

Review of Object-Oriented Concepts in JAVA

- Object-Oriented Concepts supported by JAVA.
- Advantages of Object-Orientation.
- Inheritance.
- Abstract Classes.
- Interfaces.
- Review Questions.

Object-Oriented Concepts supported by JAVA

- Java provides explicit support for many of the fundamental Object-Oriented Concepts. Some of these are:
 - **Classification**: Grouping related things together. This is supported through classes, inheritance & packages.
 - **Encapsulation**: Representing data and the set of operations on the data as a single entity - exactly what classes do.
 - **Information Hiding**: An object should be in full control of its data, granting specific access only to whom it wishes.
 - **Inheritance**: Java allows related classes to be organized in a hierarchical manner using the extends keyword.
 - **Polymorphism**: Same code behaves differently at different times during execution. This is due to dynamic binding.

Advantages of Object-Orientation.

- A number of advantages can be derived as a result of these object-oriented features. Some of these are:
 - **Reusability**: Rather than endlessly rewriting same piece of code, we write it once and use it or inherit it as needed.
 - **Extensibility**: A class can be extended without affecting its users provided the user-interface remains the same.
 - **Maintainability**: Again, once the user-interface does not changed, the implementation can be changed at will.
 - **Security**: Thanks to information hiding, a user can only access the information he has been allowed to access.
 - **Abstraction**: Classification and Encapsulation allow portrayal of real-world problems in a simplified model.

Review of inheritance

- Suppose we have the following **Employee** class:

```
class Employee {
    protected String name;
    protected double payRate;
    public Employee(String name, double payRate) {
        this.name = name;
        this.payRate = payRate;
    }
    public String getName() {return name;}
    public void setPayRate(double newRate) {
        payRate = newRate;
    }
    public double pay() {return payRate;}
    public void print() {
        System.out.println("Name: " + name);
        System.out.println("Pay Rate: "+payRate);
    }
}
```

Review of inheritance (contd.)

- Now, suppose we wish to define another class to represent a part-time employee whose salary is paid per hour. We inherit from the **Employee** class as follows:

```
class HourlyEmployee extends Employee {
    private int hours;
    public HourlyEmployee(String hName, double hRate) {
        super(hName, hRate);
        hours = 0;
    }
    public void addHours(int moreHours) {hours += moreHours;}
    public double pay() {return payRate * hours;}
    public void print() {
        super.print();
        System.out.println("Current hours: " + hours);
    }
}
```

Notes about Inheritance

- We observe the following from the examples on inheritance:
 - Methods and instance variables of the super class are inherited by subclasses, thus allowing for code reuse.
 - A subclass can define additional instance variables (e.g. hours) and additional methods (e.g. addHours).
 - A subclass can override some of the methods of the super class to make them behave differently (e.g. the pay & print)
 - Constructors are not inherited, but can be called using the super keyword. such a call must be the first statement.
 - If the constructor of the super class is not called, then the compiler inserts a call to the default constructor -watch out!
 - super may also be used to call a method of the super class.

Review of Abstract Classes

- Inheritance enforces hierarchical organization, the benefit of which are: reusability, type sharing and polymorphism.
- Java uses Abstract classes & Interfaces to further strengthen the idea of inheritance.
- To see the role of abstract of classes, suppose that the **pay** method is not implemented in the **HourlyEmployee** subclass.
- Obviously, the **pay** method in the **Employee** class will be assumed, which will lead to wrong result.
- One solution is to remove the **pay** method out and put it in another extension of the Employee class, **MonthlyEmployee**.
- The problem with this solution is that it does not force subclasses of **Employee** class to implement the **pay** method.

Review of Abstract Classes (Cont'd)

- The solution is to declare the pay method of the Employee class as abstract, thus, making the class abstract.

```
abstract class Employee {
    protected String name;
    protected double payRate;
    public Employee(String empName, double empRate) {
        name = empName;
        payRate = empRate;
    }
    public String getName() {return name;}
    public void setPayRate(double newRate) {payRate = newRate;}

    abstract public double pay();

    public void print() {
        System.out.println("Name: " + name);
        System.out.println("Pay Rate: "+payRate);
    }
}
```

Review of Abstract Classes (Cont'd)

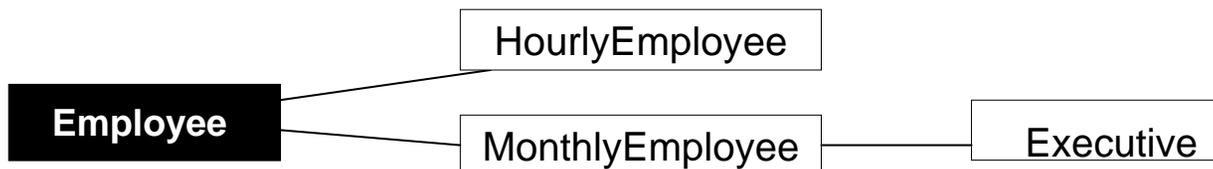
- The following extends the Employee abstract class to get MonthlyEmployee class.

```
class MonthlyEmployee extends Employee {
    public MonthlyEmployee(String empName, double empRate) {
        super(empName, empRate);
    }
    public double pay() {
        return payRate;
    }
}
```

- The next example extends the MonthlyEmployee class to get the Executive class.

Review of Abstract Classes (Cont'd)

```
class Executive extends MonthlyEmployee {
    private double bonus;
    public Executive(String exName, double exRate) {
        super(exName, exRate);
        bonus = 0;
    }
    public void awardBonus(double amount) {
        bonus = amount;
    }
    public double pay() {
        double paycheck = super.pay() + bonus;
        bonus = 0;
        return paycheck;
    }
    public void print() {
        super.print();
        System.out.println("Current bonus: " + bonus);
    }
}
```



Review of Abstract Classes (Cont'd)

- The following further illustrates the advantages of organizing classes using inheritance - same type, polymorphism, etc.

```
public class TestAbstractClass {
    public static void main(String[] args) {
        Employee[] list = new Employee[3];
        list[0] = new Executive("Jarallah Al-Ghamdi", 50000);
        list[1] = new HourlyEmployee("Azmat Ansari", 120);
        list[2] = new MonthlyEmployee("Sahalu Junaidu", 9000);
        ((Executive)list[0]).awardBonus(11000);
        for(int i = 0; i < list.length; i++)
            if(list[i] instanceof HourlyEmployee)
                ((HourlyEmployee)list[i]).addHours(60);
        for(int i = 0; i < list.length; i++) {
            list[i].print();
            System.out.println("Paid: " + list[i].pay());
            System.out.println("*****");
        }
    }
}
```

```
Name: Jarallah Al-Ghamdi
Pay Rate: 50000.0
Current bonus: 11000.0
Paid: 61000.0
*****
Name: Azmat Ansari
Pay Rate: 120.0
Current hours: 60
Paid: 7200.0
*****
Name: Sahalu Junaidu
Pay Rate: 9000.0
Paid: 9000.0
*****
```

The Program Output

Review of Interfaces

- Interfaces are not classes, they are entirely a separate entity.
- They provide a list of abstract methods which **MUST** be implemented by a class that implements the interface.
- Unlike abstract classes which may contain implementation of some of the methods, interfaces provide **NO** implementation.
- Like abstract classes, the purpose of interfaces is to provide organizational structure.
- More importantly, interfaces are here to provide a kind of "multiple inheritance" which is not supported in Java.
- If both parents of a child implement a method, which one does the child inherit? - Multiple inheritance confusion.
- Interfaces allow a child to be both of type A and B.

Review of Interfaces (contd.)

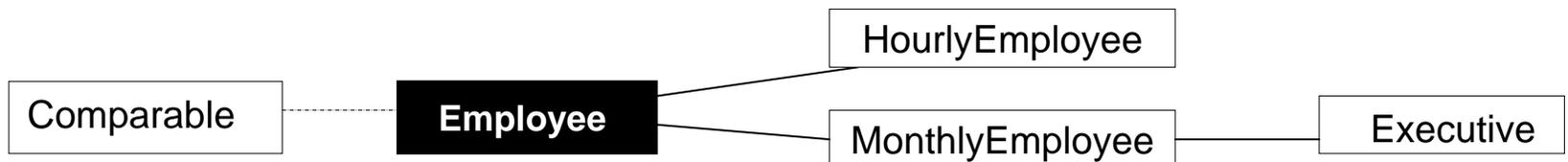
- Recall that Java has the Comparable interface defined as:

```
interface Comparable {  
    int compareTo(Object o);  
}
```

- Recall also that java has the java.util.Arrays class, which has a sort method that can sort any array whose contents are either primitive values or Comparable objects.
- Thus, to sort our list of Employee objects, all we need is to modify the Employee class to implement the Comparable interface.
- Notice that this will work even if the Employee class is extending another class or implementing another interface.
- This modification is shown in the next page.

Review of Interfaces (contd.)

```
abstract class Employee implements Comparable {
    protected String name;
    protected double payRate;
    public Employee(String empName, double empRate) {
        name = empName;
        payRate = empRate;
    }
    public String getName() {return name;}
    public void setPayRate(double newRate) {
        payRate = newRate;
    }
    abstract public double pay();
    public int compareTo(Object o) {
        Employee e = (Employee) o;
        return name.compareTo( e.getName());
    }
}
```



Review of Interfaces (contd.)

- Since Employee class implements the Comparable interface, the array of employees can now be sorted as shown below:

```
import java.util.Arrays;
public class TestInterface {
    public static void main(String[] args) {
        Employee[] list = new Employee[3];
        list[0] = new Executive("Jarallah Al-Ghamdi", 50000);
        list[1] = new HourlyEmployee("Azmat Ansari", 120);
        list[2] = new MonthlyEmployee("Sahalu Junaidu", 9000);
        ((Executive)list[0]).awardBonus(11000);
        for(int i = 0; i < list.length; i++)
            if(list[i] instanceof HourlyEmployee)
                ((HourlyEmployee)list[i]).addHours(60);
        Arrays.sort(list);
        for(int i = 0; i < list.length; i++) {
            list[i].print();
            System.out.println("Paid: " + list[i].pay());
            System.out.println("*****");
        }
    }
}
```

The program output

```
Name: Azmat Ansari
Pay Rate: 120.0
Current hours: 60
Paid: 7200.0
*****
Name: Jarallah Al-Ghamdi
Pay Rate: 50000.0
Current bonus: 11000.0
Paid: 61000.0
*****
Name: Sahalu Junaidu
Pay Rate: 9000.0
Paid: 9000.0
*****
```

Review Questions

- How does an interface differ from an abstract class?
- Why does Java not support multiple inheritance? What feature of Java helps realize the benefits of multiple inheritance?
- An Abstract class must contain at least one abstract method, (true or false)?
- A subclass typically represents a larger number of objects than its super class, (true or false)?
- A subclass typically encapsulates less functionality than its super class does, (true or false)?
- An instance of a class can be assigned to a variable of type any of the interfaces the class implements, (true or false)?