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

Polymorphism

(IT069IU)

Le Duy Tan, Ph.D.

ldtan@hcmiu.edu.vn

leduytanit.com

Previously,

- **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
  - Exercise for University Staff & Lecturer

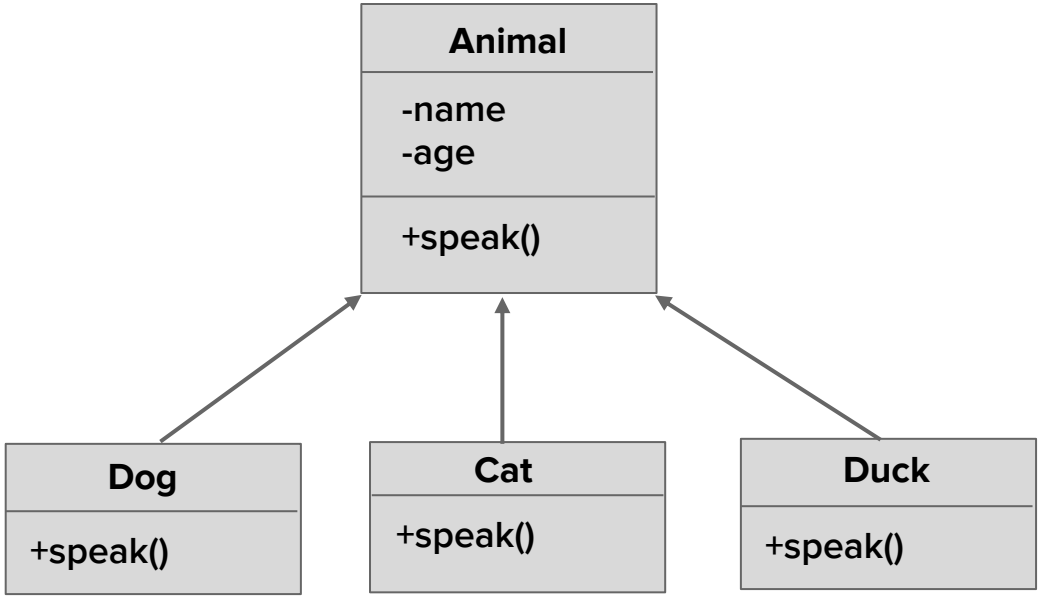
"DRY (Don't Repeat Yourself)"

Agenda

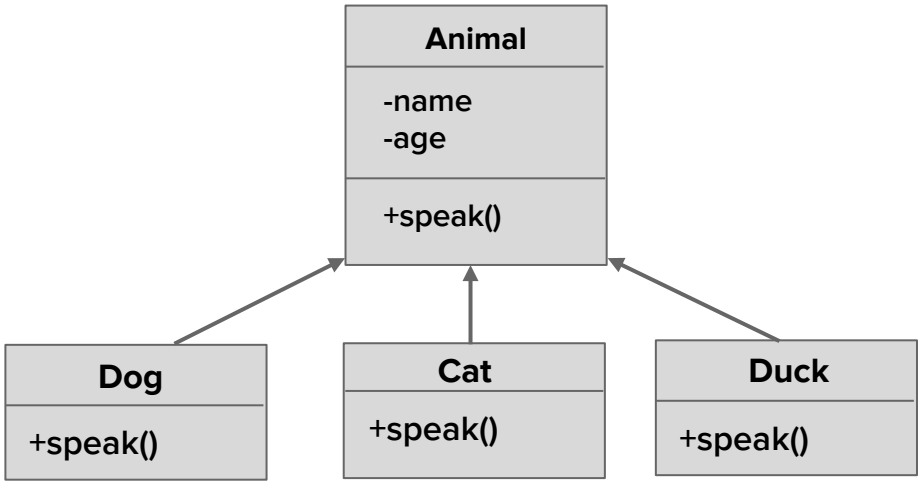
- **Polymorphism**
  - Method overriding in Inheritance
    - Zoo Example
  - **Abstraction**
    - Abstract Class
    - Abstract Method
    - Examples:
      - Zoo Example
      - Company Payroll Example
  - **Interface**
    - Interface in real life examples
    - Upgrade Company Payroll with Invoices Example
  - Abstract vs Interface vs Composition

Polymorphism

- Polymorphism literally means **“many shapes”.**
- The **same method** is called in **different types of objects** has **different results**.
- Enables you to **“program in the general”** rather than **“program in the specific.”**
- 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());

        // 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.printf("%s speaks %s, belongs to %s.\n",
                    myZoo[i].getName(), myZoo[i].speak(), myZoo[i].getClass());
        }
    }
}
```

Output:

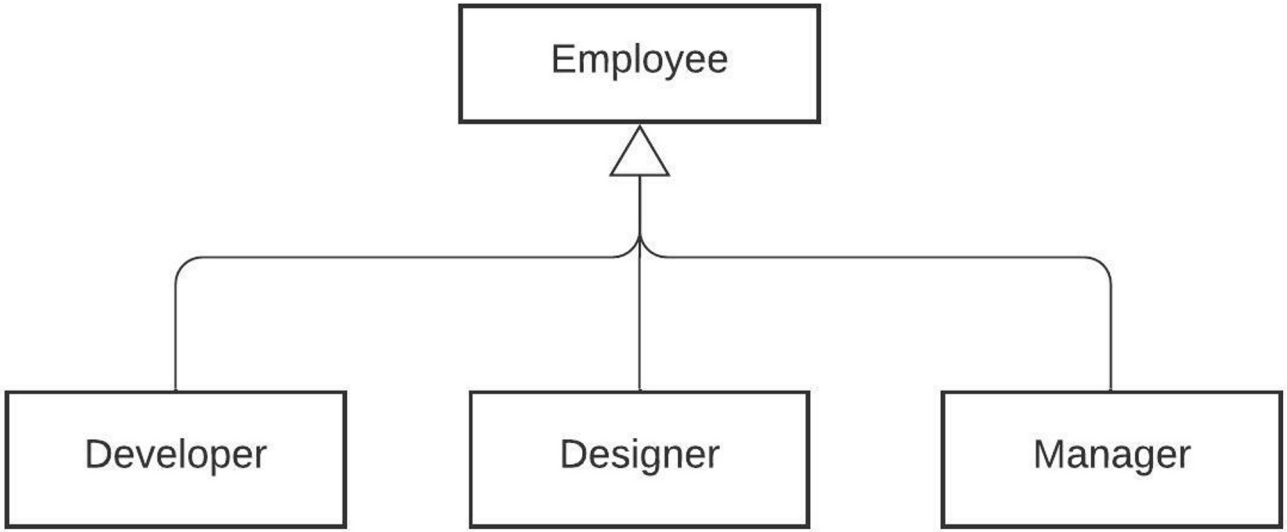
```
Treat them as their own subclass:
Dog speaks Woof
Duck speaks Quack
Cat speaks Meow

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

Group them into an collection of Animal type:
Corgi speaks Woof, belongs to class Dog.
Daisy speaks Quack, belongs to class Duck.
Garfield speaks Meow, belongs to class Cat.
```

Company Payroll Example

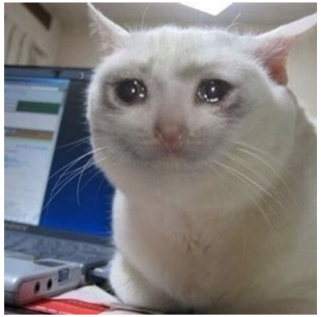
- A company has many employees. Each employee can be:
  - **Developer**
  - **Designer**
  - **Manager**


- 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.

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;

    public Developer(String name, double baseSalary,
                     int numberProjects, double bonusPerProject) {
        super(name, baseSalary);
        this.numberProjects = numberProjects;
        this.bonusPerProject = bonusPerProject;
    }

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

    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; }
}
```

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

Class Designer

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

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

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

    public boolean isBonus13thMonth() { return bonus13thMonth; }

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

[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);
        this.numTeamMembers = numTeamMembers;
        this.bonusPerTeamMember = bonusPerTeamMember;
    }

    @Override
    public double earning(){
        return this.getBaseSalary() + getNumTeamMembers()*getBonusPerTeamMember();
    }

    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; }
}
```

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

Class Company for Testing

```java
import java.util.ArrayList;
public class Company {
    public static void main(String[] args) {
        // Of course, a Designer is a Designer
        Designer trangDesginer = new Designer("Trang", 1000, true);
        System.out.printf("%s earns as a Designer $%.2f\n",
                trangDesginer.getName() ,trangDesginer.earning());

        // Surprisingly, a Designer is also a Employee
        Employee trangEmployee = new Designer("Trang", 1000, true);
        System.out.printf("%s earns as an Employee $%.2f\n",
                trangEmployee.getName() ,trangEmployee.earning());

        // By treating everyone as an Employee, you can manage them like so!
        // Java will figure out at the runtime which subclass each Employee is and
        // use the correct method for it!
        ArrayList<Employee> employeesGroup = new ArrayList<>();
        employeesGroup.add(new Designer("Trang", 1000, true));
        employeesGroup.add(new Developer("Tom", 2000, 4, 400));
        employeesGroup.add(new Developer("Jerry", 1500, 2, 400));
        employeesGroup.add(new Manager("Charles", 2000, 10, 50));
        System.out.println("\nThe whole company earning report:");
        for (Employee e:employeesGroup) {
            System.out.printf("%s earns $%.2f\n",e.getName() ,e.earning());
        }
    }
}
```

Three ways to assign **superclass** and **subclass** references to **variables** of **superclass** and **subclass** types:

1. Assigning a **superclass** reference to a **superclass** variable is **straightforward.**
2. Assigning a **subclass** reference to a **subclass** variable is **straightforward.**
3. Assigning a **subclass** reference to a **superclass** variable is **safe**, because the **subclass object is an object of its superclass.** However, **only** the **superclass** variable can be used to refer to **superclass** members.

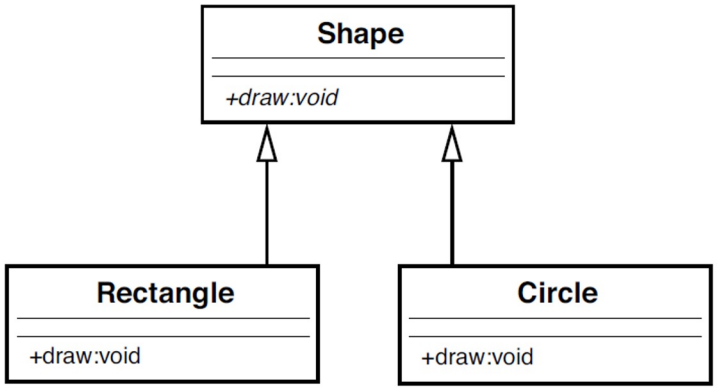
Output:

```
Trang earns as a Designer $1083.33
Trang earns as an Employee $1083.33

The whole company earning report:
Trang earns $1083.33
Tom earns $3600.00
Jerry earns $2300.00
Charles earns $2500.00
```

Abstraction

Abstraction

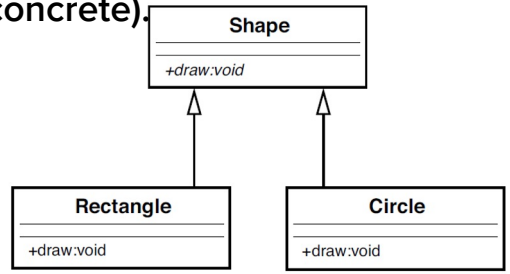
- An **abstract class** is a **class** that **contains** one or more **methods** that **do not have any implementation provided**.
- For example that you have an abstract class called Shape . It is **abstract** because you **cannot instantiate (create)** it.
  - If you ask someone to draw a shape, the 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, Cat or Lecturer.
- **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)

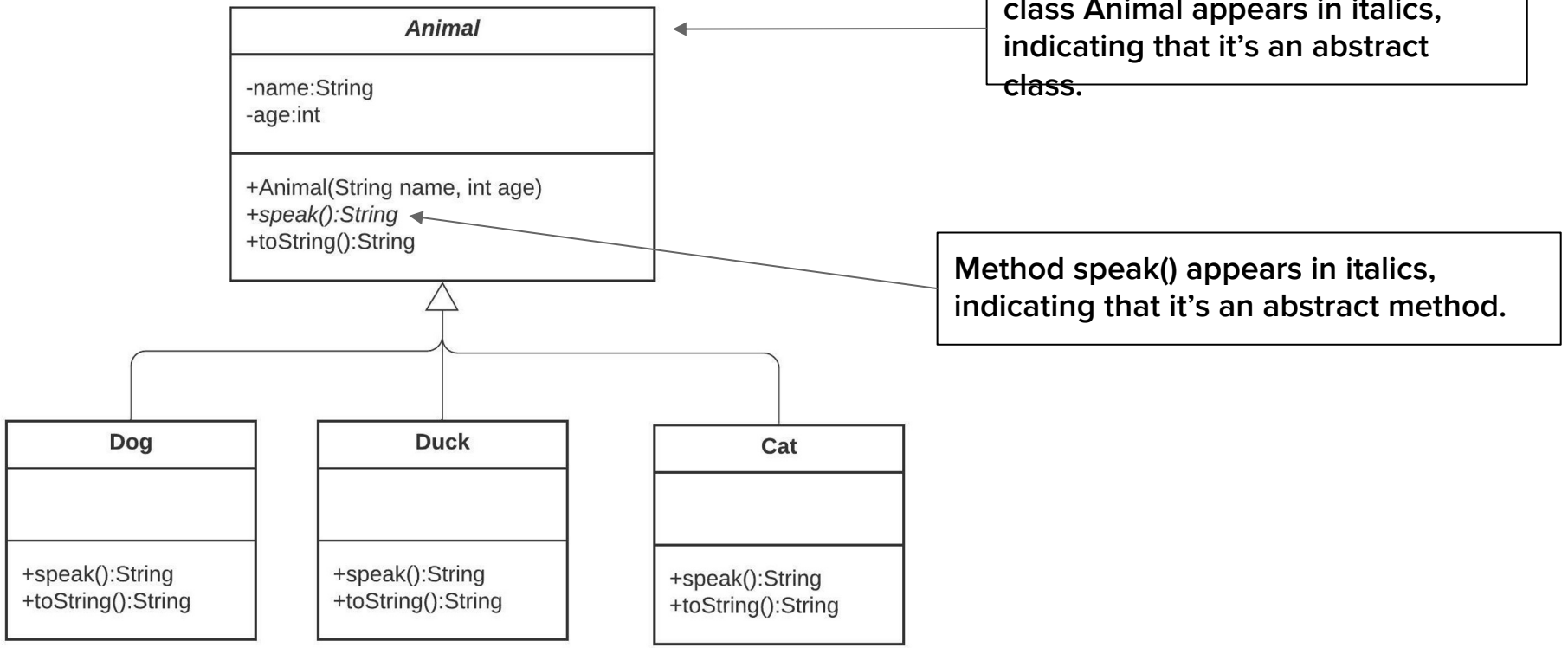
Class Zoo with main method for Testing

```
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**.

Abstract Class Employee

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

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

    // abstract method earning()
    // leave the specific implementation to subclass to handle
    public abstract double earning();

    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] the **class Employee contains** one **abstract method earning()** that is why class Employee is an **abstract class.**
Also, because **method earning() is abstract**, we **don't need to provide any implementation** and **leave that responsibility to other subclasses to complete that abstract method.**

Class Developer

```java
public class Developer extends Employee{
    private int numberProjects;
    private double bonusPerProject;

    public Developer(String name, double baseSalary,
                     int numberProjects, double bonusPerProject) {
        super(name, baseSalary);
        this.numberProjects = numberProjects;
        this.bonusPerProject = bonusPerProject;
    }

    // override the abstract method earning of the abstract class Employee
    // that makes class Developer become a concrete class
    @Override
    public double earning(){
        return this.getBaseSalary() + numberProjects*bonusPerProject;
    }

    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; }
}
```

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

Class Designer

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

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

    // override the abstract method earning of the abstract class Employee
    // that makes class Manager become a concrete class
    @Override
    public double earning(){
        if (bonus13thMonth==true){
            return (this.getBaseSalary()/12)*13;
        } else{
            return this.getBaseSalary();
        }
    }

    public boolean isBonus13thMonth() { return bonus13thMonth; }

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

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

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);
        this.numTeamMembers = numTeamMembers;
        this.bonusPerTeamMember = bonusPerTeamMember;
    }

    // override the abstract method earning of the abstract class Employee
    // that makes class Manager become a concrete class
    @Override
    public double earning(){
        return this.getBaseSalary() + getNumTeamMembers()*getBonusPerTeamMember();
    }

    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; }
}
```

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

Class Company with main method for Testing

```java
import java.util.ArrayList;
public class Company {
    public static void main(String[] args) {
        // No way to initial the Employee object as class Employee is abstract
        // Employee myEmployee = new Employee("Sang", 2000);

        // Of course, a Designer is a Designer
        Designer trangDesginer = new Designer("Trang", 1000, true);
        System.out.printf("%s earns as a Designer $%.2f\n",
                trangDesginer.getName() ,trangDesginer.earning());

        // Surprisingly, a Designer is also a Employee
        Employee trangEmployee = new Designer("Trang", 1000, true);
        System.out.printf("%s earns as an Employee $%.2f\n",
                trangEmployee.getName() ,trangEmployee.earning());

        // By treating everyone as an Employee, you can manage them like so!
        // Java will figure out at the runtime which subclass each Employee is and
        // use the correct method for it!
        ArrayList<Employee> employeesGroup = new ArrayList<>();
        employeesGroup.add(new Designer("Trang", 1000, true));
        employeesGroup.add(new Developer("Tom", 2000, 4, 400));
        employeesGroup.add(new Developer("Jerry", 1500, 2, 400));
        employeesGroup.add(new Manager("Charles", 2000, 10, 50));
        System.out.println("\nThe whole company earning report:");
        for (Employee e:employeesGroup) {
            System.out.printf("%s earns $%.2f\n",e.getName() ,e.earning());
        }
    }
}
```

Output:

```
Trang earns as a Designer $1083.33
Trang earns as an Employee $1083.33

The whole company earning report:
Trang earns $1083.33
Tom earns $3600.00
Jerry earns $2300.00
Charles earns $2500.00
```

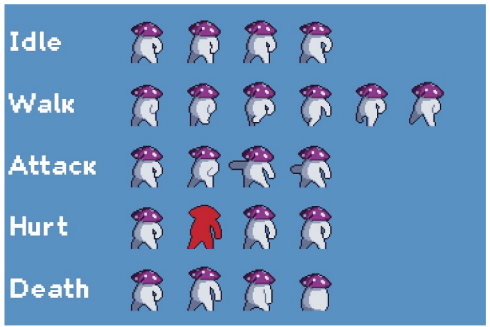
Polymorphism in Video Games

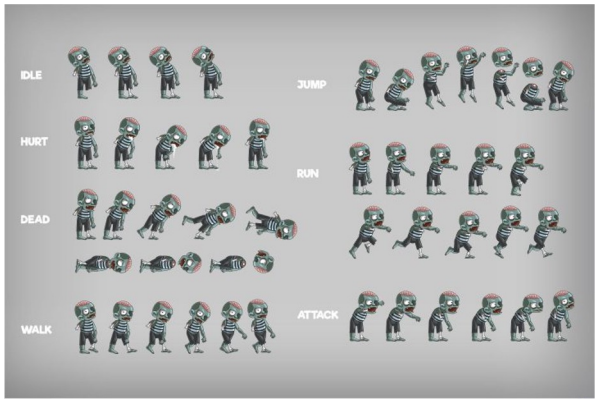
- Example: Game Objects in a Video Game
  - A video game manipulates objects of classes **Zombie**, **SpaceShip** and **FlyingMonster**. Each inherits from an abstract class **GameObject** and overrides its **method draw().**
  - A screen manager class maintains **a collection of references to objects of the various classes** and periodically **sends each object the same message**—draw()

    Each object responds in a unique way:
    - A **Zombie** object might draw itself in green skin and ugly appearance.
    - A **SpaceShip** object might draw itself as a bright silver flying ship.
    - A **FlyingMonster** object might draw itself as a cute flying pig monster across the screen.
  - The same message (in this case, draw) sent to a variety of objects has “many forms” of results.

Polymorphism & Abstraction in Video Games

- A screen manager class might **use polymorphism** to facilitate **adding new classes to a system with minimal modifications to the system's code.**
- To add new objects to our video game:
  - Build a **subclass** that **extends abstract class GameObject** and **provides its own draw method implementation.**
  - When objects of that class appear in the **GameObject** collection, the screen manager code invokes method draw, exactly as it does for every other object in the collection, regardless of its type.
  - So the new objects simply "plug right in" without any modification of the screen manager code by the programmer.



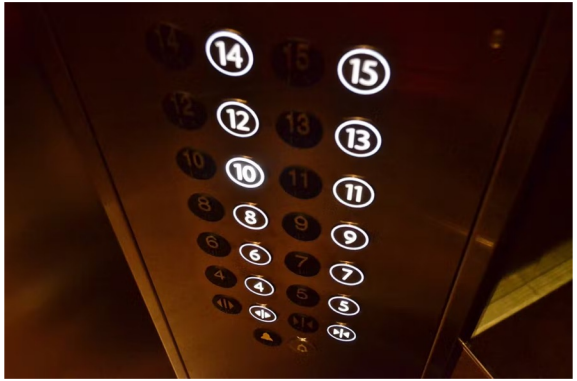


Interface

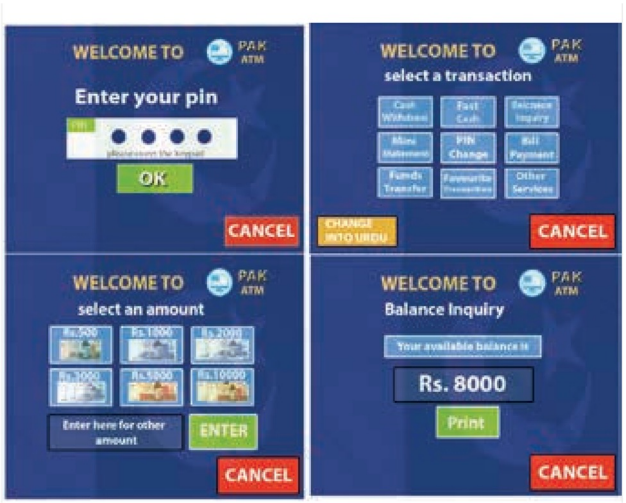
Abstraction on steroid

Interface in real life (Car)

Interface in real life (Elevator)



Interface in real life (ATM)



Interface in real life (Piano)

Interface

- Interface offers a capability requiring that unrelated classes implement a set of common methods.
- Standardizing Interactions: Interfaces define and standardize the ways in which things such as people and systems can interact with one another
- Interfaces define a set of methods, attributes that any class that uses the interface are required to provide.
- You can implement multiple interfaces at the same time. You can also implement interfaces and inherit from a superclass at the same time.

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);
}
```

Option 1: A class which implements an interface needs to implement all required methods of the interface to be a concrete class.

```
public class myClass implements InterfaceName{
  // implement all method of InterfaceName
}
```

Option 2: An abstract class can partly implement some of the required methods of the interface. Then let a concrete class to inherit that abstract class and implement the rest of the required methods.

```
public abstract class myAbstractClass implements InterfaceName{
  // one or more method of InterfaceName are not yet implemented
  // so the class must be abstract
}

public class myConcreteClass extends myAbstractClass{
  // implement the remaining method in InterfaceName
}
```

Revisited the Company Payroll

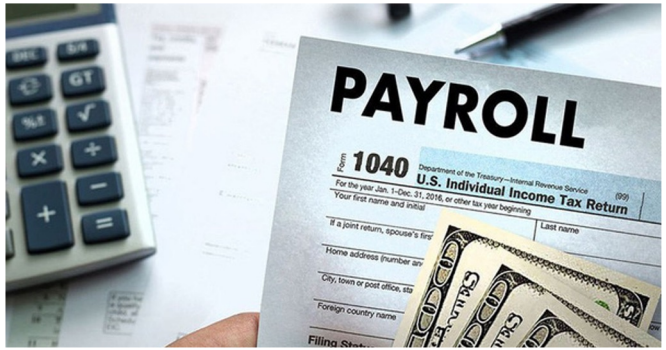
Suppose that the company involved wishes to **perform several accounting operations** in a **single accounts payable application**:

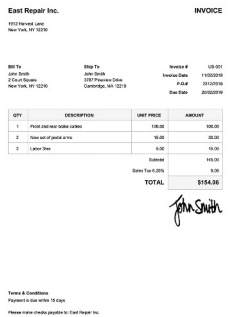
- **Calculate the earnings that must be paid to each employee**,
- **Calculate the payment due on each of several invoices** (i.e., bills for products purchased).

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**.

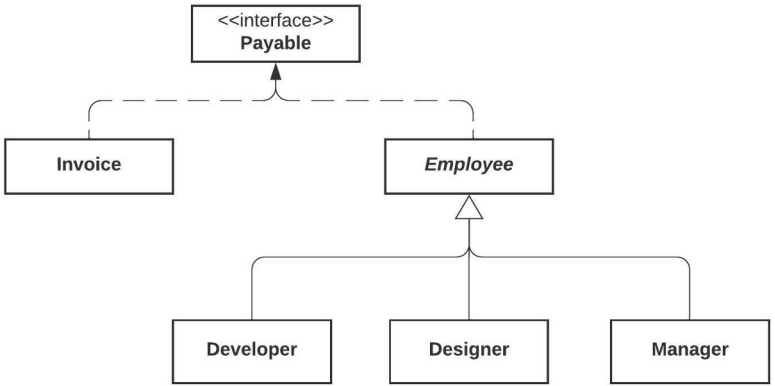
Can we calculate such different things as the payments due for employees and invoices in a single application polymorphically? Does Java offer a capability requiring that unrelated classes implement a set of common methods (e.g., a method that calculates a payment amount)?





Upgrade our Company Payroll with Interface

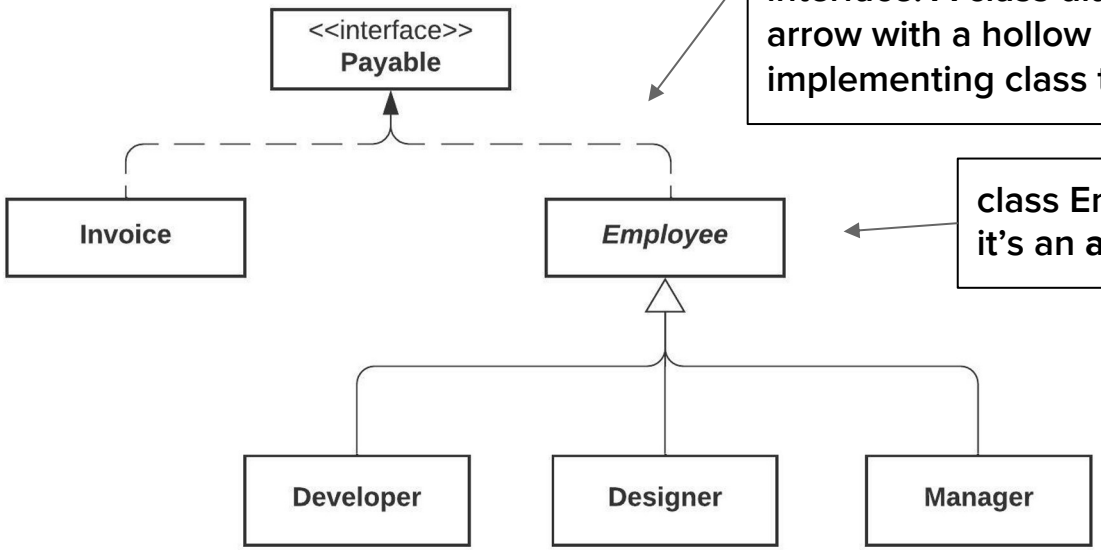
- To build an application that can determine payments for employees and invoices alike, we first **create interface Payable**, which **contains method getPaymentAmount()** that **returns the amount that must be paid for an object of any class that implements the interface.**
- **Method getPaymentAmount()**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

}
```

Class Invoice

[Info] After declaring interface Payable, we introduce class Invoice, which implements, interface Payable.

```java
public class Invoice implements Payable{
    private int quantity;
    private double pricePerItem;
    private String description;

    public Invoice(int quantity, double pricePerItem, String description) {
        this.quantity = quantity;
        this.pricePerItem = pricePerItem;
        this.description = description;
    }

    // method required to actually implement the interface Payable
    @Override
    public double getPaymentAmount() {
        return getQuantity() * getPricePerItem(); // calaculate the cost
    }

    @Override
    public String toString() {
        return "Invoice{" +
                "quantity=" + quantity +
                ", pricePerItem=" + pricePerItem +
                ", description='" + description + '\'' +
                '}';
    }

    public int getQuantity() {
        return quantity;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }

    public double getPricePerItem() {
        return pricePerItem;
    }

    public void setPricePerItem(double pricePerItem) {
        this.pricePerItem = pricePerItem;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }
}
```

Class Employee

[Info] We then modify class Employee such that it also implements interface Payable.

```
// abstract class Employee
public abstract class Employee implements Payable {
    private String name;
    private double baseSalary;

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

    // Note: We do not implement Payable method getPaymentAmount here so
    // this class must be declared abstract to avoid a compilation error.


    @Override
    public String toString() {
        return "Basic Employee Info {" +
                "name='" + name + '\'' +
                ", baseSalary=" + 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] The class Employee implements the interface Payable but it does not make sense for an employee to know how to get paid as there are many types of employees (**Employee is still too general**).
That is why **we don't implement the method getPaymentAmound() yet so the class Employee has to be abstract.**

Class Developer

```java
public class Developer extends Employee{
    private int numberProjects;
    private double bonusPerProject;

    public Developer(String name, double baseSalary,
                     int numberProjects, double bonusPerProject) {
        super(name, baseSalary);
        this.numberProjects = numberProjects;
        this.bonusPerProject = bonusPerProject;
    }

    // calculate earnings; implement interface Payable method
    // that was abstract in superclass Employee
    @Override
    public double getPaymentAmount()
    {
        return this.getBaseSalary() + numberProjects*bonusPerProject;
    }

    @Override
    public String toString() {
        return "Developer{ " + super.toString() +
                ", numberProjects=" + numberProjects +
                ", bonusPerProject=" + bonusPerProject +
                '}';
    }

    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;
    }
}
```

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

Class Designer

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

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


    // calculate earnings; implement interface Payable method
    // that was abstract in superclass Employee
    @Override
    public double getPaymentAmount()
    {
        if (isBonus13thMonth()==true){
            return (this.getBaseSalary()/12)*13;
        } else{
            return this.getBaseSalary();
        }
    }

    @Override
    public String toString() {
        return "Designer{" + super.toString() +
                ", bonus13thMonth=" + isBonus13thMonth() +
                '}';
    }

    public boolean isBonus13thMonth() {
        return bonus13thMonth;
    }

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

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

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);
        this.numTeamMembers = numTeamMembers;
        this.bonusPerTeamMember = bonusPerTeamMember;
    }

    // calculate earnings; implement interface Payable method
    // that was abstract in superclass Employee
    @Override
    public double getPaymentAmount()
    {
        return this.getBaseSalary() + getNumTeamMembers()*getBonusPerTeamMember();
    }

    @Override
    public String toString() {
        return "Manager{ " + super.toString() +
                ", numTeamMembers=" + numTeamMembers +
                ", bonusPerTeamMember=" + bonusPerTeamMember +
                '}';
    }

    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;
    }
}
```

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

Class CompanyWithInterfaceTesting

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

        // create five-element Payable array
        Payable[] payableObjects = new Payable[5];

        // populate array with objects that implement Payable
        payableObjects[0] = new Invoice(4, 375.00 , "New macbooks");
        payableObjects[1] = new Invoice(1, 79.95, "Electrical bill");
        payableObjects[2] = new Designer("Trang", 1000, true);
        payableObjects[3] = new Developer("Tom", 2000, 4, 400);
        payableObjects[4] = new Manager("Charles", 2000, 10, 50);

        System.out.println("Invoices and Employees processed polymorphically:");

        // generically process each element in array payableObjects
        for (Payable currentPayable : payableObjects) {
            // output currentPayable and its appropriate payment amount
            System.out.printf("\n%s \n%s: $%,.2f\n",
                    currentPayable.toString(),        // could invoke implicitly
                    "payment due", currentPayable.getPaymentAmount());
        }
    } // end main
}  // end class CompanyWithInterfaceTesting
```

Output:

```
Invoices and Employees processed polymorphically:

Invoice{quantity=4, pricePerItem=375.0, description='New macbooks'}
payment due: $1,500.00

Invoice{quantity=1, pricePerItem=79.95, description='Electrical bill'}
payment due: $79.95

Designer{Basic Employee Info {name='Trang', baseSalary=1000.0}, bonus13thMonth=true}
payment due: $1,083.33

Developer{ Basic Employee Info {name='Tom', baseSalary=2000.0}, numberProjects=4, bonusPerProject=400.0}
payment due: $3,600.00

Manager{ Basic Employee Info {name='Charles', baseSalary=2000.0}, numTeamMembers=10, bonusPerTeamMember=50.0}
payment due: $2,500.00
```

Class CompanyWithInterfaceTesting illustrates that interface Payable can be used to process a set of Invoices and Employees polymorphically in a single application.

Abstract VS Interface vs Composition

Abstract VS Interface vs Composition?

- The obvious question is this: **If an abstract class can provide the same functionality as an interface, why do Java bother to provide an interface**?
- If both **abstract classes and interfaces provide abstract methods, what is the real difference between the two?**
- As we saw before, **an abstract class provides both abstract and concrete methods, whereas an interface provides only abstract methods. Why is there such a difference?**

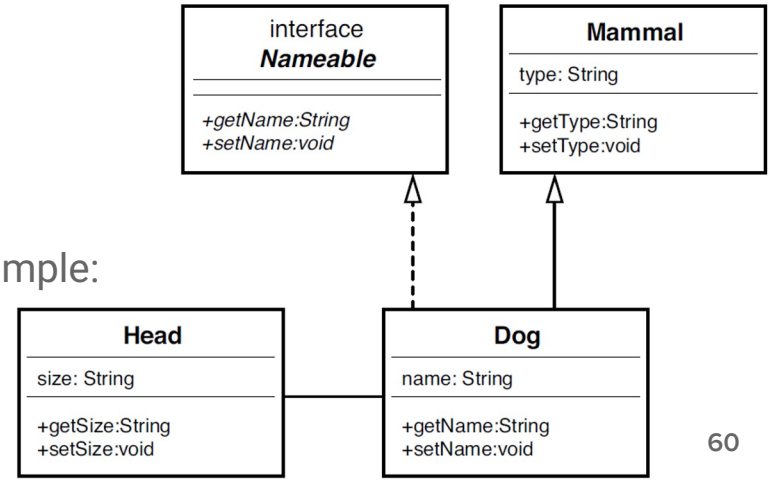
- In a nutshell, Java **build objects in three ways**: **inheritance**, **interfaces**, and **composition**.
- This example illustrates when you should use which one. When do you choose an abstract class? When do you choose an interface? When do you choose composition?
- You should be familiar with the following concepts:
  - Dog is a Mammal , so the relationship is **inheritance**.
  - Dog implements Nameable , so the relationship is an **interface**.
  - Dog has a Head , so the relationship is **composition**.

Although **inheritance** is a **strict is-a relationship**, an **interface** is **not** quite. For example:

- A dog is a mammal.
- A toy is not a mammal.

Thus, a Toy class could not inherit from the Mammal class.
However, an interface applies for the various classes. For example:

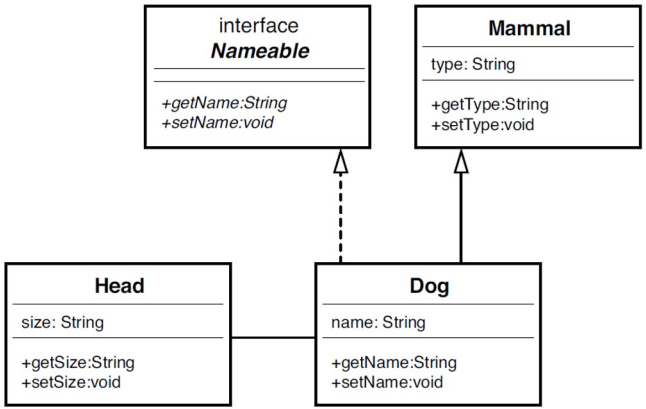
- A dog is nameable.
- A teddy bear is nameable.



```
public abstract class Mammal {

    public void generateHeat() {System.out.println("Generate heat");}

    public abstract void makeNoise();

}
```

```
public class Head {

    String size;

    public String getSize() {

        return size;

    }

    public void setSize(String aSize) { size = aSize;}

}
```

```
public interface Nameable {

    String getName();
    void setName (String aName);

}
```




```
public class Dog extends Mammal implements Nameable {

    String name;

    Head head;

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

    public void setName (String aName) {name = aName;}
    public String getName () {return (name);}

}
```

Interfaces vs. Abstract Classes

- Consider using **abstract classes** if any of these statements apply to your situation:
  - You want to **share code among several closely related classes**.
  - You want classes that inherit your abstract class have many common methods or attributes.
  - You want to declare **final, non-final, static and non-static variables.**
  - You want to declare **both abstract and non-abstract methods.**
  - You **only need the subclass to only inherit (extends) from one class.**
- Consider using **interfaces** if any of these statements apply to your situation:
  - You expect that **unrelated classes would implement your interface.**
  - You want to **specify the behavior of a particular data type**, but **not concerned about which class implements its behavior.**
  - You want to declare **only static and final variables.**
  - You want to declare **only abstract methods.**
  - You want to **take advantage of multiple interface**.
  - Your problem makes the statement **"A is capable of [doing this]".**
    - "Clonable is capable of cloning an object"
    - "Drawable is capable of drawing a shape",
    - "Payable is capable of returning a payment amount".

Recap

- **Polymorphism**
  - Method overriding in Inheritance
    - Zoo Example
  - **Abstraction**
    - Abstract Class
    - Abstract Method
    - Examples:
      - Zoo Example
      - Company Payroll Example
  - **Interface**
    - Interface in real life examples
    - Upgrade Company Payroll with Invoices Example
  - Abstract vs Interface vs Composition

Thank you for your listening!

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

Jim Ryun