CSE-2102
Object Oriented Programming

Dr. Muhammad Ibrahim
Assistant Professor
Dept. of Computer Science and Engineering
University of Dhaka

Restrictions on Method Overriding

- Overridden method (i.e. declared in subclass) must not be more restrictive.
- Private, static and final methods of superclass cannot be overridden.
- Only methods can be overridden, not the variables (data members).

Examples: next slides

Restrictions on Method Overriding

```
class A{
  protected void msg(){System.out.println("Hello java");}
}
public class Simple extends A{
void msg(){System.out.println("Hello java");}  //Compile time error
 public static void main(String args[]){
   Simple obj=new Simple();
   obj.msg();
   }
}
```

The default modifier is more restrictive than protected. That is why, there is a compile-time error.

Restrictions on Method Overriding

```
class A{
  int n = 10;
}
class B extends A{
  int n = 1000;
}

public class demo {

 public static void main(String args[]){
   A ob = new A();  System.out.println(ob.n); // prints 10
    ob = new B();   System.out.println(ob.n); // still prints 10
}
```

Restrictions on Method Overriding: A Confusing Story?

```
class myclass {
      int i=10;
    void f(){
        System.out.println("Inside f() of myclass, i: " + i);
    }
}

public class yourclass extends myclass {
    int i = 100000;
    void f(){
        System.out.println("Inside f() of yourclass, i: " + i);
        System.out.println("Inside f() of yourclass, super.i: " + super.i);
    }
}
```

Restrictions on Method Overriding: A Confusing Story?

```
public static void main(String []args){
        myclass m = new myclass();
        System.out.println("Referencing a superclass object using superclass
variable: " + m.i);
        m.f();
        m = new yourclass();
        System.out.println("Referencing a subclass object using superclass
variable: " + m.i);
        m.f();
     }
```

Output:

```
Referencing a superclass object using superclass variable: 10
Inside f() of myclass, i: 10
Referencing a subclass object using superclass variable: 10
Inside f() of yourclass, i: 100000
Inside f() of yourclass, super.i: 10
```

Same Members in Superclass and Subclass

Compare and Contrast with C++

```
#include <iostream>
using namespace std;
class myclass {
    public:
    int i=10, j = 20;
    void f(){
        cout << "Inside f() of myclass, i: "  << i << endl;
    }
};
class yourclass : public myclass {
    public:
    int i = 100000;
    void f(){
        cout << "Inside f() of yourclass, i: " << i << endl;
    }
};
```

Same Members in Superclass and Subclass

Compare and Contrast with C++

```
int main (void) {
  myclass m;
  myclass *p;
  yourclass y;
  p = &m;
  cout << "Pointing to a superclass object using superclass variable: " << p->i << endl;
  p->f();
  p = &y;
  cout << "Pointing to a subclass object using superclass variable: " << p->i << endl;
  p->f();
  return 0;
}
```

Output:

```
Pointing to a superclass object using superclass variable: 10
Inside f() of myclass, i: 10
Pointing to a subclass object using superclass variable: 10
Inside f() of myclass, i: 10
```

Next Topics

“final” keyword

“Object” class

Abstract method

Abstract class

The Keyword “final”

- 1st use of “final”: to declare immutable variables.
- 2nd use: to prevent overriding.
  - Oftentimes we need to make sure that some methods are not overridden under any circumstances.
  - Type “final” before a method to make it non-overridable.
  - Sidenote: Normally Java resolves method invocation at runtime, using late binding scheme. But call of a final method is resolved in compiler time (oftentimes an inline version of a final method is generated by the compiler).
- 3rd use: to prevent inheritance.
  - Sometimes we need to make sure that some classes are not inherited under any circumstances.
  - Type “final” before the class definition to make it non-inheritable

Ways to Prevent Overriding

- Using final keyword
- Using private access modifier
- Using static keyword
- …

The Class “Object”

- Object is a special class, which is inherited by all other classes.
  - `class myclass (extends Object)  {    …   }`
- So a reference variable of type object can refer to any other class
  - `Object ob = new myclass();`
- Since arrays in Java are implemented as classes, an Object variable can refer to any array.
  - `Object ob; int array[] = {44, 4}; ob = array;`
- Object defines nine methods (check JDK version to get up-to-date info).
  - Of which four are final, so non-overridable.
  - Others can be overridden.

The Class “Object”

- Two important methods:
  - `boolean equals(Object ob)`: returns true if the calling object and the parameter object are the “same”. The matching depends on the specific object being compared. For example, for string objects the two strings are compared character by character.
    - Can be overridden
  - `String toString()`: returns a description of the calling object, i.e., the object that invokes the method call.
    - Is often overridden to describe a customized definition of the object.

Local Variable Type Inference (from JDK 10)

- Recall that we can declare `var ob = new myclass();`
- If the variable is of superclass type, that will be the inferred type of the variable, not the type of the object.
  - See the example of the textbook (Chapter 8 of 12th Ed.).
  - So now you may avoid using `var` when using inheritance to avoid confusion.

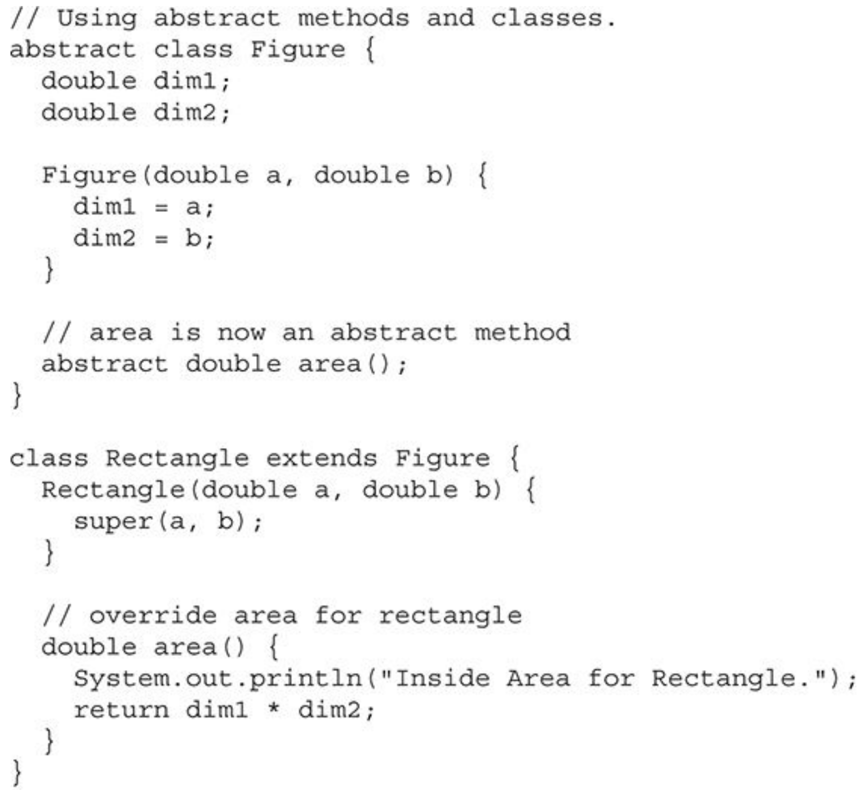
Reading materials: up to Chapter 8 of the textbook.

Abstract Method and Abstract Class

- Method: `abstract` *`ret-type`* `method-name (param-list);`
- Class: `abstract classname { … }`
- If a class in a method is declared abstract, the class must be declared as abstract too.
- No object of abstract class can be declared.
- Abstract class may have data members.
- Reference variable of abstract can be declared (and used to refer to subclass objects).
- No abstract constructors, no abstract static methods.
- Static members of an abstract class can be accessed using `<classname>.<member_name>`notation.
- A subclass must define all abstract methods of an abstract class, otherwise must declare itself abstract.

Often we need to use some code but that code is not self-sufficient, rather the code (abstract class) works as a placeholder, and facilitates “superclass variable referring to subclass objects”.

Abstract Class Example (from textbook)



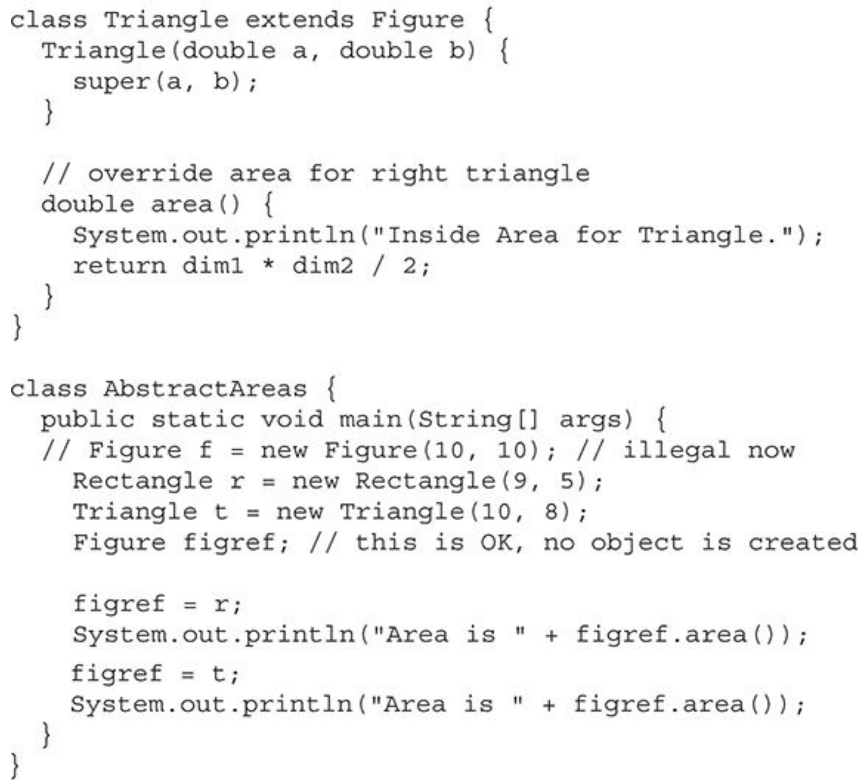
```
// Using abstract methods and classes.
abstract class Figure {
  double dim1;
  double dim2;

  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);
  }

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



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

  // override area for right triangle
  double area() {
    System.out.println("Inside Area for 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());
  }
}
```

Next Topics

- Interface
  - Motivation
  - Implementation
  - Referencing with an interface variable
  - Extending an interface
- Package

Interface

- Interface is yet another construct of Java (in addition to class and package) that facilitates encapsulation and code reuse.
- An interface is similar to a class with some additional restrictions.
- Recall that Java doesn't allow one class to inherit more than one class.
  - Why? Because two classes may have methods of same prototype, and if a third class inherits both of them, two versions of a method having the same prototype may be inherited here which causes a selection problem.
- However, in real-life scenarios, sometimes programmers do feel a need to "inherit" more than one classes.
  - Interfaces provide a way out to fulfil this need.
- So why the problem mentioned above is not present in the interfaces?
  - Because interfaces don't implement the methods, so no question of facing the problem of inheriting two different implementations of a same method.

Interface

- Interface can have both data and method.
- Only static and final data are allowed to be declared. The data are implicitly static and final.
  - Data in interface are not very frequently used.
- All data and methods are implicitly public.
- Static members of an interface can be accessed using `<interface_name>.<member_name>`notation.
- All methods in an interface are by default declared as abstract with no body.
  - Some exceptions are allowed since JDK 8, which will be discussed later.
- So an interface must be *implemented* by some classes.
  - The implementing class must define all the methods of the interface. If it does not, then the implementing class itself should be declared as abstract.
  - The methods implemented in a class must be public.
- Syntax: `access-specifier interface interface-name { ... }`

Interface Example

```
interface A {
    int a1 = 10;
    public void af();
}
interface B {
    int a1 = 100;
    public void af();
}
class C {
    public void af(){
        System.out.println("af() in class C");
    }
}
```

Output:

```
class myclass extends C implements A, B{
    public void af(){
        System.out.println("af() in myclass");
    }
    public static void main(String[] args) {
        System.out.println(B.a1);
        myclass ob = new myclass();
        ob.af();
        A o1 = ob;
        ob.af();
        B o2 = ob;
        ob.af();
        C o3 = ob;
        ob.af();
        C o4 = new C();
        o4.af();
        //super.af(); non-static method cannot
//be called inside a static context
    }
}
```

Interface Example

```
interface A {
    int a1 = 10;
    public void af();
}
interface B {
    int a1 = 100;
    public void af();
}
class C {
    public void af(){
        System.out.println("af() in class C");
    }
}
```

Output:
100
af() in myclass
af() in myclass
af() in myclass
af() in myclass
af() in class C

```
class myclass extends C implements A, B{
    public void af(){
        System.out.println("af() in myclass");
    }
    public static void main(String[] args) {
        System.out.println(B.a1);
        myclass ob = new myclass();
        ob.af();
        A o1 = ob;
        o1.af();
        B o2 = ob;
        o2.af();
        C o3 = ob;
        o3.af();
        C o4 = new C();
        o4.af();
        //super.af(); non-static method cannot
be called inside a static context
    }
}
```

Interface

- An interface tells what a class must do, but not how to do
    - The class(es) that implements it specifies how to do.
- Any number of classes can implement an interface.
    - Obviously, the method signatures must match between class and interface.
- Importantly, a class can implement more than one interface.
    - Sounds contrast with inheritance? Yes, indeed.
    - If two interfaces that are implemented by a single class have the same method, then a single implementation of the said method will be in the implementing class.
- To implement an interface, a class must implement all of the methods of the interface, or declare itself as abstract.

Interface

- A class can extend another class and implement interfaces at the same time.
  - However, obviously an interface cannot extend a class.
  - The class can have additional data and methods and it usually has.
- A .java file can contain at most one public interface.
  - Recall that a class can have one of two access modes: public or default.
  - Interfaces also follow the same rule.
- Also recall that public means the class/interface is accessible to entire world, and default case means it is acknowledged in only it's own package.
- An implementing class can obviously add its own member functions besides implementing (abstract) methods of interfaces.

Extending an Interface

- Interfaces can be extended, just like classes.
    - When a class implements an interface that extends another interface, the class must implement all the methods - both original and inherited - of the implemented interface.
- Nested Interface: possible like nested class, but not much used.
    - Your self-study.
- Note: In JDK 8, interfaces do provide a mechanism to write the body of its methods. Since this feature is not very much used till date, so we would omit that discussion.

Referencing with an Interface Variable

Recall that a superclass reference variable can refer to a subclass object. Similar mechanism is available in interface.

- A reference variable of type interface can refer to an object of implementing class.
  - But this variable cannot have access to those members of object that are added.
- Like the superclass-subclass scenario, when an implemented method will be invoked using the said interface reference variable, the type of object will decide which version of the method (among several classes) will be called.
- This method dispatch is resolved dynamically, i.e., in runtime. That means that the code that is calling a method doesn't need to know during compilation anything about the class that contains the method.
  - “The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them. The calling code can dispatch through an interface without having to know anything about the *callee*.” -- your textbook.
- Accessing multiple implementations of an interface through an interface reference variable is the one of the most powerful ways that Java achieves run-time polymorphism.

Differences between Abstract Class and Interface

Major difference between an interface and any class:

a class can maintain state information (i.e., variables), but an interface cannot (only static and final, i.e., constants are allowed).

Other differences: Self-study.

End of Lecture 11.

Reading material: up to Chapter 9 (except package) of the textbook