OBJECT ORIENTED PROGRAMMING

Affefah Qureshi

Department of Computer Science

Iqra University, Islamabad Campus.

ABSTRACT CLASSES

- We may want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.
- That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.
- Such a class determines the nature of methods that the subclasses must implement.
- One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method.
- If a superclass is unable to create a meaningful implementation for a method, we can handle this situation two ways.
  - One way is to simply have it report a warning message. While this approach can be useful in certain situations—such as debugging—it is not usually appropriate.
  - We may have methods which must be overridden by the subclass in order for the subclass to have any meaning.
- That is, if we want some way to ensure that a subclass does, indeed, override all necessary methods.
  - **Java's solution to this problem is the abstract method.**

ABSTRACT CLASSES

- To declare an abstract method, we use this general form:
  - **abstract type name(parameter-list);**
- These methods are sometimes referred to as subclasses responsibility because they have no implementation specified in the superclass
- **Any class that contains one or more abstract methods must also be declared abstract.**
- To declare a class abstract, we simply use the abstract keyword in front of the class keyword at the beginning of the class declaration.
- There can be no objects of an abstract class, because an abstract class is not fully defined.
- Also, we cannot declare abstract constructors, or abstract static methods.
- Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared abstract..

ABSTRACT CLASS EXAMPLE

```
abstract class A {
  abstract void callme();
  // concrete methods are still allowed in abstract classes
  void callmetoo() {
    System.out.println("This is a concrete method.");
  }
}
class B extends A {
  void callme() {
    System.out.println("B's implementation of callme.");
  } }
class AbstractDemo {
  public static void main(String args[]) {
    B b = new B();
    b.callme();
    b.callmetoo(); } }
```

Output:

B's implementation of callme.

ABSTRACT CLASS EXAMPLE

Output:

Not working
Working as employee!!

```
//abstract class
public abstract class Person {
private String name;
private String gender;
public Person(String nm, String
gen){
        this.name=nm;
        this.gender=gen; }
//abstract method
public abstract void work(); } }

public class Employee extends
Person {
private int empId;
public Employee(String nm, String
gen, int id){
super(nm, gen);
this.empId=id; }

public void work() {
if(empId == 0){
System.out.println("Not working");
}
else{
System.out.println("Working as
employee!!"); } }

public static void main(String
args[]){

Person student = new
Employee("Dove","Female",0);

Person employee = new
Employee("Pankaj","Male",123);

student.work();
employee.work();

}
```

ABSTRACT CLASSES

- Although abstract classes **cannot be used to instantiate objects**, they **can be used to create object references**, because Java's approach to run-time polymorphism is implemented through the use of superclass references.

ABSTRACT CLASS EXAMPLE

```
abstract class Figure {
  double dim1;
  double dim2;
  Figure(double a, double b) {
    dim1 = a;
    dim2 = b;
  }
  // area is now an abstract method
  abstract double area();
}
class Rectangle extends Figure {
  Rectangle(double a, double b) {
    super(a, b);  }

double area() {
    System.out.println("Inside Area for
    Rectangle.");
    return dim1 * dim2;  } }

class Triangle extends Figure {
  Triangle(double a, double b) {
    super(a, b);
  }

double area() {
    System.
    Triangle.");
    return dim1 * dim2 / 2;  } }

class AbstractAreas {
  public static void main(String args[]) {
  // Figure f = new Figure(10, 10);
// illegal now

    Rectangle r = new Rectangle(9, 5);
    Triangle t = new Triangle(10, 8);

    Figure figref; // this is OK, no object is
    created

    figref = r;

    System.out.println("Area is " +
    figref.area());

    figref = t;

    System.out.println("Area is " +
    figref.area()); } }
```

Output:

```
Inside Area for Rectangle.
Area is 45.0
Inside Area for Triangle.
Area is 40.0
```

USING FINAL WITH INHERITANCE

- The keyword final has three uses:
  - First, it can be used to create the equivalent of a named constant.
  - The other two uses of final apply to inheritance.
    - To prevent overriding
    - To prevent inheritance

USING FINAL TO PREVENT OVERRIDING

- While method overriding is one of Java's most powerful features, there will be times when we will want to prevent it from occurring.
- To disallow a method from being overridden, specify final as a modifier at the start of its declaration.
- Methods declared as **final** cannot be overridden.

USING FINAL TO PREVENT OVERRIDING

```
class A {
  final void meth() {
    System.out.println("This is a final method.");
  }
}

class B extends A {
  void meth() { // ERROR! Can't override.
    System.out.println("Illegal!");
  }
}
```

USING FINAL TO PREVENT INHERITENCE

- Sometimes we will want to prevent a class from being inherited.
- To do this, precede the class declaration with **final**.
- Declaring a class as final implicitly declares all of its methods as final, too.
- As you might expect, it is **illegal to declare a class as both abstract and final**
  - Since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations

USING FINAL TO PREVENT OVERRIDING

```
final class A {
  // ...
}


// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
  // ...
}
```