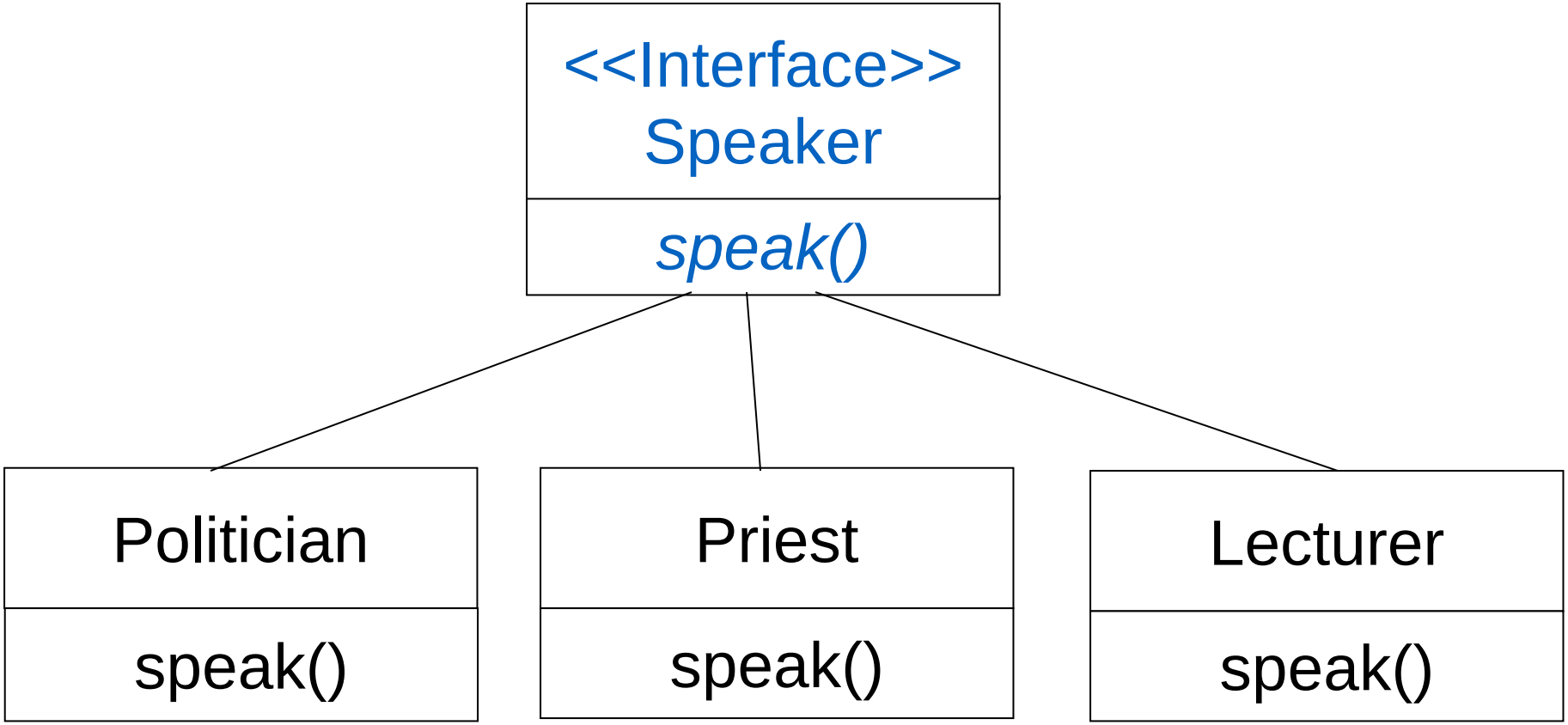
Interfaces

- ***Interface*** is a conceptual entity similar to a Abstract class.
- Can contain only **constants (final variables)** and **abstract method** (no implementation) - Different from Abstract classes.
- Use when a number of classes share a common interface.
- Each class should implement the interface.

Interfaces: An informal way of realizing multiple inheritance

- An interface is basically a kind of class—it contains methods and variables, but they have to be only abstract classes and final fields/variables.
- Therefore, it is the responsibility of the class that implements an interface to supply the code for methods.
- A class can implement any number of interfaces, but cannot extend more than one class at a time.
- Therefore, interfaces are considered as an informal way of realizing multiple inheritance in Java.

Interface - Example



Interfaces Definition

- Syntax (appears like abstract class):

```
interface InterfaceName {
   // Constant/Final Variable Declaration
   // Methods Declaration - only method body
}
```

- Example:

```
interface Speaker {
      public void speak( );
}
```

Implementing Interfaces

- Interfaces are used like super-classes who properties are inherited by classes.
- This is achieved by creating a class that implements the given interface as follows:

```
class ClassName implements InterfaceName [,
InterfaceName2, …]
{
  // Body of Class
}
```

Implementing Interfaces Example

```
class  Politician implements Speaker {
    public void speak(){
      System.out.println("Talk politics");
    }
}
```

```
class  Priest implements Speaker {
    public void speak(){
       System.out.println("Preaches");
    }
}
```

```
class  Lecturer implements Speaker {
    public void speak(){
      System.out.println("Talks Object
Oriented Design and Programming!");
    }}
```

Exceptions

- An exception is anything **unexpected**, or **unusual**, about a program's execution.
- It is a situation that is **not normal**.
- An exception could be:
  - caused by programmer error
    - For example, division by zero
  - caused by user error
    - For example, a user could enter a negative value for age
  - Caused by physical resources
    - For example, run out of memory

Java Exception class

- Exceptions can be:
  - **Checked exception** – and exception that is checked by the compiler during compilation time.
    - Should be handled by the programmer

```
import java.io.*;
public class Test {
     public static void main(String  args[]){
        File input = new File("c:/logs.txt");
      FileReader fr = new FileReader(input);
     }
}
```

```
C:\> javac Test.java
Test.java:5: error: unreported exception FileNotFoundException; must be caught
or declared to be thrown
      FileReader fr = new FileReader(input);
                                    ^
1 error
```

Exception Handling

- When an exception occurs the normal flow of the program ins disrupted and the program terminates abnormally
  - -Exceptions should be handled
- **Exception handling** is the process of dealing with the exceptional circumstance.
- Exception handling code must decide how to either return the program's execution to normality, or terminate the program.
- Exceptions are **thrown.**
- An exception handler **catches** an exception.

Catching Exceptions

- Exceptions are caught using a **`try`**…**`catch`** block.
- A section of code is **tried**, and any exceptions that are thrown are **caught** by catch blocks.

```
try {
        … normal program code
    }
catch(ExceptionType exceptionName) {
        … exception handling code
}
```

- Syntax

```
try
      {
          resource acquisition statements
       }
catch()    {
                …
       }
finally
      {
          resource release statements
      }
```

```
import java.util.*;
class Test {
public static int div(int a , int b) {
      if(b==0)
              throw new Exception()
      else
                return a/b;
               }
public static void main(String  args[]){
       Scanner input = new Scanner(System.in);
try{
      System.out.println("enter the first number");
      int num1=  input.nextInt();
      System.out.println("enter the second number");
      int num2=  input.nextInt();
      int result = div(num1, num2);
      System.out.println(result);
 }
catch (Exception e){
         System.err.println(e);
             }      }    }
```

References

- *Data Structures & Problem Solving Using Java,* 4th Edition, by Mark Allen Weiss
- Data Structures and Abstraction with Java, 5th Edition, Frank Carrano