Lecture 3
Inheritance, Interfaces, Polymorphism

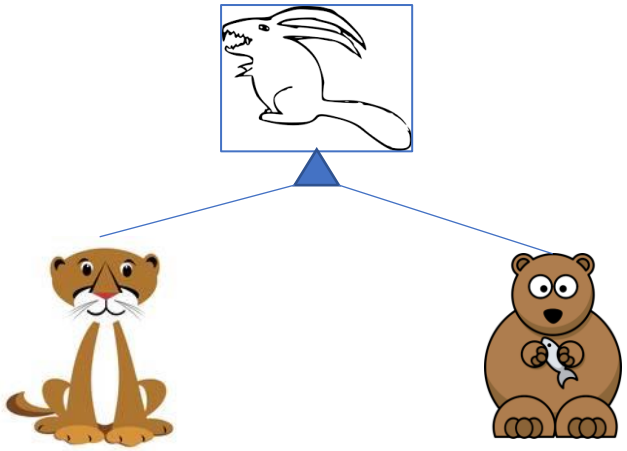
Creating New Types (Classes)

- Rarely we build the entire new class from scratch: we reuse other classes

- There are two primary techniques for doing this
  - Composition (Aggregation)
  - Inheritance

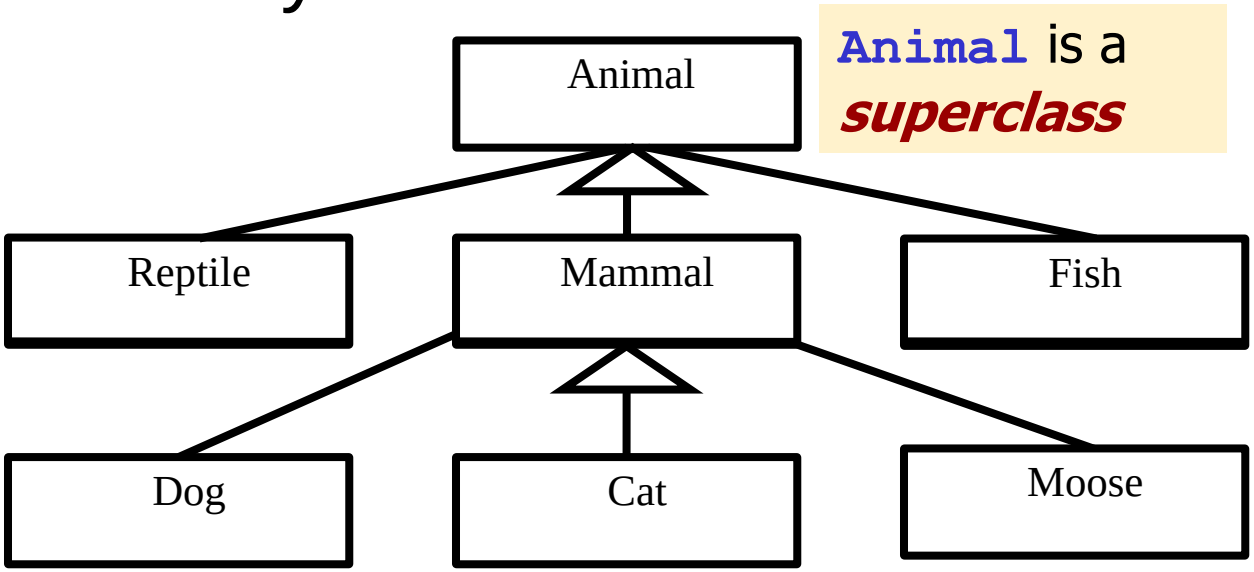
Factoring-out similarities

- When we define a set of new types (classes) we often find that there are similarities among them
- For example:
  - Class ***Tiger*** and class ***Bear*** – both have a lot in common:
    move(), eat(), sleep(), makeNoise()
  - Instead of repeating these methods for each class, we can factor out similarities and define these methods in a single class *Animal*



Tiger **is a**n Animal Bear **is a**n Animal

Inheritance hierarchy



`Animal` is a ***superclass***

Subclass of class Animal

- Where there's inheritance, there's an *Inheritance Hierarchy* of classes
  - **`Mammal`** "*is an*" **`Animal`**
  - **`Cat`** "*is a*" **`Mammal`**
  - Transitive relationship: **`Cat`** "*is an*" **`Animal`** too

- We can say:
  - **`Reptile, Mammal`** and **`Fish`** *inherit from* **`Animal`**
  - **`Dog, Cat,`** and **`Moose`** *inherit from* **`Mammal`**

Inheriting properties (fields) and capabilities (methods)

- Subclass *inherits* all capabilities of its superclass
  - if `Animal`s eat and sleep, then `Reptile`s, `Mammal`s, and `Fish` eat and sleep
  - if `Vehicle`s move, then `SportsCar`s move

- Subclass *specializes* its superclass
  - ***adding*** new fields and methods
  - ***overriding*** *(redefining)* existing methods

- **Superclass** *factors out* capabilities ***common*** among its subclasses
- **Subclasses** are defined by their ***differences*** from their superclass

Inheritance: constructor of a subclass

- A subclass inherits all the ***protected*** members (fields, methods, and nested classes) from its superclass
- **Constructors are not inherited by subclasses**, but the constructor of the superclass can be invoked from the subclass

```
public class Animal {
    public Animal() {
        this.name = "?";
        this.energyLevel = 100;
        this.x = 0;
        this.y=0;
    }

    public Animal(String name) {
        this();
        this.name = name;
    }

    public Animal(String name, String diet) {
        this(name);
        this.diet = diet;
    }
}
```

using *super*() is not compulsory. Even if *super*() is not used in the subclass constructor, the compiler implicitly calls the default constructor of the superclass.

```
public class Cat extends Animal{
    public Cat() {
        super("Cat", "mice");
    }
    …
}
```

When to use inheritance

- When one class is a more specific version of another:

SportsCar extends *Car*

- When you have a method that is the same for a set of classes:

Square, *Circle*, *Triangle* all need to have *move()* method in the animation program, so make *Shape* their superclass

- Test:
  - if you can say: **X IS A Y**, then use **inheritance**
  - If you can say: **X HAS A Y** use **composition**

“IS A” test

- Which of the following is the correct use of inheritance:

A. class *Oven* extends *Kitchen*

B. class *Guitar* extends *Instrument*

C. class *Ferrari* extends *Engine*

D. class *Person* extends *Student*

E. None of the above

What is printed?

```java
public class A {
    int iVar;

    public void hello() {
        System.out.println("Hello from A: " + iVar);
    }

    public void work() {
        iVar ++;
    }
}

public class B extends A{
    public void work() {
        iVar += 5;
    }
}

public class C extends A {
    public void hello () {
        System.out.println("Hello from C: " + iVar);
    }

}
```

```java
IN MAIN:
A a = new A();
B b = new B();
C c = new C();

a.work();
b.work();

a.hello();
b.hello();
c.hello();
```

- A

```
Hello from A: 1
Hello from B: 5
Hello from C: 0
```

- B

```
Hello from A: 6
Hello from B: 5
Hello from C: 6
```

- C

```
Hello from A: 1
Hello from A: 5
Hello from C: 0
```

- D

None of the above

Why use inheritance

- Get rid of duplicate code by factoring out and implementing common behavior
- Modify in one place, and the change is ‘magically’ carried out to all subclasses
- Add new subclasses easily, and they have some methods and properties right away
- **Guarantee that all classes grouped under a certain supertype have a common protocol**

Polymorphism

- The reference and the object can be of different types in Java:

```
Animal c = new Cat();
Superclass   Subclass
```

- Subclass **is a** superclass, and subclass objects can be assigned to superclass variables
- Not vice versa!
- Superclass IS NOT a subclass and superclass objects cannot be assigned to subclass variables

- `c` can be used both as an *Animal* and as a *Cat*
- `c` has "many forms" – **polymorphism**

- We can use polymorphic variables as **method arguments**, **return types** or **array types**

Example: superclass and a subclass

```java
public class Animal {
    protected String name;
    protected int energyLevel, x, y;
    protected String diet;

    public String getName() {return this.name;}

    public void move(int dX, int dY) {
        this.x += dX;
        this.y += dY;
        this.energyLevel-=(dX + dY);
    }

    public void eat() {
        System.out.println(name +
                    " is eating " + diet);
        this.energyLevel ++;
    }

    public void sleep() {
        this.energyLevel ++;
    }

    public void makeNoise() {
    }
}
```

```java
public class Cat extends Animal{
    public Cat() {
        super("Cat", "mice");
    }

    public void eat() {
        System.out.println("Cat is eating "
                                + diet);
        this.energyLevel += 3;
    }

    public void makeNoise() {
        System.out.println("Purrr");
    }
}
```

Example: polymorphism

- Because *Dog, Cat* and *Lion* are also *Animals*, we can store them in array of *Animals*
- *makeNoise* is declared in *Animal* (though it has an empty body), so we can call it on each element of the *Animal* array

```
public class Animals {
    public static void main(String [] args) {
        Animal [] animals = new Animal[3];

        animals[0] = new Dog();
        animals[1] = new Cat();
        animals[2] = new Lion();

        for (Animal a: animals) {
            System.out.println(a);
            a.makeNoise();
        }
    }
}
```

Each animal makes their own noise

Java classes: single-root hierarchy

- All classes in Java (including our new custom classes) are subclasses of a single root superclass called *Object*
- When we create a new class that does not extend anything, this means implicitly:

```
public class Dog extends Object
```

So what's in *Object?*

- Important public methods implemented in Object (see here):

```
public String toString();
public boolean equals(Object obj);
public int hashCode();
```

- If you do not override these methods, you inherit them from the Object class

Extreme polymorphism

- This means that *Dog* inherits all the methods of *Object* That also means that you can have a variable of class Object and store in it a reference to any other type:

```
Object o = new Dog();
o = new String("hello");
```

However with this declaration we can only call the methods of the *Object* class from variable *o*

Abstract classes

- We factored out all the common code into class Animal

- However a generic Animal does not know how:

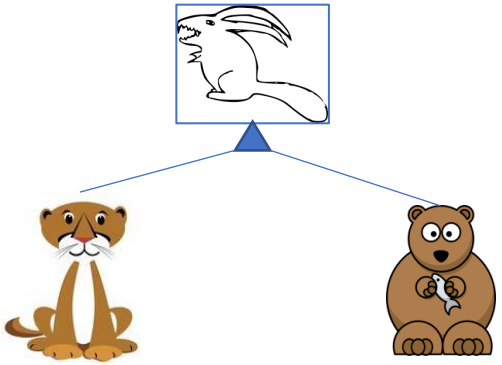
  `makeNoise()`

  `getPicture()`

  `getColor()`

  …

- All these methods are not applicable to a generic class Animal



Lion **is a**n Animal Bear **is a**n Animal

How does Animal() look like?

We want to prevent anyone from making an instance of Animal()

Animal class is too abstract!

Declare Animal to be an **abstract** class

```
public abstract class Animal {
    protected String name;
    protected int energyLevel;
        ...

    public void move(int dX, int dY) {
        this.x += dX;
        this.y += dY;
        this.energyLevel --;
    }

    public void eat() {
        ...
        this.energyLevel ++;
    }

    public void sleep() {
        this.energyLevel ++;
    }

    public abstract void makeNoise();

    public abstract Picture getPicture();
```

- Shared code which is applicable to all subclasses is still in concrete methods
- We can declare all the other methods ***abstract***
- Abstract methods do not have body

- If the class has at least one abstract method, it must be declared abstract
- You must implement all abstract methods in a subclass, if you want to create instances of this subclass

No instances of abstract Animals

```
public abstract class Animal {
    protected String name;
    protected int energyLevel;
        ...

    public void move(int dX, int dY) {
        this.x += dX;
        this.y += dY;
        this.energyLevel --;
    }

    public void eat() {
        ...
        this.energyLevel ++;
    }

    public void sleep() {
        this.energyLevel ++;
    }

    public abstract void makeNoise();

    public abstract Picture getPicture();
```

- **You cannot create instances of an abstract class:**

```
Animal a = new Animal();
```

This will not compile

Why we use Abstract classes

- Inheritance allows to store shared code in a superclass
- But sometimes we cannot find any generic code useful to all subclasses
- In this case we declare a method in the superclass *abstract* (and the entire superclass becomes abstract)
- Even though there is no code in an abstract method, it still **defines a common protocol** that can be used in polymorphic programs: each subclass of Animal must know how to *makeNoise*()
- Compiler **forces** the concrete subclasses of an abstract class to implement all the abstract methods
- If a subclass did not implement all the abstract methods, then it by itself must be declared abstract

Polymorphism and Dynamic (late) Binding

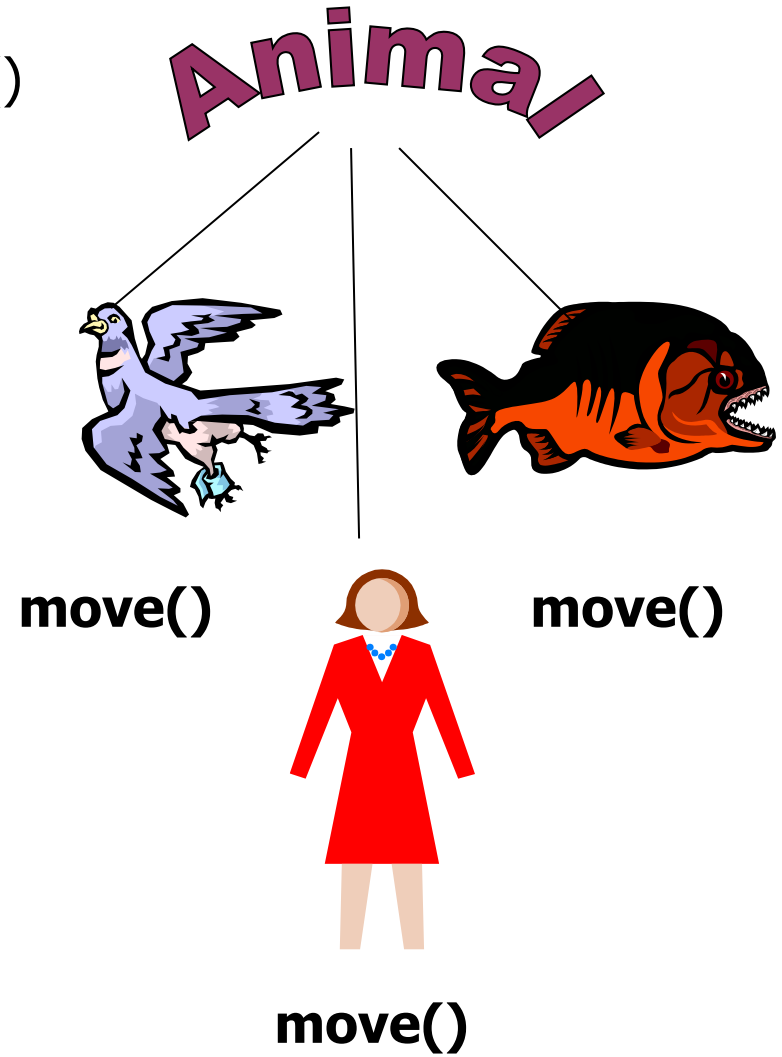
- Polymorphism is implemented utilizing two important ideas

1. Method overriding
   - A method defined in a superclass is redefined in a subclass with an identical method signature
   - For a subclass object, the definition in the subclass **replaces** the version in the superclass, even if a superclass reference is used to access the object
2. Late binding
   - The code is associated with the method call during run-time
   - The actual method executed is determined by the type of the object, not the type of the reference

Example: Different Ways of Moving

Ex. each subclass overrides the move() method in its own way

```
Animal [] A = new Animal[3];
A[0] = new Bird();
A[1] = new Person();
A[2] = new Fish();
for (int i = 0; i < A.length; i++)
        A[i].move();
```



- Each call is syntactically identical: method signature is the same
- Code executed is based on type of actual object the variable points to
- The JVM finds what actual code to execute during run time (late binding)

Factoring out partial commonalities

- The Animal class defines a contract for all Lion, Hippo, Cat and Dog types
- We can use this hierarchy for Animal Simulation program

- But now we want to reuse some of the code from our Pet Store program
- We want to add *play()* method to some animals but not to all
- Basically we want some of the animals have an additional contract defined in superclass *Pet*

Java solution to multiple-inheritance problem

- Java does not allow a class to extend more than one superclass = it does not allow multiple inheritance
- However we can guarantee *Pet* behavior for all pet animals if we define all shared methods in a special Java class – *Interface*

  Not a GUI interface, not a colloquial use as in "public methods provide interface", but a special Java keyword *Interface*

Pet interface

- In *Interface* **all** methods are abstract
- All subclasses must implement all of them
- Subclass **extends** a Superclass and **implements** Interface

```
public interface Pet {
    public void play();
}

public class Dog extends Animal
                         implements Pet{

    …
    public void makeNoise() {
        System.out.println("Wuff");
    }

    public void play() {
        System.out.println("Dog playing");
        this.makeNoise();
    }

}

public class Cat extends Animal
                         implements Pet{
```

Why use Interface

```
public class PetStore {
    public static void main

        (String [] args) {
         Pet [] pets = new Pet [4];

         pets[0] = new Cat();
         pets[1] = new Cat();
         pets[2] = new Cat();
         pets[3] = new Dog();

         for (Pet p: pets) {
             p.play();
         }
    }
}
```

If all the methods in Interface are abstract – how is this code reuse?

- A subclass can extend one superclass and implement multiple interfaces
- Common interface can be used for polymorphism

Java Interface

- A Java interface is (primarily) a named set of abstract methods
- Think of it as an abstract class with no concrete methods and no instance variables
  - Static constants are allowed
  - Static methods are allowed
  - No instance variables are allowed
  - Regular methods have no bodies
  - Interface itself cannot be instantiated
- Any Java class (no matter what its inheritance) can implement an additional interface by implementing the methods defined in it
- A class can implement any number of interfaces

Example: zoo and pet store simulations