MIN 545 - Lecture IV
Inheritance and Polymorphism
A Puzzle...

All **HourlyEmployees** are also **Employees**. Some **Employees** are **Salaried**. **Executives** are always **Salaried**. **HourlyEmployees** are never **Salaried**. **Technical staff** are **Salaried employees** that are not **Executives**. Some **Hourly employees** are **Part-time**. **Technical staff** are either **Engineers**, **Technicians** or **Clerks**. **Full-time hourly employees** are **Hourly Employees** but not **Part-time**.
**Class Inheritance** allows us to:

- Create simple parent objects and to make them more specialized.
- Have specializations share the variables & methods of their common ancestors.
- Write the code once, and re-use in many children! (Laziness)
Derived Class, subclass, child class all mean the same thing.

Base class, superclass, parent class all mean the same thing.

Base Class:

```java
public class Employee
{
    protected String name;
    protected long idNumber;
}
```

Child (derived class):

```java
public class HourlyEmployee extends Employee
{
    private float hourlyRate;
    private int hoursWorked;
}
```

The `HourlyEmployee` objects have `name` and `idNumber` inherited from `Employee`, but they also include new variables: `hourlyRate` and `hoursWorked`. 
In UML, inheritance relationships are shown with an arrow with a closed, white arrowhead.

The arrow is from the child to the parent and is called an “Is-A” relationship.

- e.g. Bird is an Animal

Notice that the variables/methods inherited are not re-shown in the child diagrams.
Structure of a Derived Class

- Every child class is its own type and also of its parent’s type(s).
- Creation of a derived object involves the creation of its parent as well.
- The child holds its parent’s spec within itself.
  - e.g. you have your parents’ DNA.
- Each derived object refers to its parent version using the reference `super`
Visibility in Inheritance

Remember the three classes of visibility, in order of permissiveness:

- **public**: Public variables and methods are accessible by everyone.
- **protected**: Protected variables and methods are only accessible by the class and its derived classes.
- **private**: Private variables and methods are accessible only by the class itself.

- **super** and **this** are always **private**.
- **super** is even more restrictive than private as it cannot be returned in a method or assigned to another reference. It can only be used directly.
Exercise

Assume the following case:

class A {
    private int vA1;
    protected float vA2;
    public String vA3;

    public void fA1() { System.out.println(vA1); }
    protected int fA2(int m) { return m * (int) vA2; }
    private boolean fA3() { return (vA1 < vA2); }
}

class B extends A {
    private int vB1 = 3;
    public float vB2;
}

class C {
    private A vC1;
    private B vC2;
}
Example

Can we add the following method to the definition of class C?

```java
public float fC4() {
    return vC1.vA2;
}
```

NO. Class C is not a child of class A. Therefore it cannot access protected variables.
Exercise

Example

Can we add the following method to the definition of class B?

```java
class B {
    public String fB1() {
        return vA3;
    }
}
```

YES. vA3 is an instance variable of B inherited from parent class A. It is public so everyone can access it.
Example

Can we add the following method to the definition of class B?

```java
public float fB2() {
    return vA2;
}
```

YES. vA2 is an instance variable of B inherited from parent class A. It is protected and B is a child of A, so B can access it.
Exercise

Example

Can we add the following method to the definition of class B?

```java
public int fB3()
{
    return vA1;
}
```

**NO.** vA1 is an instance variable of B inherited from parent class A. But, it is **private** and even if B is a child of A, B cannot access it.
Example

Can we add the following method to the definition of class C?

```java
public void fC4()
{
    vC2.fA1();
}
```

YES. fA1 is a method of B inherited from parent class A. It is `public` so it can be accessed by objects of class C.
Exercise

Example

Can we add the following method to the definition of class C?

```java
public int fC5(int a) {
    return vC2.fA2(a);
}
```

**NO.** fA2 is a **protected** method of B inherited from parent class A. Since C is not a child of A, it cannot access fA2.
Exercise

Example

Can we add the following method to the definition of class B?

```java
public int fB4(int a) {
    return fA2(a);
}
```

YES. fA2 is a protected method of B inherited from parent class A. Since B is a child of A, it can access fA2.
Exercise

Example

Can we add the following method to the definition of class B?

```java
public boolean fB5()
{
    return fA3();
}
```

NO. fA3 is a private method of A. Although B is a child of A, it can not access fA3.
Example

Can we add the following method to the definition of class B?

```java
public B fB6()
{
    return this;
}
```

**YES.** this is a `private` variable of B. B can access and return its own address. However, this is sort of useless... If you can call fB6() on some object, it means you already have its reference!
Exercise

Example

Can we add the following method to the definition of class B?

```java
public A fB6()
{
    return super;
}
```

**NO.** super is a `private` variable of B and normally this would be fine. But super is an exceptional case in that B can use super but can never give it away.
Overriding Methods

- A method is uniquely defined by it’s **name** and **parameter list**.
  - This is called the method’s **signature** *(or fingerprint)*
- A class can define two methods with the same name but different parameters.
  - Remember, this is called **overloading**
- Java will not allow two different methods with the same signature in the same class.
- However a child class can define a new method with the same signature as that of its parent class.
  - Assume class A has:
    ```java
    public void fA1() { System.out.println(vA1); }
    ```
  - Child class B can define:
    ```java
    public void fA1() { System.out.println(vA2*2); }
    ```
  - This is called **overriding**.
Example

```java
public class Person {
    protected String name;
    public void getName() {
        return name;
    }
}

public class Doctor extends Person {
    @Override
    public void getName() {
        return "Dr. " + name;
    }
}

The override annotation (@Override) is optional but recommended

- The compiler will warn you if your override is not OK (misspelled name &c).
Java has a predefined class named **Object**

Every class in Java is a child of the Object class

- Even if a class does not extend another class, it is assumed to extend Object.

Two important methods provided by Object:

- **public String toString()**
  - Gives a string representation of the object.
  - System.out.printXX family of functions use this if given an arbitrary object to print.
  - The default is to a string of the form "<Classname>@<Address>"

- **public boolean equals(Object other)**
  - The default is to check whether they have the same address.

*http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html*
Using super for Overrides

- If a method is overridden by the child class, you can use `super` to access the parent version.

Example

```java
public class Person {
    protected String name;
    public void getName()
    {
        return name;
    }
}

double class Doctor extends Person {
    @Override
    public void getName()
    {
        return "Dr. " + super.getName();
    }
}
```
super in Constructors

- The construction of every object requires construction of the parent object as well.
- This is usually done automatically by Java, if the parent has a default constructor.
- **What if it doesn’t?**
  - The child must define a constructor and call the parent constructor explicitly.

### Example

```java
public class Person {
    protected String name;
    public Person(String name) {
        this.name = name;
    }
}

public class Doctor extends Person {
    public Doctor(String name) {
        super(name);
    }
}
```

Like this, `super` must be the first thing called in a constructor!
Case Study: Aircraft

Aircraft

Helicopter

Airplane

Airport

Airliner

MilitaryPlane

Airbase

B747

F16
public class Aircraft {
    // Every craft has a unique registration ID (e.g. TC-ACA)
    public static final int MAX_ALT = 50000;
    private String regID;
    protected int altitude;

    public Aircraft(String id) { regID = id; }
    public String getRegID() { return regID; }
    public int getAltitude() { return altitude; }

    public void sendRadioMsg(String msg) {
        System.out.printf("%s to Base: %s%n", regID, msg);
    }

    public boolean takeOff(int alt) {
        if (alt > MAX_ALT) {
            sendRadioMsg("Can’t go to " + alt + " m.");
            return false;
        }
        sendRadioMsg("Taking off!");
        altitude = alt;
        return true;
    }
}
public class Helicopter extends Aircraft {

    public Helicopter(String regID) {
        super(regID);
    }

    public boolean takeOff(int alt) {
        System.out.println("Starting rotor...");
        return super.takeOff(alt);
    }

}
Late Binding

- Consider the following method invocation:
  - `obj.doIt();`

- At some point, this invocation is bound to the definition of the method that it invokes:
  - At compile time,
  - At initial run,
  - During runtime.

- If this binding occurred at compile time, then that line of code would call the same method every time

- However, Java defers method binding until run time – this is called **dynamic binding** or **late binding**
Polymorphism

- **Polymorphism**: “having many forms”
- A polymorphic reference is a reference that may refer to different objects.
  - e.g. Every object in Java is of type Object.
  - `Object o` may refer to any object of the class Object.
- The method invoked through a polymorphic reference can change from one invocation to the next.
- All object references in Java are potentially polymorphic.
  - Any object can be referred to as its own type or any of its parent types.
  - Java decides which method to invoke at the last possible step.
Example

```java
Person p1 = new Person("Ali");
Person p2 = new Doctor("Banu");
System.out.println(p1.getName());
System.out.println(p2.getName());
```

Output:

Ali
Dr. Banu

- From Java’s perspective, at first, both p1 and p2 are Person objects.
- When `getName()` needs to be invoked, Java binds the invocation.
  - Realizes at that point that p2 is a Doctor
The `instanceof` operator

- Type membership can be checked using the `instanceof` operator.
- Expression format:
  
  `<object reference> instanceof <class name>`

- Returns `true` if the object belongs to the class, `false` if not.

**Example**

```java
Person p1 = new Person("Ali");
Person p2 = new Doctor("Banu");

p1 instanceof Person; // returns true;
p2 instanceof Person; // returns true;
p1 instanceof Doctor; // returns false;
p2 instanceof Doctor; // returns true;
p1 instanceof Object; // returns true;
p2 instanceof String; // returns false;
```
public class Airport
{
    private int runwayLength;
    private String code;

    public Airport(int rwLen, String code)
    {
        runwayLength = rwLen;
        this.code = code;
    }

    public int getRunwayLength()
    {
        return runwayLength;
    }

    public boolean getPermission(Aircraft a)
    {
        if (a instanceof MilitaryPlane)
            return false;
        return true;
    }
}
public class Airplane extends Aircraft {
    private int minRunway;
    private Airport myAirport;

    public Airplane(String regID, int minRunway) {
        super(regID);
        this.minRunway = minRunway;
    }

    public void setAirport(Airport a) {
        myAirport = a;
    }
}
public boolean takeOff(int alt) {
    if (myAirport == null) {
        sendRadioMsg("No airport!");
        return false;
    }
    sendRadioMsg("Starting engine.");
    sendRadioMsg("Moving to runway.");

    if (!myAirport.getPermission(this)) {
        sendRadioMsg("No permission to take-off!");
        return false;
    }

    if (myAirport.getRunwayLength() < minRunway) {
        sendRadioMsg("Runway too short!");
        return false;
    }

    return super.takeOff(alt);
}
Case Study: MilitaryPlane Class

```java
public class MilitaryPlane extends Airplane {
    public MilitaryPlane (String regID, int rl) {
        super(regID, rl);
    }
}
```
public class Airliner extends Airplane
{
    public Airliner(String id, String Model, int rl)
    {
        super(Model + "-" + id, rl);
    }

    public boolean takeOff(int alt)
    {
        sendRadioMsg("Turning on seatbelt sign.");
        if (super.takeOff(alt))
        {
            sendRadioMsg("Turning off seatbelt sign.");
            return true;
        }
        else
        {
            return false;
        }
    }
}
public class B747 extends Airliner {
    public B747(String id)
    {
        super(id, "B747", 2000);
    }
    public boolean takeOff(int alt)
    {
        return super.takeOff(alt);
    }
}
public class F16 extends MilitaryPlane {
    public F16(String id) {
        super("F16-" + id, 1000);
    }

    public boolean takeOff(int alt) {
        return super.takeOff(alt);
    }
}
Multiple Inheritance

- Sometimes an class will naturally be a subclass of two different classes.

**Example**
The V-22 Osprey, flies like a plane, lands like a helicopter.

- Some programming languages allow this.
  - **Multiple Inheritance**
  - In Java, multiple inheritance is **not allowed**.
    - A class is allowed to extend only one class.
    - Why?
The Problem with Multiple Inheritance

Assume that the following:
- Aircraft class has a method called land(), so all its children must have it.
- Airplane and Helicopter have their own overrides of land()
- Osprey has no implementation of land()

What happens when someone invokes land() on an Osprey object?
The Diamond Problem

- This is called the **Diamond Problem**
  - “The Deadly Diamond of Death”
- It is the major reason why some programming languages eschew multiple inheritance.
- In Java, the diamond problem would be very common.
  - All classes descend from Object.
  - The Object methods (e.g. `toString()`) would frequently cause problems.
Abstract Methods and Classes

- An **abstract method** is a method prototype without the implementation.
  - It is the *promise* of a method.
- They are declared just like normal methods but with the keyword `abstract` and no body. e.g.
  ```java
  public abstract void foo(int x);
  ```
- Classes that declare abstract methods cannot be instantiated.
  - What if somebody actually calls the abstract method?
  - These are called **Abstract Classes**
- The child of an abstract class **must** override the abstract method or also be abstract itself.
- Abstract classes are declared using the `abstract` keyword.
  ```java
  public abstract class Hede { ... }
  ```
Motivating Problem

Let’s say you create the following design:

- **Shape**
  - double width
  - double height
  + Shape(double, double)
  + getWidth() : double
  + getHeight() : double
  + getArea() : double

- **Ellipse**
- **Rectangle**
- **Triangle**

- The problem is **getArea()**
  - How do you calculate the area of an undefined shape?
  - If only we could leave it for later...
Solution: Abstract Parent Class

- Make shape abstract
  - The child classes will have to deal with calculating the area.

```java
public abstract Shape {
    private double width;
    private double height;

    public Shape(double w, double h) {
        width = w; height = h;
    }

    public double getWidth() { return width; }
    public double getHeight() { return height; }

    /* The implementation of getArea
     * is left to child classes */
    public abstract double getArea();
}
```
public class Rectangle {
    public Rectangle (double w, double h) { super(w,h); }
    public double getArea() {
        return getWidth() * getHeight();
    }
}

public class Triangle {
    public Triangle (double w, double h) { super(w,h); }
    public double getArea() {
        return 0.5 * getWidth() * getHeight();
    }
}

public class Ellipse {
    public Ellipse (double w, double h) { super(w,h); }
    public double getArea() {
        return 0.25 * Math.PI * getWidth() * getHeight();
    }
}
Rules of the Game

- You cannot instantiate an abstract class even if it has a constructor and no abstract methods
  - e.g. `Foo f = new Foo()` is not allowed if Foo is abstract.
  - “Foo is abstract; cannot be instantiated”

- Concrete (non-abstract) classes cannot declare abstract methods.
  - “MyClass is not abstract and does not override abstract method foo() in MyClass”

- Concrete classes cannot not override abstract methods of their parents.
  - “MyChild is not abstract and does not override abstract method foo() in MyParent”

- Abstract methods cannot be private
  - “illegal combination of modifiers: abstract and private”
Interfaces are a special form similar to abstract classes that embody a property or ability of objects:

- Cannot be instantiated.
- Cannot have variables.
- Can only have public abstract methods.

They are declared using the interface keyword:

```java
public interface Colorful {
    public Color getColor();
    public void setColor(Color c);
}
```

Notice that you do not need to declare individual methods abstract.

It is assumed since all methods of an interface are abstract.
Use of Interfaces

- Interfaces sometimes embody the *adjectives* in a requirements specification.
  - e.g. "...some kinds types *fabric* will be *colorful*.”
  - Typically interface names end in *-able* or *-er*
    - e.g. ConnectionHandler, Comparable, Iterator

- Classes **implement** the interfaces:

```java
class CottonFabric extends Fabric implements Colorful {
    Color color;
    //...
    public Color getColor() {return color}
    public void setColor(Color c) {color = c;}
}
```

- A concrete class that implements an interface must implement all its methods.
Since interfaces do not have state or concrete methods, they do not cause the diamond problem.

Therefore a class can extend one class and implement one or more interfaces

Example

class Employee extends Person implements Taxpayer, Employable

class Company extends Organization implements Taxpayer, Employer

class Charity extends Organization implements Employer

class Intern extends Person implements Employable

Therefore Java uses them to simulate multiple inheritance

Other languages have similar constructs called Mixins
Abstract Classes and Interfaces in UML

- Abstract classes and interfaces are marked above their names.
- The *implements* relationships are shown with dashed arrows.
Solution to the Osprey Problem

- The problem with the Osprey is that it is a plane in all respects, except it can take off and land like a helicopter.
  - Helicopters can do Vertical Take Off and Landing (VTOL) only
  - Airplanes can do Convential Take-Off and Landing (CTOL) only
  - The Osprey can do both
- So if the Osprey becomes the child of Airplane but also has VTOL, we solve the problem.
- We might encapsulate these into two interfaces:

```java
interface VTOL {
    public void VTakeOff();
    public void VLand();
}

interface CTOL {
    public void TakeOff();
    public void Land();
}
```
Solution to the Osprey Problem (cont.)

- The Osprey now inherits its CTOL behaviour from Airplane.
- It gets the VTOL behaviour by implementing VTOL separately.
Polymorphism With Interfaces

- You can have references to interfaces.
  - Can refer to any object implementing that class.
- Hence, interface references are polymorphic:

Example

```java
VTOL v[] = new VTOL[2];
v[0] = new Helicopter();
v[1] = new Osprey();
CTOL c = (CTOL) v[1];
```

However, `CTOL c = (CTOL) v[0]` will cause a `ClassCastException` since `v[0]` is a `Helicopter` and does not implement `CTOL`.

- When referring to an object through an interface reference, you can only call methods defined by that interface.
Interface Hierarchies

- Interfaces can inherit from other interfaces, just like classes.
- The child interface inherits all the abstract methods of the parent.
- A class implementing the child interface must also implement the parent interface.

Example

```java
public interface Walker {
    public void walk();
}

public interface Runner extends Walker {
    public void run();
}
```

Anything that implements `Runner` must have both the `walk()` and `run()` methods.
## Summary

<table>
<thead>
<tr>
<th>Category</th>
<th>Concrete Class</th>
<th>Abstract Class</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal Methods</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>Abstract Methods</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Variables</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>Statics</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>Constants</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Instantiation</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>