Vietnam National University of HCMC
International University
School of Computer Science and Engineering

Polymorphism

(IT069IU)

Nguyen Trung Ky

ntky@hcmiu.edu.vn

it.hcmiu.edu.vn/user/ntky

Previous lecture

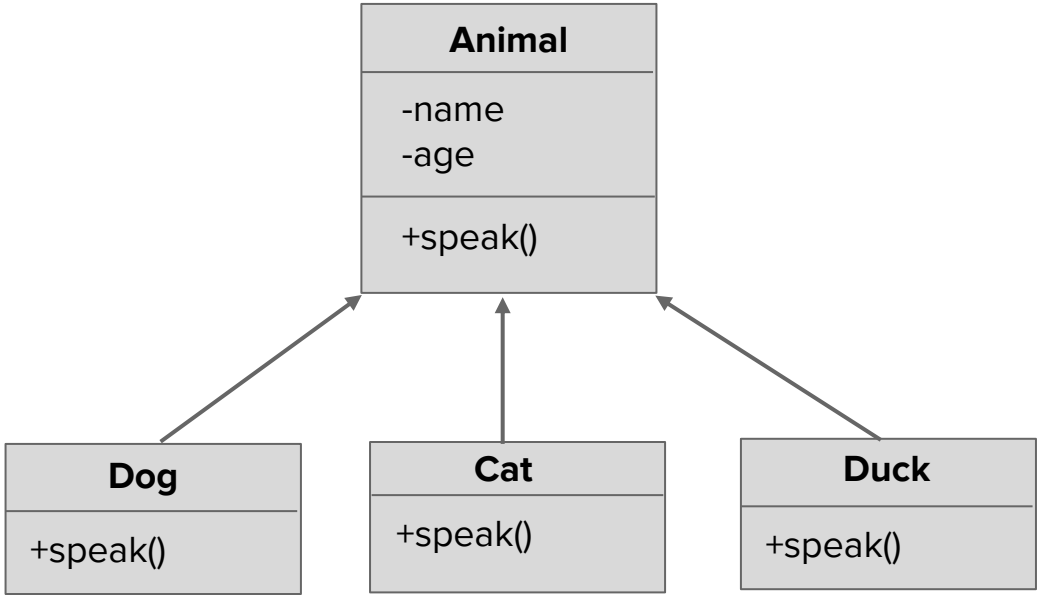
- **Inheritance**
  - Definition and Examples
  - Types of Inheritance
  - UML Diagram
  - Animal Inheritance Example
    - Without Inheritance
    - With Inheritance
    - Method Overriding
  - Constructors in Subclasses
    - Keyword Super
  - Method Overriding
    - Keyword Super
  - Access Modifier (Protected)
- **Overloading**
  - Method Overloading
  - Constructor Overloading
- **Final** Keyword
  - Constant Variable
- **Static** Keyword
  - Static Variable
  - Static Method

Agenda's today

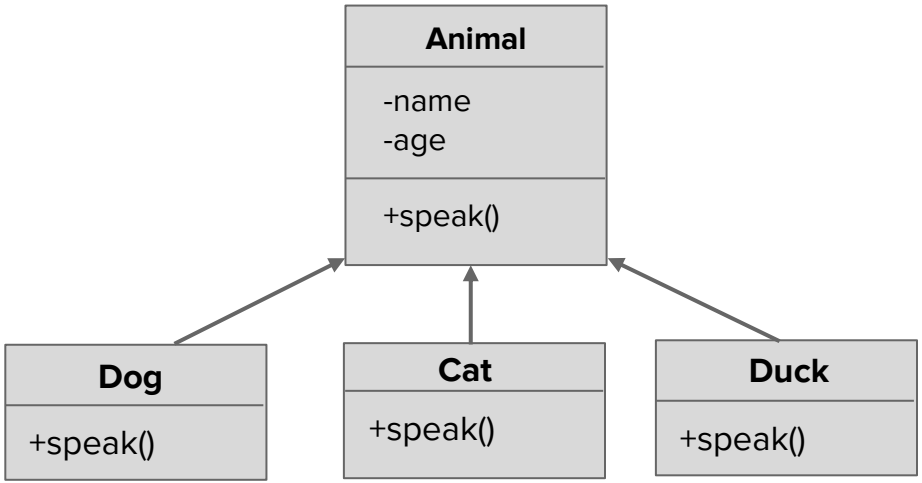
- **Polymorphism**
  - Method overriding and overloading in Inheritance
    - Zoo Example
    - Company Payroll Example
  - **Abstraction**
    - Abstract Class
    - Abstract Method
    - Why we need abstract class
    - Examples:
      - Zoo Example
      - Company Payroll Example
  - **Interface**
    - Interface in real life examples
    - Upgrade Company Payroll with Invoices Example
  - Abstract vs Interface

Polymorphism

- Polymorphism literally means **“many forms”.**
- Polymorphism allows us to perform a single action in different ways. The **same method** is called in **different types of objects** has **different results**. We can perform polymorphism in java by **method overloading** and **method overriding**.
- Polymorphism **enables you to write programs that process objects of subclasses that share the same superclass as if they’re all objects of the superclass**; this **can simplify programming.**



- Each specific type of Animal responds to the method speak() in a unique way:
  - a **Dog** speaks **“Woof”**
  - a **Duck** speaks **“Quack”**
  - a **Cat** speaks **“Meow”**
- The program issues the same message (i.e., speak) to each animal object, but **each object knows how to use its correct method of speaking.**
- **Relying on each object to know how to “do the right thing” in response to the same method call is the key concept of polymorphism**.
- With **polymorphism**, we can **design and implement systems that are easily extensible.**



Let's live code in Java!

Zoo Polymorphism

Superclass Animal

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

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

    public String speak(){
        return "No Sound";
    }

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public int getAge() { return age; }

    public void setAge(int age) { this.age = age; }
}
```

[Info] Just notice for the superclass Animal, we declare the method speak() with a dummy implementation for the body of this method.
(We can improve this later with abstract class)

Class Dog

```java
public class Dog extends Animal{
    public Dog(String name, int age) {
        super(name, age);
    }

    @Override
    public String speak(){
        return "Woof";
    }
}
```

[Info] The subclass Dog inherits superclass Animal and override the method speak() of the superclass.

Class Cat

```java
public class Cat extends Animal{
    public Cat(String name, int age) {
        super(name, age);
    }

    @Override
    public String speak(){
        return "Meow";
    }
}
```

[Info] The subclass Cat inherits superclass Animal and override the method speak() of the superclass.

Class Duck

```java
public class Duck extends Animal{

    public Duck(String name, int age) {
        super(name, age);
    }

    @Override
    public String speak(){
        return "Quack";
    }
}
```

[Info] The subclass Duck inherits superclass Animal and override the method speak() of the superclass

Class ZooPolymorphism for Testing

```java
public class ZooPolymorphism {
    public static void main(String[] args) {

        // You can treat each animal as its subclass
        System.out.println("Treat them as their own subclass:");
        Dog myDog = new Dog("Kiki", 5);
        Duck myDuck = new Duck("Donald", 2);
        Cat myCat = new Cat("Tom", 3);
        System.out.printf("Dog speaks %s\n",myDog.speak());
        System.out.printf("Duck speaks %s\n",myDuck.speak());
        System.out.printf("Cat speaks %s\n",myCat.speak());

        // Or you can treat them as its superclass (Animal)
        System.out.println("\nTreat them as a superclass Animal:");
        Animal anotherDog = new Dog("Corgi", 3);
        Animal anotherDuck = new Duck("Daisy", 4);
        Animal anotherCat = new Cat("Garfield", 2);
        System.out.printf("Dog speaks %s\n",anotherDog.speak());
        System.out.printf("Duck speaks %s\n",anotherDuck.speak());
        System.out.printf("Cat speaks %s\n",anotherCat.speak());
```

Output:

```
Treat them as its subclass
Dog speak: Woof
Cat speak: Meow
Duck speak: Quack
Treat them as its superclass
Dog speak: Woof
Cat speak: Meow
Duck speak: Quack
```

Company Payroll Example

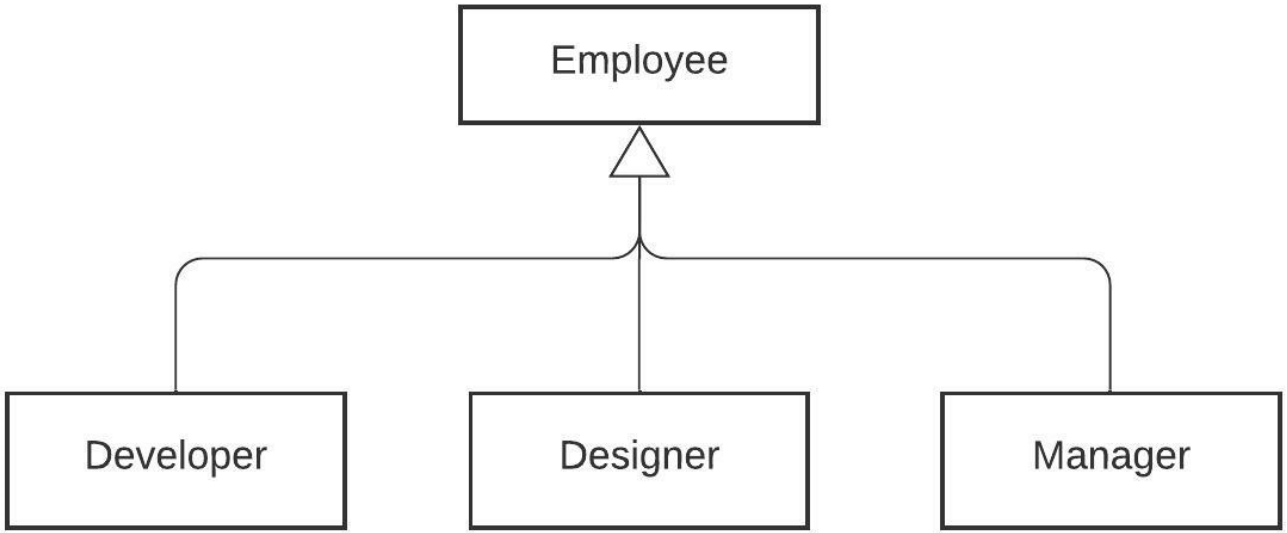
- A company has many employees. Each employee can be:
  - **Developer**
  - **Designer**
  - **Manager**


- Every employee will has a fixed base salary but each type of employee can have a different way to have bonus to be added for their salary:
  - **Developer**:
    - Base Salary + How many projects he did * The bonus for each project.
  - **Designer**:
    - Base Salary + 13th Salary Month if that person does a good job.
  - **Manager**:
    - Base Salary + How big is the team they manage * The bonus to manage each person.

a. Write all necessary classes.

b. Write a Test class which creates three objects: developer, designer and manager and print out all information from these objects.

UML Diagram for Company Payroll



Let's live code in Java!

Superclass Employee

```java
public class Employee {
    private String name;
    private double baseSalary;

    public Employee(String name, double baseSalary) {
        this.name = name;
        this.baseSalary = baseSalary;
    }

    public double earning(){
        return baseSalary;
    }

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public double getBaseSalary() { return baseSalary; }

    public void setBaseSalary(double baseSalary) { this.baseSalary = baseSalary; }
}
```

[Info] This time, for the superclass Employee, we implement the body of the method earning() with something useful.

Class Developer

```java
public class Developer extends Employee {
    private int numberProjects;
    private double bonusPerProject;
    /**
     * @param numberProjects
     * @param bonusPerProject
     */
    public Developer(String name, double baseSalary,
            int numberProjects, double bonusPerProject) {
        super(name, baseSalary);
        this.numberProjects = numberProjects;
        this.bonusPerProject = bonusPerProject;
    }
    public Developer(String name, double baseSalary) {
        super(name, baseSalary);
    }

    public int getNumberProjects() {
        return numberProjects;
    }
    public void setNumberProjects(int numberProjects) {
        this.numberProjects = numberProjects;
    }
    public double getBonusPerProject() {
        return bonusPerProject;
    }
    public void setBonusPerProject(double bonusPerProject) {
        this.bonusPerProject = bonusPerProject;
    }
    @Override
    public double earning() {
        return this.getBaseSalary() + numberProjects*bonusPerProject ;
    }

    public double earning(int numberProjects, double bonusPerProject) {
        return this.getBaseSalary() + numberProjects*bonusPerProject ;
    }

}
```

[Info] The subclass Developer inherits from the superclass Employee and override the method earning() of the superclass.

Class Designer

```
public class Designer extends Employee{
    private boolean bonus13thMonth;

    /**
     * @param bonus13thMonth
     */
    public Designer(String name, double baseSalary,boolean bonus13thMonth) {
        super(name, baseSalary);
        this.bonus13thMonth = bonus13thMonth;
    }

    public Designer(String name, double baseSalary) {
        super(name, baseSalary);
    }


    public boolean isBonus13thMonth() {
        return bonus13thMonth;
    }

    public void setBonus13thMonth(boolean bonus13thMonth) {
        this.bonus13thMonth = bonus13thMonth;
    }

    public double earning(){
        if (bonus13thMonth == true) {
            return (this.getBaseSalary()/12)*13;
        }
        else
            return this.getBaseSalary();
    }
    public double earning(boolean bonus13thMonth) {
        if (bonus13thMonth == true) {
            return (this.getBaseSalary()/12)*13;
        }
        else
            return this.getBaseSalary();
    }
}
```

[Info] The subclass Designer inherits from the superclass Employee and override the method earning() of the superclass.

Class Manager

```java
public class Manager extends Employee {
    private int numTeamMembers;
    private double bonusPerTeamMember;

    public Manager(String name, double baseSalary, int numTeamMembers,
            double bonusPerTeamMember) {
        super(name, baseSalary);
        // TODO Auto-generated constructor stub
        this.numTeamMembers = numTeamMembers;
        this.bonusPerTeamMember = bonusPerTeamMember;
    }

    public Manager(String name, double baseSalary) {
        super(name, baseSalary);
    }

    public int getNumTeamMembers() {
        return numTeamMembers;
    }

    public void setNumTeamMembers(int numTeamMembers) {
        this.numTeamMembers = numTeamMembers;
    }

    public double getBonusPerTeamMember() {
        return bonusPerTeamMember;
    }

    public void setBonusPerTeamMember(double bonusPerTeamMember) {
        this.bonusPerTeamMember = bonusPerTeamMember;
    }
    public double earning() {
        return this.getBaseSalary() + numTeamMembers*bonusPerTeamMember;
    }
    public double earning(int numTeamMembers, double bonusPerTeamMember ) {
        return this.getBaseSalary() + numTeamMembers*bonusPerTeamMember;
    }

}
```

[Info] The subclass Manager inherits from the superclass Employee and override the method earning() of the superclass.

Class Company for Testing

```java
public class Company {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Designer trangDesigner = new Designer("Trang", 1000);
        System.out.printf("%s earn as a Designer $%.2f \n", trangDesigner.getName(), trangDesigner.earning(true));

        Developer tomDev = new Developer("Tom", 2000);
        System.out.printf("%s earn as a Developer $%.2f \n", tomDev.getName(), tomDev.earning(4, 400));

        Manager lyMan = new Manager("Ly", 2000);
        System.out.printf("%s earn as a Designer $%.2f \n", lyMan.getName(), lyMan.earning(10, 50));
    }
}
```

Output:

```
Trang earn as a Designer $1083.33
Tom earn as a Developer $3600.00
Ly earn as a Designer $2500.00
```

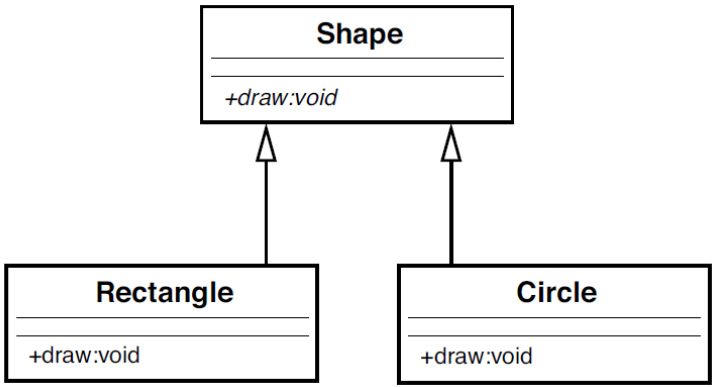
Why do we need abstract class

```
public class Company {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Designer trangDesigner = new Designer("Trang", 1000);
        System.out.printf("%s earn as a Designer $%.2f \n", trangDesigner.getName(), trangDesigner.earning(true));

        Developer tomDev = new Developer("Tom", 2000);
        System.out.printf("%s earn as a Developer $%.2f \n", tomDev.getName(), tomDev.earning(4, 400));

        Manager lyMan = new Manager("Ly", 2000);
        System.out.printf("%s earn as a Designer $%.2f \n", lyMan.getName(), lyMan.earning(10, 50));

        //Why we need abstract class and method
        //Employee xuanDesigner = new Designer("Xuan", 1000);
        //System.out.printf("%s earn as a Designer $%.2f \n", xuanDesigner.getName(), xuanDesigner.earning(false));
    }
}
```

Abstraction

Abstraction

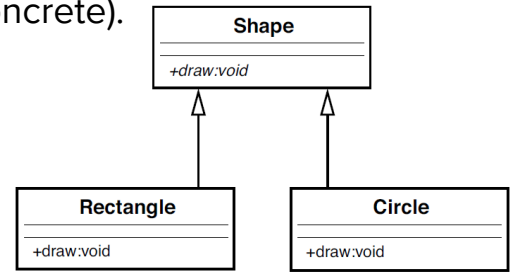
- An **abstract class** is a **class** that **contains** one or more **methods** that **do not have any implementation provided**.
- For example that you have an abstract class called Shape. It is **abstract** because you **cannot instantiate (create)** it.
  - If you ask someone to draw a shape, the first thing the person will most likely ask you is, “What kind of shape?” Thus, the concept of a shape is **abstract**.
  - However, if someone asks you to draw a circle, this is easier because a circle is a **concrete concept**. You know what a circle looks like. You also know how to draw other shapes, such as rectangles.



Abstraction in Shape

The class Shape does not provide any implementation for draw() ; basically there is no code, and this is what makes the method abstract (providing any code would make the method concrete).

```
public abstract class Shape {

    public abstract void draw(); // no implementation

}
```



We want the subclasses to provide the implementation. Let's look at the Circle and Rectangle classes:

```
public class Circle extends Shape {

    public void Draw() {System.out.println ("Draw a Circle")};

}

public class Rectangle extends Shape {

    public void Draw() {System.out.println ("Draw a Rectangle")};

}
```

```
circle.draw();          // draws a circle
rectangle.draw();       // draws a rectangle
```

The Draw method can be invoked for every single shape in the system, and invoking each shape produces a different result:
- Invoking the Draw method on a Circle object draws a circle.
- Invoking the Draw method on a Rectangle object draws a rectangle.

In essence, sending a message to an object evokes a different response, depending on the object. This is **the essence of polymorphism**.

Abstract Class

- Sometimes it's useful to declare **classes for which you never intend to create objects.**
  - For Example, we created superclass like Animal or Employee but we would never create any object from it but only use it to extends other subclasses like Dog or Cat.
- **Abstract Class** allow you to **create blueprints for concrete classes**. An abstract class provides a superclass from which other classes can inherit and thus share a common design.
- **Abstract Class** can **contains**:
  - **At least one Abstract method**: method without implementation (no body).
  - **Zero or more Concrete method**: normal method with implementation.
- Abstract Class **cannot be used to create objects** because it is **incomplete**. Abstract class are **too general to create real objects**—they specify only what is common among subclasses.
- **Subclasses must override abstract method (provide the implementations) to become "concrete" classes**, which you **can create objects**; **otherwise, these subclasses, too, will be abstract**, which cannot create objects.

Abstract Class Syntax

- You **make a class abstract by declaring** it with **keyword abstract**, for example:

```
public abstract class Animal {...}  // abstract class
```

- An **abstract class** normally contains **one or more abstract methods**

```
public abstract void eat();  // abstract method
```

- Abstract methods **do not provide implementations**. (Without a body)
- **A class that contains abstract methods must be an abstract class** even if that class contains some concrete (non-abstract) methods.
- Each **concrete subclass** of an abstract superclass also **must provide concrete implementations of each of the superclass's abstract methods**.
- **Constructors** and **static methods cannot be declared abstract**.
- Can use **abstract superclass** names to **invoke static methods declared in those abstract superclasses.**

Revisited Zoo with Abstract Class Animal

- Remember we would **never create any object from superclass like Animal** but **only use it to extends other subclasses** like Dog, Cat and Duck.
- Also, we got to provide a **dummy implementation** for the method speak() even though.

```
// We don't create any object directly from class Animal
public class Animal {

    // a dummy implementation of method speak()
    public String speak() {
        return "No Sound";
    }

}
```

- With **abstract class**, we can **indicate that class should never be used to create object directly**. Also, the **abstract class can provide abstract methods for other classes to inherit it to provide specific implementation.**

Abstract Class Zoo

```java
// Abstract class Animal
public abstract class Animal {
    private String name;
    private int age;

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

    // abstract method with no implementation here
    public abstract String speak();

    @Override
    public String toString() {
        return String.format("%s is %d year old", getName(), getAge());
    }

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public int getAge() { return age; }

    public void setAge(int age) { this.age = age; }
}
```

[Info] The class Animal contains one **abstract method speak()** that is why class Animal is an **abstract class**.

Also, because **method speak() is abstract**, we **don't need to provide any implementation** and **leave that responsibility to other subclasses to complete that abstract method.**

Class Dog

```java
public class Dog extends Animal{
    public Dog(String name, int age) {
        super(name, age);
    }

    // override abstract method speak() in Animal
    // to make Dog class to be a concrete class
    @Override
    public String speak(){
        return "Woof";
    }

    @Override
    public String toString() {
        return String.format("%s, speaks %s.\n",
                super.toString(), this.speak());
    }
}
```

[Info] The class Dog inherits the abstract superclass Animal and it does complete the implementation of the abstract method by overriding it so that's why class Dog is a **concrete class** (which can be used to create objects of class Dog)

Class Duck

```java
public class Duck extends Animal{
    public Duck(String name, int age) {
        super(name, age);
    }

    // override abstract method speak() in Animal
    // to make Duck class to be a concrete class
    @Override
    public String speak(){
        return "Quack";
    }

    @Override
    public String toString() {
        return String.format("%s, speaks %s.\n",
                super.toString(), this.speak());
    }
}
```

[Info] The class Duck inherits the abstract superclass Animal and it does complete the implementation of the abstract method by overriding it so that's why class Duck is a **concrete class** (which can be used to create objects of class Duck)

Class Cat

```java
public class Cat extends Animal{
    public Cat(String name, int age) {
        super(name, age);
    }

    // override abstract method speak() in Animal
    // to make Cat class to be a concrete class
    @Override
    public String speak(){
        return "Meow";
    }

    @Override
    public String toString() {
        return String.format("%s, speaks %s.\n",
                super.toString(), this.speak());
    }
}
```

[Info] The class Cat inherits the abstract superclass Animal and it does complete the implementation of the abstract method by overriding it so that's why class Cat is a **concrete class** (which can be used to create objects of class Cat)

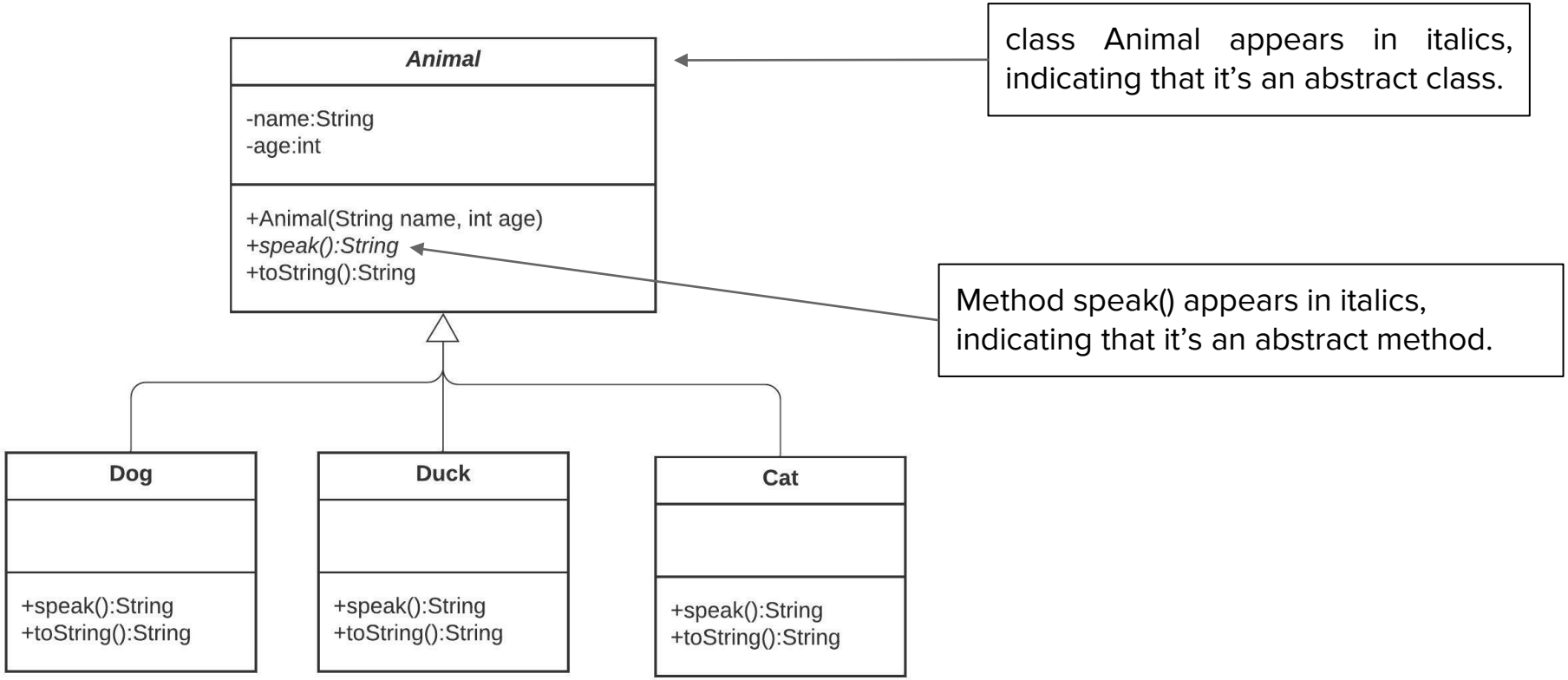
Class Zoo with main method for Testing

```
public class ZooPolymorphism {
    public static void main(String[] args) {

        // Not possible anymore to create an object from the abstract class Animal
        // Animal myAnimal = Animal("James", 5);

        // You can treat each animal as its subclass individually
        System.out.println("Treat them individually as their own subclass:");
        Dog myDog = new Dog("Kiki", 5);
        Duck myDuck = new Duck("Donald", 2);
        Cat myCat = new Cat("Tom", 3);
        System.out.printf("Dog speaks %s\n",myDog.speak());
        System.out.printf("Duck speaks %s\n",myDuck.speak());
        System.out.printf("Cat speaks %s\n",myCat.speak());



        // Or you can treat them as its superclass (Animal) (Polymorphism)
        System.out.println("\nTreat them as a superclass Animal (Polymorphism):");
        Animal anotherDog = new Dog("Corgi", 3);
        Animal anotherDuck = new Duck("Daisy", 4);
        Animal anotherCat = new Cat("Garfield", 2);
        System.out.printf("Dog speaks %s\n",anotherDog.speak());
        System.out.printf("Duck speaks %s\n",anotherDuck.speak());
        System.out.printf("Cat speaks %s\n",anotherCat.speak());

        // Advantage: organize and group them into an array or ArrayList of Animal
        System.out.println("\nGroup them into an collection of Animal type:");
        Animal[] myZoo = new Animal[3];
        myZoo[0]=anotherDog;
        myZoo[1]=anotherDuck;
        myZoo[2]=anotherCat;
        for (int i=0; i<myZoo.length; i++){
            System.out.print(myZoo[i]);
        }
    }
}
```

Output:

```
Treat them individually as their own subclass:
Dog speaks Woof
Duck speaks Quack
Cat speaks Meow

Treat them as a superclass Animal (Polymorphism):
Dog speaks Woof
Duck speaks Quack
Cat speaks Meow

Group them into an collection of Animal type:
Corgi is 3 year old, speaks Woof.
Daisy is 4 year old, speaks Quack.
Garfield is 2 year old, speaks Meow.
```

UML Diagram for Abstract Class and Abstract Method



Revisited Payroll System with Abstract Class

- Use an abstract method and polymorphism to perform payroll calculations based on the type of inheritance by an employee.
- **Abstract class Employee represents the general concept of an employee.**
- **Subclasses**: Developer, Designer, Manager.
- **Abstract superclass Employee declares the "protocol" —that is, the set of methods that a program can invoke on all Employee objects.**
  - We use the term "protocol" here in a general sense to refer to the various ways programs can communicate with objects of any Employee subclass.
  - Each employee has a name and a base salary defined in abstract superclass Employee.

Revisited Payroll System with Abstract Class

- **Class Employee provides methods earnings and toString**, in addition to the get and set methods that manipulate Employee's instance variables.
- **An earnings method applies to all employees, but each earnings calculation depends on the employee's class.**
  - An **abstract method—there is not enough information to determine what amount earnings should return.**
  - **Each subclass overrides earnings with an appropriate implementation.**
- **Iterate through the array of Employees and call method earnings for each Employee subclass object.**
  - **Method calls processed polymorphically.**
- Declaring the earnings method abstract indicates that each **concrete subclass must provide an appropriate earnings implementation** and that a program will be able to use **superclass Employee variables to invoke method earnings polymorphically for any type of Employee**.

Interface

Abstraction on steroid

Interface in real life (Piano)

Interface

- An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.
- **The interface in Java is *a mechanism to achieve* *abstraction***. There can be only abstract methods in the Java interface, not method body.
- It is used to achieve abstraction and multiple inheritance in Java.

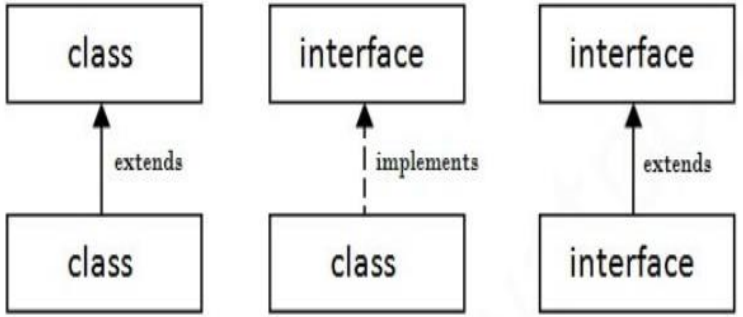
- **Interface offers a capability requiring that unrelated classes implement a set of common methods**.
- Java Interface also **represents the IS-A relationship**.
- It cannot be instantiated just like the abstract class.

Interface Syntax

Declare an interface with the name "InterfaceName" with two methods:

```
// declare an interface
public interface InterfaceName {
    // interface method
    public void myMethod();

    // interface method
    public double anotherMethod(int x, int y);
}
```

Relationship between class and interface:



A class extends another class, an interface extends another interface, but a **class implements an interface**.

Revisited the Company Payroll

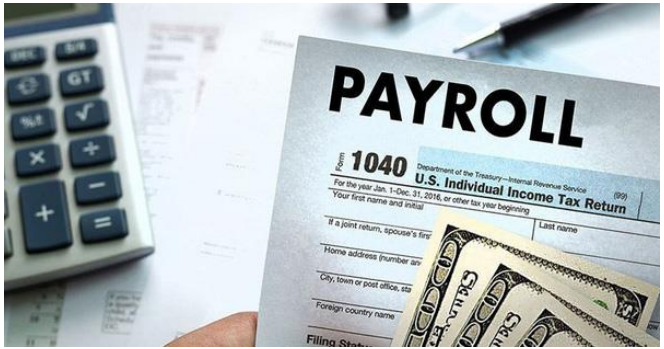
Suppose that the company involved wishes to **perform several accounting operations** in a **single accounts payable application**:

- **Calculate the earnings that must be paid to each employee**,
- **Calculate the payment due on each of several invoices** (i.e., bills for products purchased).

Though applied to **unrelated things** (employees and invoices), **both want to do a kind of payment amount:**

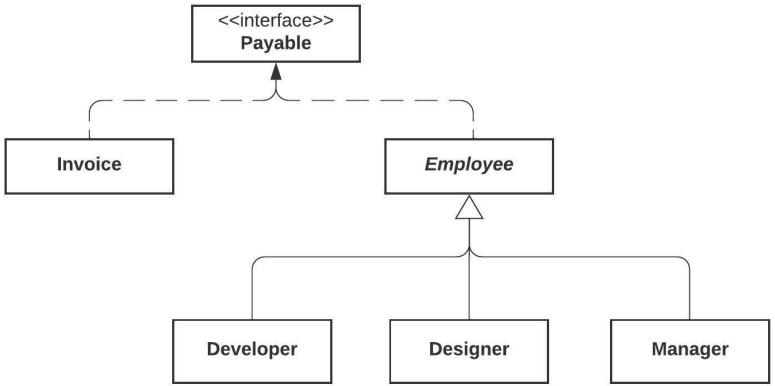
- For an **employee**, the payment refers to the **employee's earnings.**
- For an **invoice**, the payment refers to the **total cost of the goods listed on the invoice**.

Can we calculate such different things as the payments due for employees and invoices in a single application polymorphically? Does Java offer a capability requiring that unrelated classes implement a set of common methods (e.g., a method that calculates a payment amount)?



Upgrade our Company Payroll with Interface

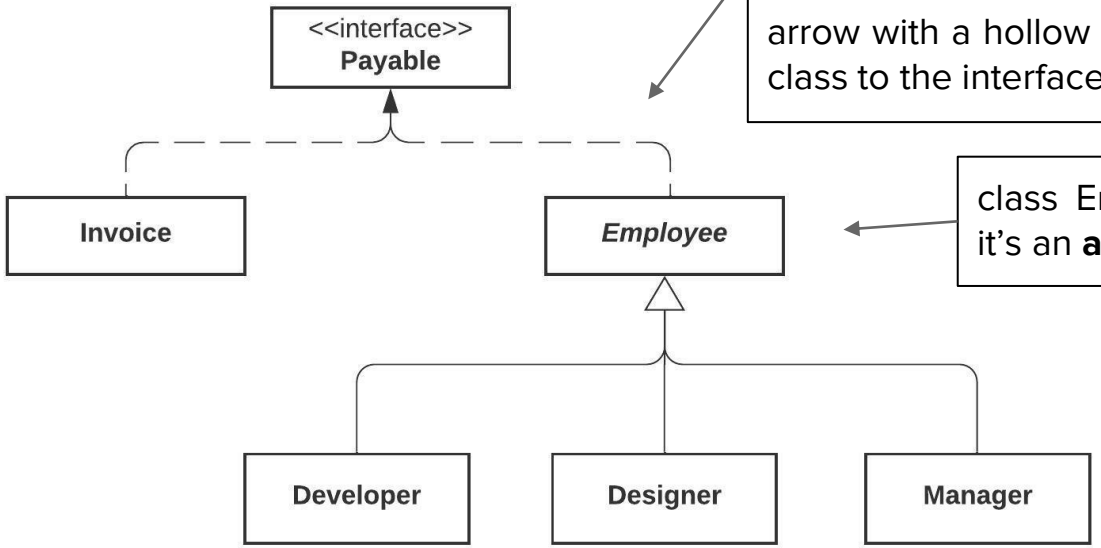
- To build an application that can determine payments for employees and invoices alike, we first **create interface Payable**, which **contains method getPaymentAmount()** that **returns the amount that must be paid for an object of any class that implements the interface.**
- **Method getPaymentAmount()** is a **general-purpose version of method earnings().**
- Classes Invoice and Employee both represent things for which the company must be able to calculate a payment amount. Both classes **implement the Payable interface**, so a program **can invoke method getPaymentAmount() on Invoice objects and Employee objects alike.**



UML Diagram

The UML distinguishes an interface from other classes by placing the word “interface” in guillemets (« and ») above the interface name.

The UML expresses the **relationship between a class and an interface** through a relationship known as **realization**. A class is said to realize, or implement, the methods of an interface. A class diagram models a realization as a dashed arrow with a hollow arrowhead pointing from the implementing class to the interface.



class Employee appears in **italics**, indicating that it’s an **abstract class**.

Concrete class Developer, Designer and Manager extends Employee, inheriting its superclass’s realization relationship with interface Payable.

Interface Payable

- To build an application that can determine payments for employees and invoices alike, we first **create interface Payable**, which **contains method getPaymentAmount()** that **returns the amount that must be paid for an object of any class that implements the interface.**

```
// Payable interface declaration
public interface Payable {

    double getPaymentAmount(); // calculate payment with no implementation

}
```

Recap

- **Polymorphism**
  - Method overriding and overloading in Inheritance
    - Zoo Example
    - Company Payroll Example
  - **Abstraction**
    - Abstract Class
    - Abstract Method
    - Why we need abstract class
    - Examples:
      - Zoo Example
      - Company Payroll Example
  - **Interface**
    - Interface in real life examples
    - Upgrade Company Payroll with Invoices Example
  - Abstract vs Interface

Thank you for your listening!

“Motivation is what gets you started. Habit is what keeps you going!”

Jim Ryun