

A large, two-story, light-colored building with a red-tiled roof and a central tower, surrounded by green grass and trees under a clear blue sky.

# MAHARISHI UNIVERSITY of MANAGEMENT

*Engaging the Managing Intelligence of Nature*

## Computer Science Department

**CS390 Fundamental Programming  
Practices (FPP)  
Professor Paul Corazza**



© 2016 Maharishi University of Management, Fairfield, Iowa

All rights reserved. No part of this slide presentation may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying or recording, or by any information storage and retrieval system, without permission in writing from Maharishi University of Management.

# Lecture 7: Inheritance, Interfaces, and Polymorphism

*Need a SCI subtitle*

# Wholeness of the Lesson

Java supports inheritance between classes in support of the OO concepts of inherited types and polymorphism. Interfaces support encapsulation, play a role similar to abstract classes, and provide a safe alternative to multiple inheritance. Likewise, relationships of any kind that are grounded on the deeper values at the source of the individuals involved result in fuller creativity of expression with fewer mistakes.

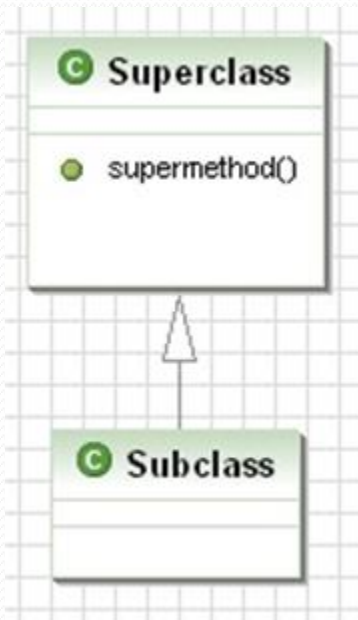
# Outline of Topics

- Introduction to Inheritance – Example of Subclassing a Class
- The "IS-A" and LSP Criteria for Proper Use of Inheritance
- Rules for Subclass Constructors
- Inheritance and the Object Class
- Inheritance for Generalization and Introduction to Polymorphism
- Order of Execution with Inheritance
- Introduction to Java Interfaces, `Comparable`, Functional Interfaces
- New Java 8 Features for Interfaces
- Introduction to the Reflection Library
  - `TheClass` Class
  - `TheConstructor` Class
- The `protected` Keyword and Inheritance Hierarchies
- The Object Class
  - The `toString` Method
  - The `equals` Method
  - The `hashCode` Method
  - The `finalize` Method
  - The `clone` Method: Shallow and Deep Copies



# Introduction to Inheritance

- *Definition.* A class Subclass *inherits from* another class Superclass if objects of type Subclass have automatic access to the "available" methods and variables that have been defined in class Superclass. By "automatic access" we mean that no explicit instantiation of (or reference to) the class Superclass is necessary in order for objects of type Subclass to be able to call methods defined in class Superclass. By "available" methods and variables, we mean methods and variables that have been declared either *public* or *protected* (or have *package level* access if in the same package).



```
class Superclass {
    protected void supermethod() {
        int x = 0;
    }
}

class Subclass extends Superclass {
}

class Main {
    public static void main(String[] args) {
        Subclass sub = new Subclass();
        sub.supermethod();
    }
}
```

A class, method, variable labeled *protected* is accessible to all *subclasses*.

- *Motivation.* In our programming projects, we may find that we define two classes that have many of the same fields and methods. It is natural to think of a single class that generalizes the two classes and that contains the code needed by both.

Secretary

properties:

name

address

phone number

drivesVehicle

salary

behavior:

computeSalary()

Professor

properties:

name

address

phone number

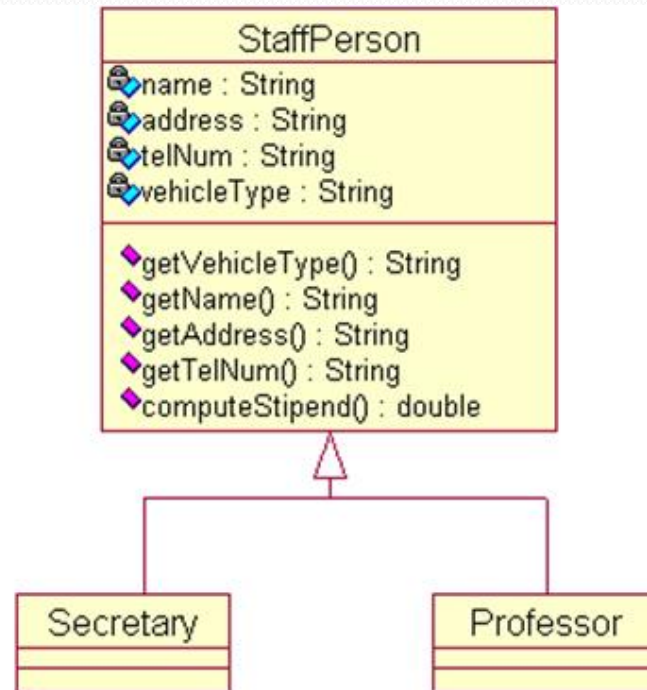
drivesVehicle

salary

behavior:

computeSalary()

Strategy: Create a *generalization* of Secretary and Professor from which both of these classes *inherit*. A StaffPerson class can be defined having all four fields and related methods, and Secretary and Professor can be defined so they are *subclasses* of StaffPerson.





When the classes have this relationship, we may view the type of an instance of a subclass as being that of the superclass. For example, we can instantiate like this:

```
StaffPerson person1 = new Professor();  
StaffPerson person2 = new Secretary();
```

This is similar in spirit to the automatic conversions that are done for primitive types:

```
byte b = 8;  
int k = b;
```

# An Example of Superclass and Subclass:

## Manager subclass of Employee

```
//Employee class, as defined in previous lessons
class Employee {
    Employee(String aName, double aSalary, int aYear, int aMonth, int aDay) {
        name = aName;
        salary = aSalary;
        hireDay = LocalDate.of(aYear,aMonth,aDay);
    }

    // instance methods
    public String getName() {
        return name;
    }
    public double getSalary() {
        return salary;
    }
    public LocalDate getHireDay() {
        return hireDay;
    }
    public void raiseSalary(double byPercent) {
        double raise = salary * byPercent / 100;
        salary += raise;
    }

    //instance fields
    private String name;
    private double salary;
    private LocalDate hireDay;
}
```

```
class Manager extends Employee {
    public Manager(String name, double salary, int year,
        int month, int day) {
        super(name, salary, year, month, day);
        bonus = 0;
    }
    @Override
    public double getSalary() {
        //no direct access to private variables of
        //superclass
        double baseSalary = super.getSalary();
        return baseSalary + bonus;
    }
    public void setBonus(double b) {
        bonus = b;
    }
    private double bonus;
}
```

```
class ManagerTest {
    public static void main(String[] args) {
        Manager boss = new Manager("Boss Guy", 80000,
            1987, 12, 15);
        boss.setBonus(5000);

        Employee[] staff = new Employee[3];

        staff[0] = boss;
        staff[1] = new Employee("Jimbo", 50000, 1989, 10, 1);
        staff[2] = new Employee("Tommy", 40000, 1990, 3, 15);

        //print names and salaries
        for(Employee e : staff) {
            System.out.println("name: " + e.getName() +
                               "salary: " + e.getSalary());
        }
    }
}
```

## Points to observe:

- Manager provides all the "services" of Employee, with additional functionality (involving bonuses) and overriding functionality (getSalary method) – so it's a good candidate for *extending* Employee.
- We use the *extends* keyword to indicate that Manager is a *subclass* of Employee
- A Manager instance can freely use the getName and getHireDay methods of its superclass Employee – no need to re-code these methods. However, special methods that are unique to Manager (in particular, the setBonus method) cannot be called on an Employee instance.

```
Manager m = new Manager(. . .);  
m.getName();           //ok  
m.setBonus(5000);      //ok
```

```
Employee e = new Employee(. . .);  
e.getName();           //ok  
e.setBonus(5000);      //compiler error
```

- We *override* the `getSalary` method in the `Manager` class.
  - This means that the method is defined differently from its original version in `Employee`. A `Manager` object computes salary differently from `Employee` objects.
- Still wish to *use* `getSalary` in `Employee` , but add the value of `bonus` to it. How can this be done?
  - In general, how to access the *superclass version* of a method from within a *subclass*?

*Solution:* Use **super** to indicate that you are accessing the superclass version.

**Best Practice.** Use the `@Override` annotation on `getSalary`.  
*Two reasons :*

- It is possible for another user of your code not to realize that your method overrides a method in a superclass.
- Provides a compiler check that your method *really is* overriding a superclass method.

*Example:* Overriding the `equals` method (see below)



- In the `Manager` constructor, we wish to *reuse* the constructor that is found in `Employee`, but we also want to include more code. This is accomplished by using the **super** keyword again (but it has a different meaning here).

Like **this** in constructor, the use of **super** must occur on the first line of the constructor body.

- *Polymorphic types.* The 0th element of the `staff` array was defined to be of type `Manager`, yet we placed it in an array of `Employee` objects. The fact that an object variable can refer to an object of a given type as well as objects that belong to subtypes of the given type is called *polymorphism*.
- *Dynamic binding.* When the `getSalary` method is called on `staff[0]`, the version of `getSalary` that is used is the version that is found *in the* `Manager` *class*. This is possible because the JVM keeps track of the actual type of the object when it was created (that type is set with execution of the "new" operator). The correct method body (the version that is in `Manager`) is associated with the `getSalary` method at runtime – this "binding" of method body to method name is called *late binding* or *dynamic binding*.

# Main Point

One class (the *subclass*) inherits from another class (the *superclass*) if all protected and public data and methods in the superclass are automatically accessible to the subclass, even though the subclass may have additional methods and data not found in the superclass. Java supports this notion of inheritance. In Java syntax, a class is declared to be a subclass of another by using the *extends* keyword. Likewise, individual intelligence "inherits from" cosmic intelligence, though each "implementation" is unique.

# Correct Use of Inheritance

Here are two tests to check whether one class should inherit from another.

- `Manager` IS-A `Employee` – it's not just that the two classes have some methods in common, but a manager really is an employee. This helps to verify that inheritance is the right relationship between these classes.
- *Liskov Substitution Principle (LSP)*. Another test is: Can a `Manager` instance be used whenever an `Employee` instance is expected? The answer is yes, since every manager really is an employee, and partakes of all the properties and behavior of an employee, though managers support extra behavior.

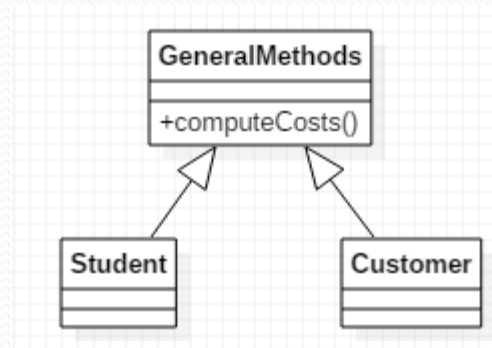
# *Common Mistake: Indiscriminate Generalization*

- The following strategy is a mistake: Place methods common to several classes into one superclass for all of them. Then all the classes have immediate access to methods that they all can use.

This is undesirable because, eventually, some methods and variables in the superclass will not be relevant for some of the subclasses – those subclasses will therefore offer "services" that they cannot possibly provide.

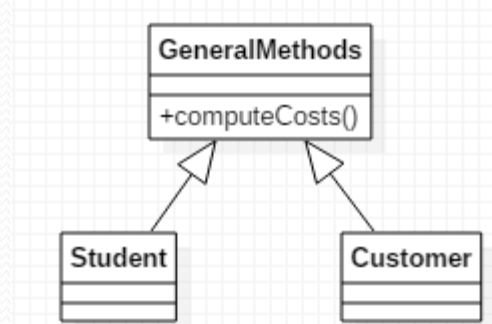
# (continued)

- At first, this may seem reasonable

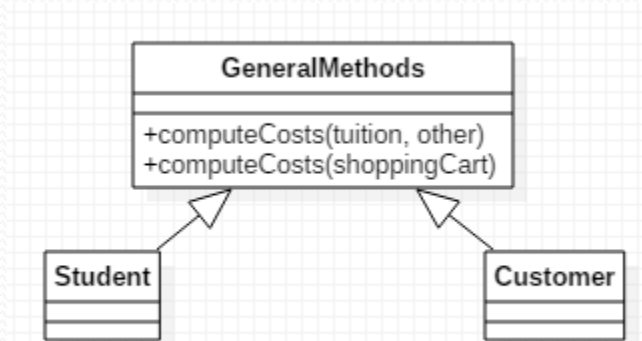


# (continued)

- At first, this may seem reasonable



- As your project evolves, you may find that different versions of `computeCosts` are needed for **Student** and **Customer**



Now a `Customer` seems to be supporting `computeCosts` with input `tuition` and `other`. This undermines the purpose of the `Customer` class



# (continued)

- Often, common methods can be placed in a *utility class*.

```
public class Student {  
    void aMethod() {  
        double tuition = 5000;  
        double other = 3000;  
        //...  
        double val = Util.computeCosts(tuition, other);  
        //...  
    }  
}  
  
public class Customer {  
    void aMethod() {  
        ShoppingCart cart = new ShoppingCart();  
        //...populate cart  
        double val = Util.computeCosts(cart);  
        //...  
    }  
}
```

```
public class Util {  
    private Util() {  
        //private constructor  
    }  
    public static double computeCosts(double tuition, double other) {  
        double cost = 0.0;  
        //...compute  
        return cost;  
    }  
  
    public static double computeCosts(ShoppingCart cart) {  
        double cost = 0.0;  
        //unpack cart and compute  
        return cost;  
    }  
}
```

# Main Point

As a matter of good design, a class C should not be made a subclass of a class D unless C "IS-A" D. Likewise, individual intelligence "is" cosmic intelligence, though this relationship requires time to be recognized as true.

# Rules for Subclass Constructors

## The Rule:

*a subclass constructor must make use of one of the constructors from its superclass*

Reason for the rule The state of the superclass (values of its instance variables) should be set *by the superclass* (not by the subclass). so during construction, the subclass must request the superclass to first set its state, and then the subclass may perform further initialization of its own state.

Example. Employee/Manager:

```
class Employee{
    Employee(String name, double salary, int y, int m, int d){
        //...//
    }
}
class Manager extends Employee {
    Manager(String name, double salary, int y, int m, int d) {
        super(name, salary, y, m, d); //makes use of superclass constructor
    }
}
```

*Note:* It is not necessary for any of the subclass constructors to have the same signature as any of the superclass constructors. However, each of the subclass constructors must access one of the superclass constructors in its implementation.

```
class Employee{
    Employee(String name, double salary, int y,
              int m, int d){
        //...//
    }
}
class Manager extends Employee {
    Manager(String name, double salary,
            double bonus, //different but ok
            int y, int m, int d) {
        super(name, salary, y, m, d);
        this.bonus = bonus;
    }
}
```

The subclass may make use of the implicit default constructor *only if* either

- A. the no-argument constructor of the superclass has been explicitly defined,  
OR
- B. no constructor in the superclass is explicitly defined

In either of these cases, the subclass may make use (possibly implicitly) of the superclass' s default constructor.

//Case A.

```
class Employee{
    Employee(String name, double salary, int y, int m, int d){
        //...//
    }
    //explicit coding of default constructor since another
    //constructor is present
    Employee() {
        //...//
    }
}

class Manager extends Employee {
    //no explicit constructor call here, so the superclass
    //default constructor is used implicitly
}
```



//Case B.

```
class Employee{  
    //...//  
}  
class Manager extends Employee {  
    //...//  
}
```



# Inheritance and the Object Class

- In Java, there is a class called `Object`. Every class created in Java (either in the Java libraries, or user-defined) belongs to the inheritance hierarchy of `Object`.

For example:

```
class MyClass {  
  
}
```

This `MyClass` class automatically inherits from `Object`, even though we do not write syntax that declares this fact.

In later slides, we will discuss the (primarily public) methods that belong to `Object`, and that are therefore inherited by every class in Java.

- Using the `instanceof` operator to check type.

The following code returns true:

```
"Hello" instanceof java.lang.String
```

In general, you can query Java about the type of any runtime object by using `instanceof`. The general syntax is

```
ob instanceof <classname>
```

where `ob` is of type `Object` (or any subtype). This expression will return true if the *runtime type* of `ob` really is of the specified type, or if the class of `ob` is a subclass of (or a subclass of a subclass of...etc) the specified type. Therefore, for example, if `e` is an instance of `Employee` and `s` is a `String`, both of the following are true

```
e instanceof Object  
s instanceof Object
```

Whenever the `instanceof` operator returns true, the object on the left side of the expression can be viewed as having type indicated on the right side (via polymorphic type assignment). So in this example, we could type `e` and `s` above as `Objects`:

```
Object ob_e = e;  
Object ob_s = s;
```

# Two Ways Inheritance Arises

- We saw `Manager` was a natural choice for a *subclass* of `Employee` because it *extends* `Employee`'s behavior.
- Another situation that gives rise to inheritance occurs when several classes are seen to naturally belong to the same general type – this is *generalization*.

In the "figures" example (next slide), it seems natural to generalize the curves `Triangle`, `Circle`, `Square`

```

final class Triangle {
    final double base;
    final double height;
    final double side1, side2, side3;
    Triangle(double base, double height) {
        if (height <= base) {
            this.base = base;
            this.height = height;
        }
        else {
            this.base = height;
            this.height = base;
        }
        side1 = height;
        side2 = base;
        side3 = Math.sqrt(base * base + height * height);
        assert(side1 <= side2 && side2 <= side3);
        assert(height <= base);
    }
    Triangle(double s1, double s2, double s3) {
        double[] arr = {s1, s2, s3};
        Arrays.sort(arr);
        double x = arr[0];
        double y = arr[1];
        double z = arr[2];
        if (x + y < z) {
            throw new IllegalArgumentException("Inputs to Triangle " +
                "are invalid.");
        }
        side1 = x;
        side2 = y;
        side3 = z;
        double u = (y * y - x * x + z * z) / (2 * z);
        double h = Math.sqrt(y * y - u * u);
        base = z;
        height = h;
        assert(side1 <= side2 && side2 <= side3);
        assert(height <= base);
    }
    double computeArea() {
        return (0.5 * base * height);
    }
}

```

```
final class Square {
    final double side;
    Square(double side) {
        this.side = side;
    }
    double computeArea() {
        return (side*side);
    }
}
final class Circle {
    final double radius;
    Circle(double radius) {
        this.radius = radius;
    }
    double computeArea() {
        return (Math.PI * radius * radius);
    }
}
```

//Illustrates a **non-OO** (= **bad**) way of computing areas

```
class Test {
    public static void main(String[] args) {
        Object[] objects = {new Triangle(5,5,5),
                             new Square(3),
                             new Circle(3)};

        //compute areas
        for(Object o : objects) {
            if(o instanceof Triangle) {
                Triangle t = (Triangle)o;
                System.out.println(t.computeArea());
            }
            if(o instanceof Square) {
                Square s = (Square)o;
                System.out.println(s.computeArea());
            }
            if(o instanceof Circle) {
                Circle c = (Circle)o;
                System.out.println(c.computeArea());
            }
        }
    }
}
```



# Points:

- Notice we can arrange `Triangle`, `Square`, `Circle` into an array of type `Object[]` by polymorphism. But `Object` does not have a `computeArea` method, so we cannot polymorphically compute areas by using a single superclass method `computeArea`.
- Instead, if we use an array `Object[]`, we have to repeatedly test the type of the `Object` in the array in order to execute the correct `computeArea` method (using the `instanceof` operator). See package `lesson7.closedcurve.bad`
- This approach needs improvement! (You should never write code that looks like this!)
- ***Aside: Assertions.*** In the `Triangle` class code, assertions are made to check conditions that are required of the instance variables. When the JVM is run with switch `-ea` (enable assertions), failed assertions will cause an `AssertionException` to be thrown. This is a useful style for checking pre- and post-conditions only during development and testing (for production code, the switch `-da` (disable assertions) causes assertion code to be stripped out and not run).

# Towards an OO Solution

- Can **generalize** the behavior of these geometric shape classes to support *polymorphic* access to a general `computeArea` method.

```
abstract class ClosedCurve {  
    abstract double computeArea();  
}
```

The keyword `abstract` on a method means the method is *unimplemented*

```

final class Triangle extends ClosedCurve {
    final double base;
    final double height;
    final double side1, side2, side3;
    Triangle(double base, double height) {
        if (height <= base) {
            this.base = base;
            this.height = height;
        }
        else {
            this.base = height;
            this.height = base;
        }
        side1 = height;
        side2 = base;
        side3 = Math.sqrt(base * base + height * height);
        assert(side1 <= side2 && side2 <= side3);
        assert(height <= base);
    }
    Triangle(double s1, double s2, double s3) {
        double[] arr = {s1, s2, s3};
        Arrays.sort(arr);
        double x = arr[0];
        double y = arr[1];
        double z = arr[2];
        if(x + y < z) {
            //TODO: Throw an exception
        }
        side1 = x;
        side2 = y;
        side3 = z;
        double u = (y * y - x * x + z * z) / (2 * z);
        double h = Math.sqrt(y * y - u * u);
        base = z;
        height = h;
        assert(side1 <= side2 && side2 <= side3);
        assert(height <= base);
    }
    double computeArea() {
        return (0.5 * base * height);
    }
}

```

```
final class Square extends ClosedCurve {
    final double side;
    Square(double side) {
        this.side = side;
    }
    double computeArea() {
        return (side*side);
    }
}
final class Circle extends ClosedCurve {
    final double radius;
    Circle(double radius) {
        this.radius = radius;
    }
    double computeArea() {
        return (Math.PI * radius * radius);
    }
}
```

//This is the **OO** (= **good**) way of computing areas

```
class Test {  
    public static void main(String[] args) {  
        ClosedCurve[] objects = {new Triangle(5,5,5),  
                                   new Square(3),  
                                   new Circle(3)};  
        //compute areas  
        for(ClosedCurve cc : objects) {  
            System.out.println(cc.computeArea());  
        }  
    }  
}
```

# Points

1. No testing of types is required to access the `computeArea` method
2. New types of objects (such as `Rectangle`) can now be introduced by adding new subclasses to `ClosedCurve`. The only change to the code that is needed is inclusion of new instances in the `ClosedCurve[]` array, when it is initialized.

This is an example of the *Open-Closed Principle*: a well-designed OO program is open to extension but closed to modification.

### 3. Points about the code: The *abstract* keyword.

- ❖ If a class is declared *abstract*, it cannot be instantiated.
- ❖ If a method is declared abstract, it cannot have a body -- it can only be declared.
- ❖ If a class has at least one abstract method, the class must be declared abstract.
- ❖ A subclass of an abstract class must implement (provide method bodies for) every abstract method in its superclass (or else declare unimplemented methods abstract).
- ❖ An abstract class is used to declare "services" that it provides. Any subclass of an abstract class promises to make those services available, though different subclasses may accomplish this in different ways (the method `computeArea` is an example of this).
- ❖ Abstract classes may include instance variables and other non-abstract (implemented) methods.

# Inheritance in Swing

In our Swing applications, the top-level GUI class is a subclass of `JFrame`. Recall that for our examples in class we had

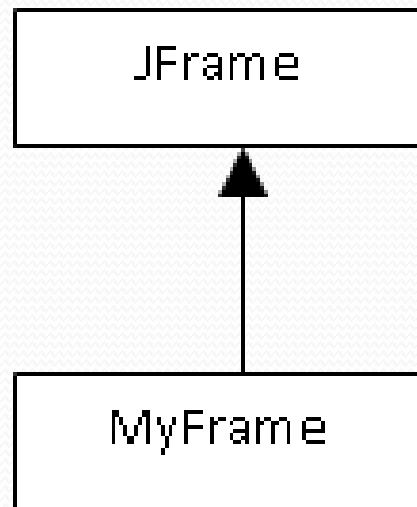
```
public class MyFrame extends JFrame {  
  
    . . .  
  
}
```

and

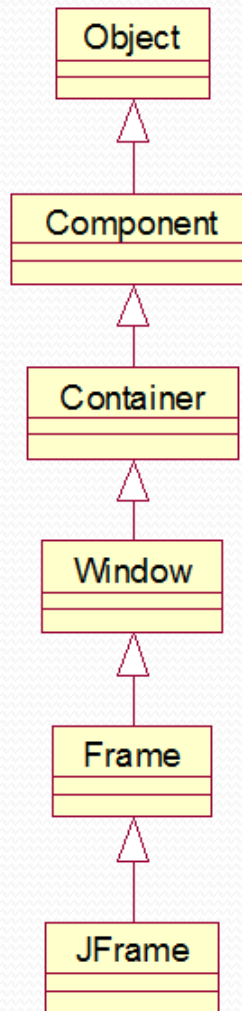
```
public class UserIO extends JFrame {  
  
    . . .  
  
}
```



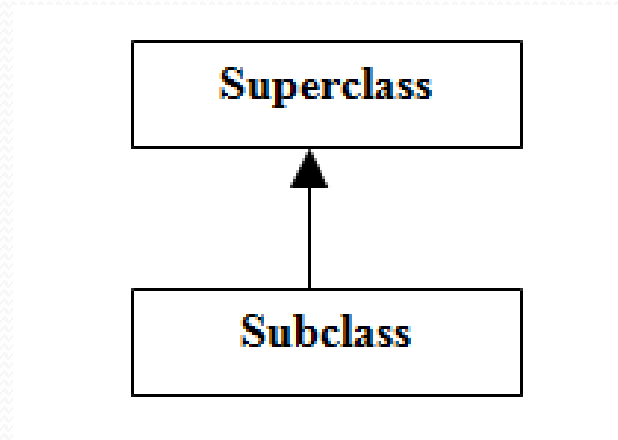
The UML class diagram for this relationship looks like this:



# Inheritance Hierarchy for JFrame



# Order of Execution with Inheritance



Suppose, as in a typical case, we have `Subclass` as a subclass of `Superclass`. When we run

```
new Subclass()
```

the sections of the code are executed according to the following scheme:

- In Superclass, all static variables are initialized and all static initialization blocks are run, in the order in which they appear in the file.
- In Subclass, all static variables are initialized and static initialization blocks are run, in the order in which they appear in the file.
- In Superclass, all instance variables are initialized and all object initialization blocks are run, in the order in which they appear in the file
- In Superclass, the (relevant) constructor is run.
- In Subclass, all instance variables are initialized and all object initialization blocks are run, in the order in which they appear in the file
- In Subclass, the (relevant) constructor is run.

[See Demo – `lesson7.orderofexec`]

# Introduction to Java Interfaces

A Java *interface* is like an abstract class except:

- No instance variables (other than variables declared final) or implemented methods can occur
- An interface is declared using the *interface* keyword, not the *class* keyword
- A class that implements an interface uses the *implements* keyword rather than the *extends* keyword
- Can implement more than one interface. Syntax:

```
MyClass implements Intface1, Intface2, Intface3
```

Can also extend *and* implement. Syntax:

```
MyClass extends SuperClass implements Intface1, Intface2
```

However, no class can have more than one superclass. (Multiple inheritance not supported)

- In the example, ClosedCurve could have been made an interface. The code would have looked like this:

```

public interface ClosedCurve {
    double computeArea();
}

class Triangle implements ClosedCurve {
    double base;
    double height;
    Triangle(double side1,
              double side2, double side3) {
        double[] arr = sort(side1, side2, side3);
        double x = arr[0];
        double y = arr[1];
        double z = arr[2];
        if(x + y < z) {
            System.out.println("Illegal sizes for a triangle:
                               "+side1+", "+side2+", "+side3);
            System.out.println("Using default sizes.");
            computeBaseAndHeight(DEFAULT_SIDE,
                                DEFAULT_SIDE, DEFAULT_SIDE);
        }
        else {
            //method body not shown
            computeBaseAndHeight(x, y, z);
        }
    }
    double computeArea() {
        return (0.5 * base * height);
    }
}

```

```
class Square implements ClosedCurve {
    double side;
    Square(double side) {
        this.side = side;
    }
    double computeArea() {
        return (side*side);
    }
}
class Circle implements ClosedCurve {
    double radius;
    Circle(double radius) {
        this.radius = radius;
    }
    double computeArea() {
        return (Math.PI * radius * radius);
    }
}
```

```
class Test {
    public static void main(String[] args) {
        ClosedCurve[] objects = {new Triangle(5,5),
                                   new Square(3),
                                   new Circle(3)};

        //compute areas
        for(ClosedCurve cc : objects) {
            System.out.println(cc.computeArea());
        }
    }
}
```

**Note!** This example illustrates the fact that a class that implements an interface may be cast as the type of that interface. A simple instance of this, like the example above, would be:

```
ClosedCurve cc = new Rectangle(2,4);
```



# Examples of Interfaces in Swing

1. An important example of interfaces in Swing is the way we must implement listeners in the event-handling model:

```
class MyButtonListener implements ActionListener {  
    public void actionPerformed(ActionEvent evt) {  
        . . . . .  
    }  
}
```

`ActionListener` has only one method in its interface (see Java docs) – `actionPerformed`. So, this is the only method that must be implemented by `MyButtonListener`, and by other typical user-created listener classes.

2. *Anonymous Inner Classes* We cannot directly create an instance of an interface, but we have seen how it is possible to create an instance “on the fly” using anonymous inner classes.

```
button.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent evt) {  
        Sytem.out.println("Button height = " +  
            button.getSize().height);  
        text.setText("button press");  
    }  
});
```

# New Interface Features in Java 8

- Before Java 8, as we have seen, none of the methods in an interface had a method body; all were unimplemented.
- In Java 8, two kinds of implemented methods are now allowed: *default methods* and *static methods*. Both can be added to legacy interfaces without breaking code.
  - A default method is a fully implemented method within an interface, whose declaration begins with the keyword `default`
  - A static method is a fully implemented method within an interface, having the same characteristics as any static method in a class.
- Demo: package `lesson7.java8interface`

# Interfaces in Java 7 and Java 8

**Interview Question:** What is the difference between an abstract class and an interface?

**Answer from the perspective of Java 7 (and before)**

- Abstract classes may have fully implemented methods, but interfaces may not
- Abstract classes may contain static methods while interfaces may not
- Abstract classes may have instance variables of any kind, whereas interfaces can have only public final variables
- All methods in an interface are public, but abstract classes may have implemented methods of any visibility and abstract methods that are either public or protected

## **Answer from the perspective of Java 8 (and after)**

- Abstract classes may have fully implemented methods; interfaces may also have implemented methods, but they must bear the keyword “default”
- Abstract classes may have instance variables of any kind, whereas interfaces can have only public final variables
- All methods in an interface are public, but abstract classes may have implemented methods of any visibility and abstract methods that are either public or protected

# Main Point

Interfaces are used in Java to specify publicly available services in the form of method declarations. A class that implements such an interface must make each of the methods operational. Interfaces may be used polymorphically, in the same way as a superclass in an inheritance hierarchy. Because many interfaces can be implemented by the same class, interfaces provide a safe alternative to multiple inheritance. The concept of an interface is analogous to the creation itself – the creation may be viewed as an “interface” to the undifferentiated field of pure consciousness; each object and avenue of activity in the creation serves as a reminder and embodiment of the ultimate reality.

# Introduction to Java's Reflection Library

- Reflection in Java allows an object to
  - determine information about other objects at runtime (such as attributes, methods, constructors)
  - instantiate another object given just the name of the class (and names or types of the parameters passed to the constructor, if any)
  - call a function based only on the name of the function, the class to which it belongs, and the names or types of the function arguments, if any
- For this course, we will see how Reflection can work in conjunction with polymorphism. We will see how techniques of Reflection give us more flexibility in creating polymorphic code.

# Challenge: Accessing Types without Violating the Open-Closed Principle

In the ClosedCurve example, how can we make it so that not only is each area printed out in the for each loop, but also the *type* of closed curve, as in the following:

```
The area of this Triangle is 12.5  
The area of this Square is 9.0  
The area of this Circle is 28.274
```

We do not want to test the type of each object in the array – this would violate the Open-Closed Principle. How can we output the type of each object in a generic way?

```
public class Test {  
    public static void main(String[] args) {  
        ClosedCurve[] objects = {new Triangle(5,5),  
                                   new Square(3), new Circle(3)};  
  
        //compute areas  
        for(ClosedCurve cc : objects) {  
            System.out.println(cc.computeArea());  
        }  
    }  
}
```



# Solution: Use the *Class* Class

1. The Java runtime keeps track of "runtime type information" about each object. This information includes the class to which it belongs, the name of that class, and many details about the structure of the class.
2. This information is accessible through an object's `Class`. The `Class` to which an object belongs is obtained like this:

```
String s = "Hello";  
Class cl = s.getClass(); //cl represents the String class
```

This same class can be specified using another Java syntax, as follows:

```
Class cl2 = String.class;  
System.out.println(cl == cl2); //true
```

If the name (in the form of a `String`) of a class is known, the `Class` can be created from this `String` as follows:

```
Class cl = Class.forName("java.lang.String");  
//again, (cl == String.class) is true  
//to run properly, this line needs to be enclosed  
//in a try/catch block – this will be covered later
```

3. From the name of the class, the `Class` can be discovered. Conversely, from the `Class` instance, the name of the class can be discovered:

```
// "java.lang.String"
String name = "Hello".getClass().getName();

// also "java.lang.String"
String name2 = "Hello".getClass().getCanonicalName();

// "String"
String name3 = "Hello".getClass().getSimpleName();
```

The difference between these “get name” methods becomes apparent when applied to *inner classes*. If `ClassA` is a nested class in `ClassB`,

```
//returns <package>.ClassB$ClassA
```

```
ClassA.getClass().getName()
```

```
//returns <package>.ClassB.ClassA
```

```
ClassA.getClass().getCanonicalName()
```

```
//returns ClassA
```

```
ClassA.getClass().getSimpleName()
```

#### 4. Another method in Class is newInstance() .

##### Example:

```
Class strClass = String.class;
try {
    //creates an empty String
    String newEmpty = (String)strClass.newInstance();
}
catch(InstantiationException e){
    //handle
}
catch(IllegalAccessException ex){
    //handle
}
```

**NOTE:** For classes whose constructor requires no parameter, this form of `newInstance()` is very convenient. If a constructor does require a parameter, more steps are required. (We discuss these steps later in this lesson.)

Application: These features of the `Class` class allow us to solve the Challenge:

```
public class TestSecond {  
    public static void main(String[] args) {  
        ClosedCurve[] objects = {new Triangle(10,9,6),  
                                   new Square(3),  
                                   new Circle(3)};  
        //compute areas  
        for(ClosedCurve cc : objects) {  
            String nameOfCurve = cc.getClass().getSimpleName();  
            System.out.println("The area of this " +  
                               nameOfCurve + " is " + cc.computeArea());  
        }  
    }  
}
```

# Challenge: Dynamic Construction with Parameters

In modern-day enterprise Java frameworks (like Spring), reflection is used to “wire together” Java classes in the background so that unnecessary dependencies between classes are eliminated.

Spring uses an XML configuration file in which the names of classes are recorded, along with information about the relationships between the classes. This configuration is then used at startup – *using Reflection* – to create instances of the main classes for the application with dependencies realized exactly as intended.

See Demo in the Reflection project

# Reflection Example to Simulate Framework Startup Behavior

See Demo in project ReflectionSample, package reflection

```
package reflection;
```

```
public class TextEditor {  
    ISpellChecker sc;  
  
    public TextEditor(ISpellChecker sc) {  
        this.sc = sc;  
    }  
    public void run() {  
        sc.doSpellCheck();  
    }  
}
```

Question: If we are given the String “reflection.TextEditor” and an instance sc of ISpellChecker, how can we create an instance of TextEditor?

# Example continued: The Constructor Class

Represents the constructors of a class. Example: `MyClass`.

Can be read in various ways from the `Class` class obtained from an object:  
Example: `myClass` is an instance of `MyClass`

```
Class cl = myClass.getClass();  
//the types of the parameters of constructor for cl  
Class[] paramTypes = ...  
Constructor c = cl.getConstructor(paramTypes)
```

Once you have a `Constructor` object for a class, you can create an instance using `newInstance(Object[] params)`

```
//instances of objects to be used as argument for  
//the constructor  
Object[] params = . . .  
MyClass instance = (MyClass)c.newInstance(params);
```

# Continued: Instantiating TextEditor Using Reflection

```
private static TextEditor constructTextEditor(ISpellChecker spCheck)
    throws ReflectionFailedException {
    TextEditor te = null;
    try {
        Class teClass = Class.forName("reflection.TextEditor");
        Class[] params = {ISpellChecker.class};
        Object [] parms = {spCheck};
        Constructor c = teClass.getConstructor(params);
        te = (TextEditor)c.newInstance(parms);

    } catch(Exception e) {
        throw new ReflectionFailedException(e.getMessage());
    }
    return te;
}
```



# Main Point

The classes in the Java Reflection package can be used to construct an instance of a class (with parameters) from a String (which stores the name of the class) or Class object; similarly, it is possible to invoke a method on another class (with parameters) simply by knowing the String name of the method. Likewise, reflection on the infinite creative power of consciousness reveals the truth of every thing and gives rise to the creation of any object.

# More on Inheritance

## 1. The `protected` keyword.

- A subclass has access to all methods and variables that have access specifier `public` or `protected`. (If the subclass happens to be in the same package as the superclass, then it also has access to all methods and variables having package level access as well.)
- Methods and variables in a class that are qualified with the `protected` access specifier are accessible to *all subclasses* of the class and all *classes in the same package* as the class. Therefore, the access specifiers in Java, listed from most to least restrictive, are:

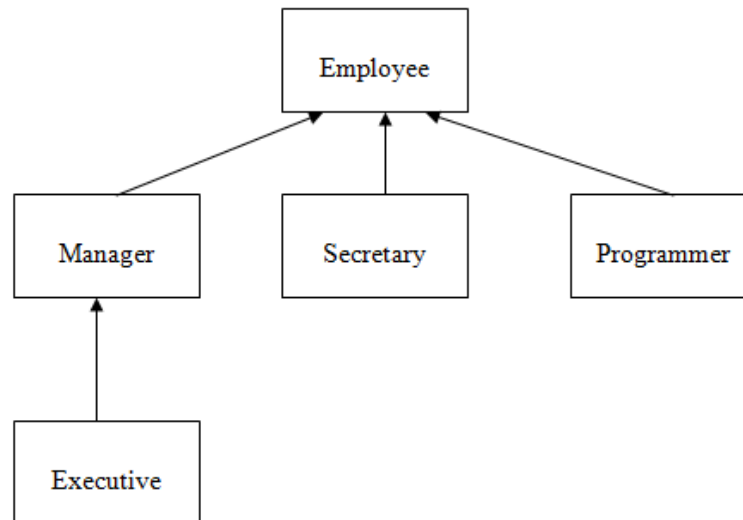
`private`, `<package level>`, `protected`, `public`

- Some people believe (like the author of the book) that a variable should *never* be given `protected` level access because it would violate encapsulation. No general agreement on this point.
- Warning! The rules governing the use of the `protected` keyword are somewhat more involved than this – these will be discussed at the end of the lesson

## 2. *Preventing/restricting inheritance.*

- Declaring a class to be final prevents creation of any subclass (recall: this is one of the techniques used to ensure that a class is *immutable*)
- Declaring a method to be final prevents any subclass from overriding it

## 3. *Inheritance hierarchies.* Sometimes a subclass of a class is further extended by another subclass. Repeated subclassing results in an *inheritance hierarchy*.



# Main Point

When a method is called on a subclass, the JVM by default uses *dynamic binding* to determine the correct method body to execute. Early binding (and hence a slight improvement in performance) can be forced by declaring a method final. In a similar way, it is said (Maharishi, *Science of Being*) that an enlightened individual need not continually plan and prepare in order to meet the needs of his daily life – instead, the enlightened enjoys spontaneous support of nature, and sees what to do as situations arise. Such individuals are an analogue to "late binding".

# The Object Class

- *Singly-rooted.* Every Java class belongs to one large inheritance hierarchy in which Object is at the top. No explicit mention of "extending" Object needs to be made in your code – it is already understood by the compiler and JVM.
- Every class has access to the following methods (and others that we will not cover here):
  - `public String toString()`
  - `public boolean equals(Object o)`
  - `public int hashCode()`
  - `protected Object clone() throws CloneNotSupportedException`
  - `protected void finalize() throws Throwable`

# The toString Method

1. If a class does not override the default implementation of `toString` given in the `Object` class, it produces output like the following:

```
public static void main(String[] args) {  
    System.out.println(new Object());  
    System.out.println(new StoreDirectory(null));  
  
}
```

```
//output  
java.lang.Object@18d107f  
scope.more.StoreDirectory@ad3ba4
```

This is a concatenation of the fully qualified class name with the hexadecimal version of the "hash code" of the object (we will discuss hash codes later in this set of slides)

2. Most Java API classes override this default implementation of `toString`. The purpose of the method is to provide a (readable) `String` representation (which can be logged or printed to the console) of the state of an object.

### Example from the Exercises:

```
// the Account object has this implementation of
// toString
public String toString() {
    String newline =
        System.getProperty("line.separator");
    String ret =
        "Account type: " + acctType + newline +
        "Current bal:  " + balance;
    return ret;
}
```

**Best Practice.** For every significant class you create, override the `toString` method.

3. `toString` is automatically called when you pass an object to `System.out.println` or include it in the formation of a `String`

#### 4. Examples:

```
Account acct = . . . //populate an
AccountString output = "The account: " + acct;
```

---

```
Account acct = . . . // populate an Account
System.out.println(acct);
```



## 5. toString for arrays – sample usage

Suppose we have the array

```
String[] people = {"Bob", "Harry", "Sally"};
```

- Wrong way to form a string from an array

```
people.toString()
```

```
//output: [Ljava.lang.String;@19e0bfd
```

- Right way to form a string from an array

```
Arrays.toString(people)
```

```
//output: [Bob, Harry, Sally]
```

# The equals Method

- Implementation in Object class:

`ob1.equals(ob2)` if and only if `ob1 == ob2`  
if and only if references point to the same object

Using the '`==`' operator to compare objects is usually not what we intend (though for comparison of *primitives*, it is just what is needed). For comparing objects, the `equals` method should (usually) be overridden to compare the *states* of the objects.

Example:

```
class Person {  
    private String name;  
    Person(String n) {  
        name = n;  
    }  
}
```

Two `Person` instances should be "equal" if they have the same name. Good way to override `equals`:

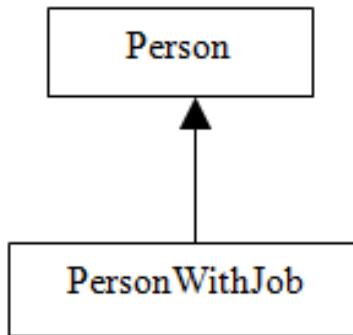
```
//an overriding equals method in the Person class
@Override
public boolean equals(Object aPerson) {
    if(aPerson == null) return false;
    if(!aPerson instanceof Person) return false;
    Person p = (Person)aPerson;
    boolean isEqual = this.name.equals(p.name)
    return isEqual;
}
```

### Things to notice:

- The argument to `equals` must be of type `Object` (otherwise, compiler error)
- If input `aPerson` is `null`, it can't possibly be equal to the current instance of `Person`, so `false` is returned immediately
- If runtime type of `aPerson` is not `Person` (or a subclass), there is no chance of equality, so `false` is returned immediately
- After the preliminary special cases are handled, two `Person` objects are declared to be equal if and only if they have the same name.

# Handling equals() in Inherited Classes

Example: Add a subclass `PersonWithJob` to `Person`:



```
class Person {
    private String name;
    Person(String n) {
        name = n;
    }
    public String getName() {
        return name;
    }
    @Override
    public boolean equals(Object aPerson) {
        if(aPerson == null) return false;
        if(!aPerson instanceof Person) return false;
        Person p = (Person)aPerson;
        boolean isEqual = this.name.equals(p.name)
        return isEqual;
    }
}
class PersonWithJob extends Person {
    private double salary;
    PersonWithJob(String n, double s) {
        super(n);
        salary = s;
    }
}
```

**The equals () method is inherited by PersonWithJob in this implementation. So objects of type PersonWithJob are compared only on the basis of the name field.**

## Example:

```
PersonWithJob joe1 = new PersonWithJob("Joe", 100000);  
PersonWithJob joe2 = new PersonWithJob("Joe", 50000);  
boolean areTheyEqual = joe1.equals(joe2);    //areTheyEqual  
                                              == true
```

***Best Practices:*** If, in your code, this kind of situation does not present a problem – if it is OK to inherit `equals()` in this way – then the implementation given here is optimal. This is called the instance of strategy for overriding equals

Best practice in the case where subclasses need to have their own form of `equals` is more complicated (discussed below)

# What Happens When Subclasses Need Their Own Form of equals()

**Example.** Provide PersonWithJob its own equals method.

```
//an overriding equals method in the PersonWithJob class
@Override
public boolean equals(Object withJob) {
    if(withJob == null) return false;
    if(!withJob instanceof PersonWithJob)
        return false;
    PersonWithJob p = (PersonWithJob)withJob;
    boolean isEqual= getName().equals(p.getName()) &&
        this.salary == p.salary;
    return isEqual;
}
```

This creates a serious problem, called *asymmetry*  
(violates contract for equality)

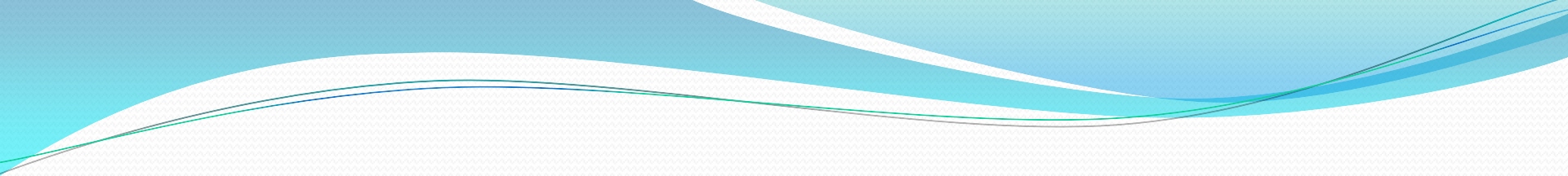
```
Person p = new Person("Joe");  
PersonWithJob withJob = new PersonWithJob("Joe",  
                                             100000);  
  
//true - using Person's equals()  
theyreEqual1 = p.equals(withJob);  
  
//false - using PersonWithJob's equals()  
theyreEqual2 = withJob.equals(p);
```



**Example.** Implement equals in a different way for both Person and PersonWithJob.

```
//alternative equals method in the Person class
@Override
public boolean equals(Object aPerson) {
    if(aPerson == null) return false;
    if(aPerson.getClass() != this.getClass())
        return false;
    Person p = (Person)aPerson;
    boolean isEqual = this.name.equals(p.name);
    return isEqual;
}
```

```
//alternative equals method in the PersonWithJob class
@Override
public boolean equals(Object withJob) {
    if(withJob == null) return false;
    if(withJob.getClass() != this.getClass())
        return false;
    PersonWithJob p = (PersonWithJob)withJob;
    boolean isEqual =
        getName().equals(p.getName()) &&
        this.salary == p.salary;
    return isEqual;
}
```



This solves the asymmetry problem – now, it is impossible for a `PersonWithJob` object to be equal to a `Person` object, using either of the `equals()` methods. This is called the same classes strategy for overriding equals.

The same classes strategy is acceptable when subclass B and superclass A need separate equals methods, *as long as there will never be a subclass of B that needs to use the same equals method as the one used in B.*

# Difficulty with the Same Classes

## Strategy

**Example:** Continuing the example from above, suppose we have a subclass `PersonWithJobInNeighborhood` of `PersonWithJob`.

```
class PersonWithJob extends Person {  
    private double salary;  
    PersonWithJob(String n, double s) {  
        super(n);  
        salary = s;  
    }  
    @Override  
    public boolean equals(Object withJob) {  
        if(withJob == null) return false;  
        if(withJob.getClass() != this.getClass())  
            return false;  
        PersonWithJob p = (PersonWithJob)withJob;  
        boolean isEqual =  
            getName().equals(p.getName()) &&  
            this.salary == p.salary;  
        return isEqual;  
    }  
}
```

```
class PersonWithJobInNeighborhood
    extends PersonWithJob {
        private double salary;
        private boolean isInMyNeighborhood;
        PersonWithJob(String n, double s,
                        boolean isIn){
            super(n, s);
            isInMyNeighborhood = isIn;
        }
    }
```

**The intention here is that PersonWithJobInNeighborhood will use the equals method of its superclass. But this creates a problem:**

```
PersonWithJob joe1 =
    new PersonWithJob("Joe", 50000);
PersonWithJobInNeighborhood joe2 = new
PersonWithJobInNeighborhood("Joe",
                            50000, true);
joe2.equals(joe1); //value is false since joe2
                  //is not of same type as joe1
```

# Best Practice When Using Same Classes Strategy

The example shows that whenever the same classes strategy is used to provide separate `equals` methods for classes B and A, where B is a subclass of A, then we should prevent the possibility of creating a subclass of B to prevent the introduction of a corrupted `equals` method.

***Best Practice – Same Classes Strategy.*** If B is a subclass of A and each class has its own `equals` method, implemented using the same classes strategy, then the class B should be declared `final` to prevent the introduction of an asymmetric definition of `equals` in any future subclass of B.

**Question.** What if we don't wish to make B `final`?

# Using Composition Instead of Inheritance

Even when a potential subclass satisfies IS-A criterion for inheritance, we might not choose to use inheritance, as long as we do not need the subclass for polymorphism.

The discussion above is one such case: Whenever classes B and A, where B is a subclass of A, require different `equals` methods, using composition instead of inheritance is a good strategy, and if making the class B `final` is not an option, it is the only safe way to handle `equals`.

**Example:** *Implementing Manager using Composition instead of Inheritance* (See sample code in package `lesson7.empmanager.usecomposition`)

# Summary of Best Practices for Overriding Equals In the Context of Inheritance

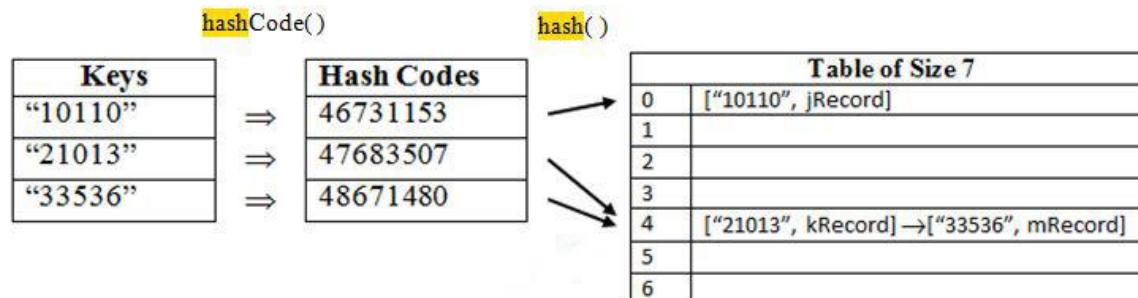
Suppose B is a subclass of A.

- If it is acceptable for B to use the same equals method as used in A, then the best strategy is the *instanceof strategy*
- If *two different equals methods* are required, two strategies are possible
  - Use composition instead of inheritance – this will always work as long as the inheritance relationship between B and A is not needed (e.g. for polymorphism)
  - Use the *same classes* strategy, but declare subclass B to be final

# Overriding hashCode()

When objects of any kind (Integers, Strings, chars, or any others) are used as keys in a hashtable (discussed in Lesson 11), Java will use the `hashCode` method available in the class to transform each key into a small integer, serving as an index in an underlying array.

User's View of Hashtable	
Key (= Employee ID)	Value (= Record)
"10110"	jRecord
"21013"	kRecord
"33536"	mRecord





# (continued)

- For this mechanism to function correctly, the following rule must be followed:

## **HashCode Rule**

*Whenever equals is overridden in a class,  
hashCode must also be overridden*

- The reason for this rule, and the details about how to override hashCode will be discussed in Lesson 11.

# The finalize Method

- The `finalize` method can be overridden in any class. The *intended* behavior of this method in any object is to perform “final” cleanup operations before this object is disposed of by the garbage collector (the garbage collector from time to time disposes of objects to which there are no further references from live objects).
- This method *is not reliable and should not be used*. As the api docs explain, there is no guarantee about which thread will actually call the `finalize` method. In practice, a consequence is that very often `finalize` is not called at all before objects are disposed of by the garbage collector. This means that it is dangerous to rely on the `finalize` method to do cleanup before garbage collection, since any critical cleanup tasks may not be performed at all.
- `finalize` is *costly*. Joshua Bloch (*Effective Java*, 2<sup>nd</sup> ed.) makes the following point about `finalize`:

There is a severe performance penalty for using finalizers. On my machine, the time to create and destroy a simple object is about 5.6 ns. Adding a finalizer increases the time to 2,400 ns. In other words, it is about 430 times slower to create and destroy objects with finalizers.

Other trials with `finalize` support the conclusion that its use is expensive, though not everyone agrees with Bloch about the severity of the performance drag.

# The clone() Method

- The following is a method of Object:

```
protected Object clone() throws  
CloneNotSupportedException
```

The `CloneNotSupportedException` is thrown when an attempt is made by an object of type A to perform a cloning operation but A does not implement the `Cloneable` interface.

- Creators of Java made this method `protected` and enforced the Cloneability requirement to give developers some control over how it is used. For instance, if a class is declared to be `final`, even if it implements `Cloneable`, it is impossible for a rogue programmer to make a clone of the class and then gain access to data by creating subclasses of the copy. (This fact follows from the Gosling rules, to be described later.)
- The rules governing the use of `clone` as a `protected` member of `Object` are based on the somewhat complex rules that govern the `protected` keyword generally.

# Access of protected Members from Within a Subclass

The rules governing protected allow a subclass to directly access protected members of its superclasses. Here is an example where the superclass is `Object` and `MyClass` is any other class (the subclass). This behavior matches the common understanding of the rules for protected members.

```
public class MyClass implements Cloneable {
    String name = "harry";

    public static void main(String[] args) {
        MyClass m = new MyClass();
        try {
            MyClass mcopy = (MyClass)m.clone();
            MyClass mcopy2 = m.getMyClass();
        } catch (CloneNotSupportedException e) {
            //handle
        }
    }

    public MyClass getMyClass()
        throws CloneNotSupportedException {
        return (MyClass) this.clone();
    }
}
```

# Attempting to Access protected Members from the Outside

The following produces a compiler error. The `CallingClass` is attempting to access the protected member of `Object` by using the object reference `cl` of type `MyClass`. Compiler error states that “`clone()` is not visible.” Note that `CallingClass` and `MyClass` are both subclasses of `Object`.

```
//MyClass plays the role of "object reference class"
public class MyClass implements Cloneable {
    String name = "harry";
}

public class CallingClass {
    public MyClass tryToClone(MyClass cl) {
        cl.clone();    //compiler error: clone() not visible
    }
}
```

## Protected Member Accessible If Object Ref Class Is a Subclass of Calling Class

This is the only way Java allows a calling class to access protected members from the outside.

NOTE: If the Object Ref Class is made to be `final`, it is inaccessible by subclassing and also by cloning – a rogue CallingClass cannot force the Object Ref Class to clone itself because of the strict rules for `protected`.

```

public class CallingClass {
    public MyClass tryToClone(MyClass cl)
        throws CloneNotSupportedException {

        //ok because MyClass extends CallingClass
        return (MyClass) cl.clone();
    }

    public static void main(String[] args) {
        CallingClass cc = new CallingClass();
        MyClass cl = new MyClass();
        try {
            //This works
            MyClass result = cc.tryToClone(cl);
        }
        catch (CloneNotSupportedException e) {
        }
    }
}

public class MyClass extends CallingClass implements Cloneable {
    String name = "harry";
}

```

# Usual Approach to Gain Access from Outside: Override

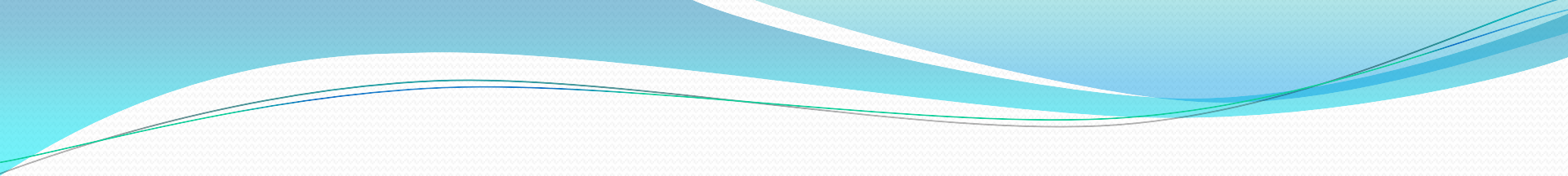
```
public class CallingClass {
    public MyClass tryToClone(MyClass cl) {
        try {
            //ok since clone() is now a public method in MyClass
            return (MyClass) cl.clone();
        } catch(CloneNotSupportedException e) {
            return null;
        }
    }

    public static void main(String[] args) {
        CallingClass cc = new CallingClass();
        MyClass cl = new MyClass();
        MyClass result = cc.tryToClone(cl);
    }
}

public class MyClass implements Cloneable {
    String name = "harry";

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```





NOTE: The implementation of `clone` in `MyClass` uses the default implementation of the `Object` version of `clone`. The code works, but this way of using `clone` is usually not what is wanted because it only produces a *shallow copy*.

Therefore, we describe *how to use* `clone` in the next slides.

# The Problem with Shallow Copies

The shallow copy produced by `Object.clone()` works fine for copying primitives, but only gives a copy of *references* to objects; this means that in the copy, the object references point to the same objects as in the original class.

```

public class Job implements Cloneable {
    int numhours;
    String typeOfJob;
    public Job(int n, String t) {
        numhours = n;
        typeOfJob = t;
    }
    @Override
    public Object clone() throws CloneNotSupportedException {
        //shallow copy is fine here - variables are primitive
        //or immutable
        return (Job) super.clone();
    }
    public String toString() {
        return typeOfJob + ": " + numhours;
    }
}

public class Person implements Cloneable {
    String name;
    Job job;
    public Person(String name, Job j) {
        this.name = name;
        job = j;
    }
    public String toString() {
        return "name: " + name + ", job: [" + job + "]";
    }
    @Override
    public Object clone() throws CloneNotSupportedException {
        //shallow copy not fine here: Job in copy is same as
        //Job in original
        return (Person) super.clone();
    }
}

```

```
public class Main {  
    public static void main(String[] args) {  
        Job joesjob = new Job(40, "Carpenter");  
        Person joe = new Person("Joe", joesjob);  
        System.out.println(joe);  
        try {  
            Person joecopy = (Person)joe.clone();  
            System.out.println(joecopy);  
            joecopy.job.typeOfJob = "Painter";  
            //modifies original object!  
            System.out.println(joe);  
        } catch(CloneNotSupportedException e) { }  
    }  
}
```

# Producing Deep Copies

A *deep copy* is produced by separately cloning all the object instance variables in the class to be cloned and inserting them into the clone.

```

public class Job implements Cloneable {
    int numhours;
    String typeOfJob;
    public Job(int n, String t) {
        numhours = n;
        typeOfJob = t;
    }
    @Override
    public Object clone() throws CloneNotSupportedException {
        //shallow copy is fine here - variables are primitive
        //or immutable
        return (Job) super.clone();
    }
    public String toString() {
        return typeOfJob + ": " + numhours;
    }
}

public class Person implements Cloneable {
    String name;
    Job job;
    public Person(String name, Job j) {
        this.name = name;
        job = j;
    }
    public String toString() {
        return "name: " + name + ", job: [" + job + "]";
    }
    @Override
    public Object clone() throws CloneNotSupportedException {
        //creates a deep copy
        Person pcopy = (Person) super.clone();
        pcopy.job = (Job) job.clone();
        return pcopy;
    }
}

```

```
public class Main {  
    public static void main(String[] args) {  
        Job joesjob = new Job(40, "Carpenter");  
        Person joe = new Person("Joe", joesjob);  
        System.out.println(joe);  
        try {  
            Person joecopy =  
                (Person)joe.clone();  
            System.out.println(joecopy);  
            joecopy.job.typeOfJob = "Painter";  
            //does not modify orig object!  
            System.out.println(joe);  
        } catch(CloneNotSupportedException e) {  
        }  
    }  
}
```

# Optional: The Gosling Rules

Whenever we have 3 Java classes, SuperClass, CallingClass, ObjectRefClass so that

- SuperClass and CallingClass belong to different packages
- SuperClass has a protected member proMem
- CallingClass and ObjectRefClass are both subclasses of SuperClass
- A method inside CallingClass attempts to access proMem by way of an object reference to ObjectRefClass – for instance

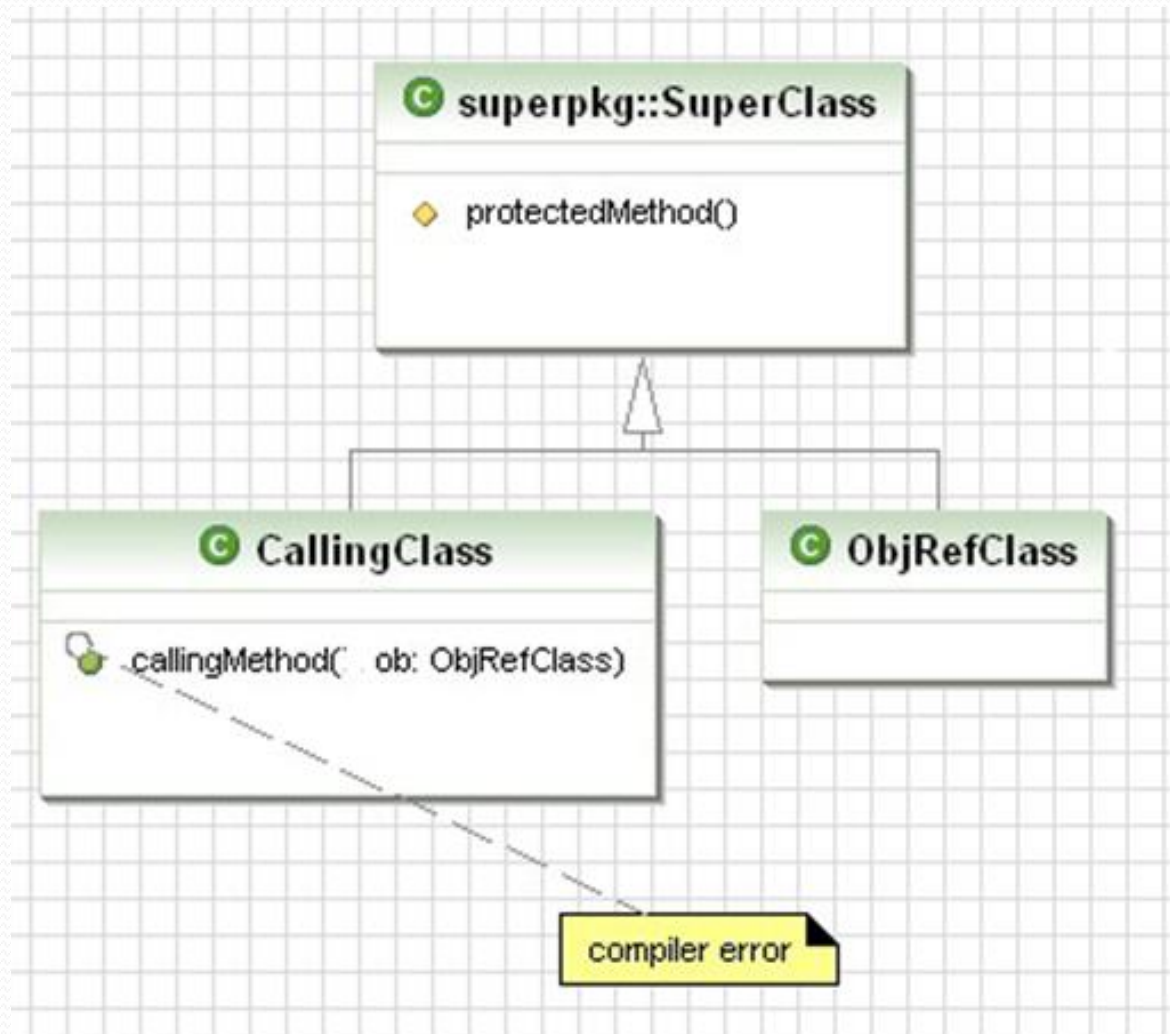
```
void aMethod(ObjectRefClass ref) {
    ref.proMem;
}
```
- AND, ObjectRefClass is NOT a subclass of CallingClass

THEN

There is a compiler error – proMem is not considered to be visible to ref.

**Example:** For our clone() example, the SuperClass is Object (and the protected member proMem is Object's clone() method), the CallingClass is CallingClass, and the ObjRefClass is MyDataClass.





# The Gosling Rules

[From *Gosling, The Java Programming Language*]

- A. A `protected` member of a class is always accessible within that class
- B. A `protected` member of a class `SuperClass` is always accessible to any method of a calling class that lives in the same package as `SuperClass` (even if the method sends a message to a subclass `ObjRefClass` (of `SuperClass`) object belonging to a different package).
- C. A `protected` member of a class `SuperClass` can also be accessed from a another class `CallingClass` in a different package, through an object reference (an instance of `ObjRefClass`) that is a narrower type than, or of the same type as, `CallingClass` – i.e. another instance of `CallingClass` or an instance of a subclass of `CallingClass`. If the type of the object reference fails to be the same as or narrower than `CallingClass`, access to the `protected` member is not allowed.

# Three Ways to Resolve the Protected Paradox (demo)

1. Make `ObjRefClass` a subclass of `CallingClass`
2. Put `CallingClass` in the same package as `SuperClass`
3. Override `SuperClass`'s protected method in `ObjRefClass` (but the new version of the method should now be `public`\*; this overriding method is then called by `CallingClass`). This 3<sup>rd</sup> solution is not related to the Gosling rules – it just uses the fact that public methods of an instance can be accessed by other objects.

\*`public` is the typical way. It can also be protected, but then, in accordance with the Gosling Rules, either `CallingClass` and `ObjRefClass` must belong to the same package, or `ObjRefClass` must be a subclass of `CallingClass`.

Solution 3 is the way `clone()` must be handled. In general, if you are subclassing a third-party class such as `SuperClass` (as we do always in relation to the class `Object`), Solution 3 is our only option. (demo)

# Summary

1. Inheritance provides subclasses with access to data and methods that may be inaccessible to other classes.
2. Inheritance supports polymorphism, which makes it possible to perform operations on many different types by performing those operations on just one supertype.
3. Inheritance must be used wisely; the IS-A and LSP criteria provide guidelines for when subclassing can be used safely.
4. Java interfaces provide even more abstraction of classes, and also support polymorphism.
5. As of Java 8, static and default methods may be included in a Java interface.
6. Java's Reflection library makes it possible at runtime for objects to instantiate classes based only on their name and constructor argument types, and to examine the structure of objects at runtime. These tools can provide a powerful addition to OO programming techniques; they play a fundamental role in the design of modern frameworks, like Spring.
7. The Object class is the single root of the inheritance hierarchy that includes all Java classes. Consequently, every Java class, including user-defined classes, automatically inherits several methods defined in Object: equals, hashCode, toString, and clone. Whenever these methods are needed, their default implementation in Object typically needs to be overridden by in user-defined classes.

# Connecting the Parts of Knowledge With the Wholeness of Knowledge

*Using Reflection to create objects at the level of "name"*

1. Ordinarily, an object of a certain type is created in Java by calling the constructor of the class that realizes this type. This is object construction on the level of *form*.
  2. Java's Reflection API allows the developer to construct an object based on the knowledge of the name (and the number and types of arguments required by the constructor). This is object construction on the level of *name*.
- 
3. **Transcendental Consciousness:** The fundamental impulses that structure both the name and form of an object have their basis in the silent field of pure consciousness.
  4. **Wholeness moving within itself:** In Unity Consciousness, the finest structuring mechanics of creation are appreciated as modes of vibration of the Self.

