Inheritance & Polymorphism

Object-Oriented Programming
Outline

- Example
- Design an inheritance structure
- IS-A and HAS-A
- Polymorphism
- protected access level
- Rules for overriding
- the Object class

- Readings:
  - HFJ: Ch. 7.
  - GT: Ch. 7.
Inheritance – Example

1. Look at what all four classes have in common.

2. They are Shapes, they all rotate and playSound, so we abstract out the common features and put them into a new class called Shape.

3. Then we link all four classes to the new Shape class in a relationship called inheritance.
Inheritance – Example

We read this as…

- Square inherits from Shape. Circle inherits from Shape. …
- Shape is the superclass of Square, Circle, Triangle, Amoeba
- The other four are subclasses of Shape.

- if Shape has the functionality, then the subclasses automatically get the same functionality
Inheritance – Example

But… Amoeba *rotate* and *playSound* differently!

4. Let Amoeba **override** the inherited rotate() and playSound()
Food for thought

Tiger                             HouseCat

Which one should be subclass/superclass?
Or, should they both be subclasses to some *other* class?
How should you design an inheritance structure?
What should be overridden?
What is inheritance?

- The subclass **inherits** from the superclass, i.e., the subclass inherits members of the superclass:
  - instance variables and methods

- The subclass **specializes** the superclass:
  - it can add new variables and methods.
  - it can override inherited methods.
Example

```
<table>
<thead>
<tr>
<th>Animal</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
<tr>
<td>makeNoise()</td>
</tr>
</tbody>
</table>

Cow
- givesMilk
- makeNoise()

Dog
- chaseCats()

add a new instance variable
override the inherited method
add a new method
```
```java
class Animal {
    String name;
    void makeNoise() {
        System.out.print("Hmm");
    }
}

class Cow extends Animal {
    boolean givesMilk;
    void makeNoise() {
        System.out.print("Moooonoooooo..." recession);
    }
}

class Dog extends Animal {
    void chaseCats() {
        System.out.print("I'm coming, cat!");
    }
}

cow = new Cow();
cow.makeNoise();
cow.givesMilk = true;
dog = new Dog();
dog.makeNoise();
dog.chaseCats();
```
Design an inheritance structure

- A program that simulates a number of animals of different species: tigers, lions, wolves, dogs, hippos, cats….
- We want other programmers to be able to add new kinds of animals to the program at any time.
Design an inheritance structure

Step 1:
Figure out the common abstract characteristics that all animals have.

- instance variables
  - food
  - hunger
  - location

- methods
  - makeNoise()
  - eat()
  - sleep()
  - roam()
Design an inheritance structure

Step 2:
Design a class that represents all common states and behaviors
Step 3:
Decide if a subclass needs any behaviours that are specific to that particular subclass
Design an inheritance structure

**Step 4:**
Look for more inheritance opportunities: **Subclasses that might need common behaviors**

![Inheritance Diagram]

- **Animal**
  - picture
  - food
  - hunger
  - boundaries
  - location
  - makeNoise()
  - eat()
  - sleep()
  - roam()

- **Feline**
  - roam()

- **Lion**
  - makeNoise()
  - eat()

- **Cat**
  - makeNoise()
  - eat()

- **Tiger**
  - makeNoise()
  - eat()

- **Hippo**
  - makeNoise()
  - eat()

- **Canine**
  - roam()

- **Dog**
  - makeNoise()
  - eat()
  - chaseCats()

- **Wolf**
  - makeNoise()
  - eat()
Overriding - Which method is called?

- Which version of the methods get called?

```java
Wolf w = new Wolf();
w.makeNoise();
w.roam();
w.eat();
w.sleep();
```

**the lowest one wins!**
IS-A and HAS-A relationship

- Triangle IS-A Shape
- Cow IS-An Animal
- Dog IS-An Animal
- House HAS-A Kitchen
- Kitchen HAS-A Sink
- Kitchen HAS-A Stove

**Inheritance**

![Inheritance Diagram]

**Composition**

![Composition Diagram]
Code reuse

- Copy & paste
  - Manually -> Error-prone
- Composition – “HAS-A” relationship
  - the new class is composed of objects of existing classes.
  - reuse the functionality of the existing class, not its form
- Inheritance – “IS-A” relationship
  - create a new class as a type of an existing class
  - new class absorbs the existing class's members and extends them with new or modified capabilities
# protected access level

<table>
<thead>
<tr>
<th>Modifier</th>
<th>accessible within</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>same class</td>
</tr>
<tr>
<td>private</td>
<td>Yes</td>
</tr>
<tr>
<td>package (default)</td>
<td>Yes</td>
</tr>
<tr>
<td>protected</td>
<td>Yes</td>
</tr>
<tr>
<td>public</td>
<td>Yes</td>
</tr>
</tbody>
</table>
**protected access level**

- protected members of a superclass are directly accessible from inside its subclasses.

```java
public class Person {
    protected String name;
    protected String birthday;
    ... 
}

public class Employee extends Person {
    protected int salary;
    public String toString() {
        String s;
        s = name + "\n", " + birthday;
        s += ", " + salary;
        return s;
    }
}
```

Subclass can directly access superclass's protected members.
public class Person {
    private String name;
    private String birthday;
    protected String getName() {
    }
}

public class Employee extends Person {
    protected int salary;
    public String toString() {
        String s;
        s = getName() + "," + getBirthday();
        s += "," + salary;
        return s;
    }
}
Each subclass object figuratively **contains** a superclass object.

Therefore

- In a sense, the subclass object has all superclass object’s data, even though it might not inherit them all (remember private members?)
- In all cases, it should ask the superclass object to initialize its data
Object constructors

```java
public class Animal {
    private String name;

    protected Animal(String _name) {...}
}
```

```java
public class Cow extends Animal {
    private boolean givesMilk;

    public Cow(String _name, boolean _givesMilk) {
        super(_name);
        ...
    }
}
```

The call to superclass constructor must be the **first statement** in subclass constructor.

Invokes `Animal(_name)`
Rules for overriding

- The principle: the subclass must be able to do anything the superclass declares
- Therefore, what are the rules for
  - Parameter types?
  - Return types?
  - Method accessibility?
Overriding – parameter types

public class Employee {
    public void f(A x) {...}
}

public class Manager extends Employee {
    public void f(B x) {...}
}

Employee e1, e2;
// .............  e2 = new Manager();  // we don’t know and don’t care

e1.f(x);  // x of type A
e2.f(x);   // this must work anyway

What’s the rule for A and B?
Overriding – return types

```java
public class Employee {
    public A g() {...}
}
```

```java
public class Manager extends Employee {
    public B g() {...}
}
```

```java
Employee e1, e2;
// ............ e2 = new Manager(); // we don’t know and don’t care
A x = e1.g();
x = e2.g(); // this must work anyway
```

What’s the rule for A and B?
Overriding – method accessibility

What’s the rule for A and B?

```java
public class Employee {
    public A g() {...}
}
```

```java
public class Manager extends Employee {
    private A g() {...}
}
```

```java
Employee e1 = new Employee();
Employee e2 = new Manager();
//.................
A x = e1.g();
x = e2.g();
```
Rules for overriding

- The principle: the subclass must be able to do anything the superclass declares

- Therefore,
  - Parameter types must be the same
    - whatever the superclass takes as an argument, the subclass overriding the method must be able to take that same argument.
  - Return types must be compatible
    - whatever the superclass declares as return type, the subclass must return the same type or a subclass type.
  - The method can't be less accessible
    - a public method cannot be overridden by a private version
Wrong overriding

This is NOT an override!
Can't change the arguments in an overriding method!

This is actually a legal overload, but not an override.

NOT LEGAL!
It's not a legal override because you restricted the access level. Nor is it a legal overload, because you didn't change arguments.
What does inheritance buy you?

1. You avoid duplicate code
   - Common features are put in one place

2. You define a common protocol for a group of classes
   - Objects of a subclass are guaranteed to have all features of the superclass.
   - Objects of a subclass can be treated as if they are objects of the superclass.
   - *Polymorphism!*
Polymorphism

- Normally,
  \[
  \text{Dog} \text{ dog} = \text{new} \text{ Dog}();
  \]

- With polymorphism:
  \[
  \text{Animal} \text{ dog} = \text{new} \text{ Dog}();
  \]

The reference type can be a superclass of the actual object type.
Polymorphic arrays

- An array is declared of type Animal. It can hold objects of Animal's subclasses.

```java
Animal[] animals = new Animal[5];
animals[0] = new Dog();
animals[1] = new Cat();
animals[2] = new Wolf();
animals[3] = new Hippo();
animals[4] = new Lion();
for (int i = 0; i < animals.length; i++) {
    animals[i].makeNoise();
}
```

- We put objects of any subclasses of Animal in the Animal array.
- We can loop through the array and call Animal-class methods, and every object does the right thing!
- The cat runs Cat's version of makeNoise(), the dog runs Dog's version,...
Polymorphic arguments and return types

- Parameters of type Animal can take arguments of any subclasses of Animal.

```java
class Vet {
    public void giveShot(Animal a) {
        // give a a shot, vaccination for example
        a.makeNoise();
    }
}
```

```java
Vet v = new Vet();
Dog d = new Dog();
Cat c = new Cat();
v.giveShot(d);
v.giveShot(c);
```

- it takes arguments of types Dog and Cat
- the Dog's makeNoise() is invoked
- the Cat's makeNoise() is invoked
class Animal {
    String name;
    ...
    public void makeNoise() {
        System.out.print("Hmm.");
    }
    public void introduce() {
        makeNoise();
        System.out.println("I'm " + name);
    }
}

class Cat extends Animal {
    ...
    public void makeNoise() {
        System.out.print("Meow...");
    }
}

class Cow extends Animal {
    ...
    public void makeNoise() {
        System.out.print("Moo...");
    }
}

Animal pet1 = new Cat("Tom Cat");
Animal pet2 = new Cow("Mini Cow");
pet1.introduce();
pet2.introduce();

Meow... I'm Tom Cat
Moo... I'm Mini Cow
What is polymorphism?

- Polymorphism: *exist in many forms*
- Object polymorphism:
  - Objects of subclasses can be treated as if they are all objects of the superclass.
    - A Dog object can be seen as an Animal object as well
  - Even when treated uniformly, objects of different subclasses interpret the same message differently
    - anAnimal.makeNoise() works differently depending on what kind of Animal anAnimal is currently referring to.
What polymorphism buy you?

- With polymorphism, you can write code that doesn't have to change when you introduce new subclass types into the program.

```java
Animal[] animals = new Animal[5];
...
for (int i = 0; i < animals.length; i++) {
    animals[i].makeNoise();
}

class Vet {
    public void giveShot(Animal a) {
        // give a a shot, vaccination for example
        a.makeNoise();
    }
}
```
class Animal {
    ...  
    public void makeNoise() {
        System.out.print("Hmm.");
    }
    public void introduce() {
        makeNoise();
        System.out.println("I'm " + name);
    }
}

class Pig extends Animal {
    public void makeNoise() {
        System.out.print("Oi oi...");
    }
}

class Duck extends Animal {
    public void makeNoise() {
        System.out.print("Quack quack...");
    }
}

You can add as many new animal types as you want without having to modify the introduce() method!

Separate things that change from things that stay the same
Object class

- All classes are subclasses to the class Object
- Inherited methods:
  - Class getClass()
  - int hashCode()
  - boolean equals()
  - String toString()

\[
\text{Car c1 = new Car();}
\text{Car c2 = new Car();}
\]

System.out.println(c1.equals(c2));
System.out.println(c1.getClass() + c1.hashCode());
System.out.println(c1.toString() + "," + c2);