School of Information Systems

SMU SINGAPORE MANAGEMENT UNIVERSITY

Object Oriented Application Development

Inheritance and Packaging

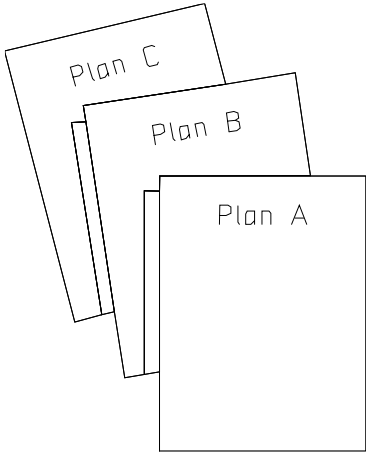
- Part I -

Ability is what you are capable of doing.
Motivation determines what you do.
Attitude determines how well you do it.

Lou Holtz

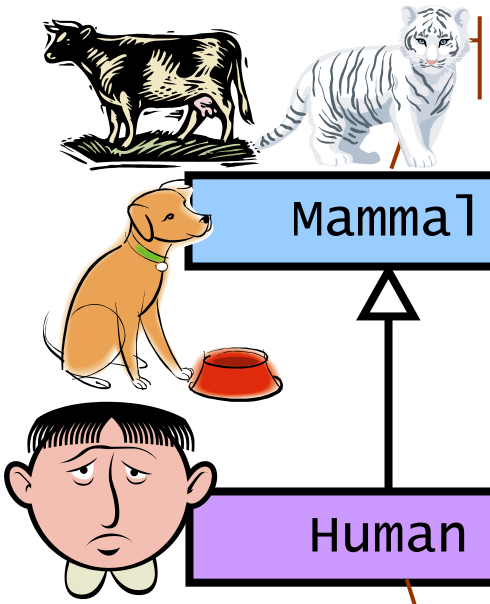
Overview

- Objective
  - To be able to use inheritance in object oriented design and implementation
- Content
  - Inheritance
  - Abstract class
  - Interface
  - Polymorphism
- After this lecture, you should be able to
  - Apply inheritance & polymorphism in your coding



Inheritance

- Organizes objects in a top-down fashion from the most general to the least general.
- *is-a* relationship



superclass/base class

A Human is a Mammal.

All characteristics of Mammal are characteristics of a Human.

subclass/derived class

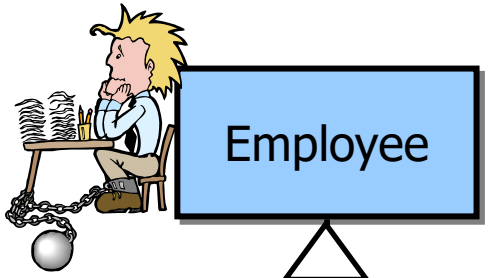
```
public class Mammal {
// ...
}
```

```
public class Human extends Mammal {
  // ...
}
```

Keyword **extends** indicates that Human is a sub-class of Mammal

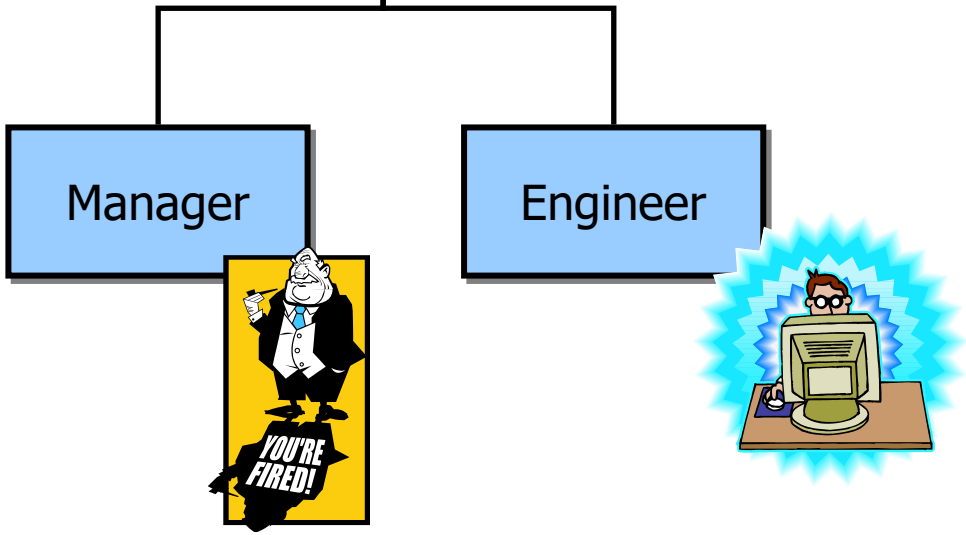
Benefit of Inheritance

- Derived class inherits properties and methods from the base class
- Primary benefit of inheritance is code reuse

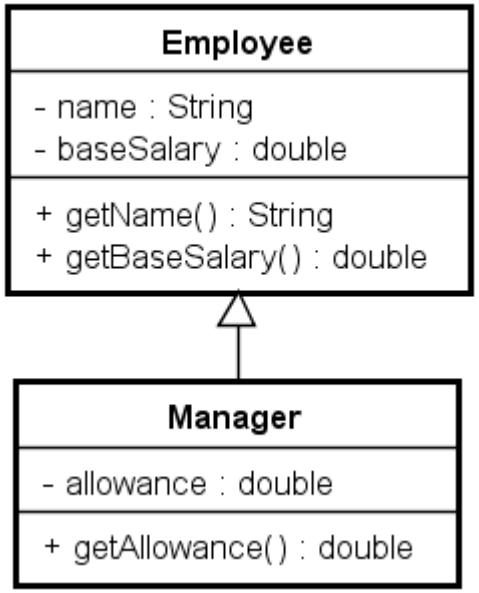


Common characteristics are defined once at the base class

Look for common behaviours



Reusing Code




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

    public String getName() {
        return name;
    }
    public double getBaseSalary(){
       return baseSalary;
    }
}
```

```
public class Manager extends Employee {
    private double allowance;

    public double getAllowance(){
       return allowance;
    }
}
```

Inheritance: Constructors

- Constructors are not inherited by subclasses
- The constructor of the superclass can be invoked from the subclass

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

    public Employee(String name,
            double baseSalary) {
        // …
    }
}
```

```
public class Manager Test {
  public static void main(String[] args) {
    // not possible
    Manager m = new Manager("Amy", 1200);

  }
}
```

```
public class Manager extends Employee {
    private double allowance;

    public Manager(String name,
            double baseSalary,
            double allowance) {
        // ...
    }

}
```

Using super in constructors

- The super keyword is used to call a parent's constructor
- The super statement must be the first statement in the constructor

```
public class Manager extends Employee {
    private double allowance;

    public Manager(String name,
            double baseSalary,
            double allowance) {
        super (name, baseSalary);
        this.allowance = allowance;
    }

}
```

```
public class Manager extends Employee {
    private double allowance;

    public Manager(String name,
            double baseSalary,
            double allowance) {
        this.allowance = allowance;
        super (name, baseSalary);
    }

}
```

Using super in constructors

- If you did not provide the super statement, Java will insert a default call
  - It will invoke the no-arg constructor in the super class. In other words, super()

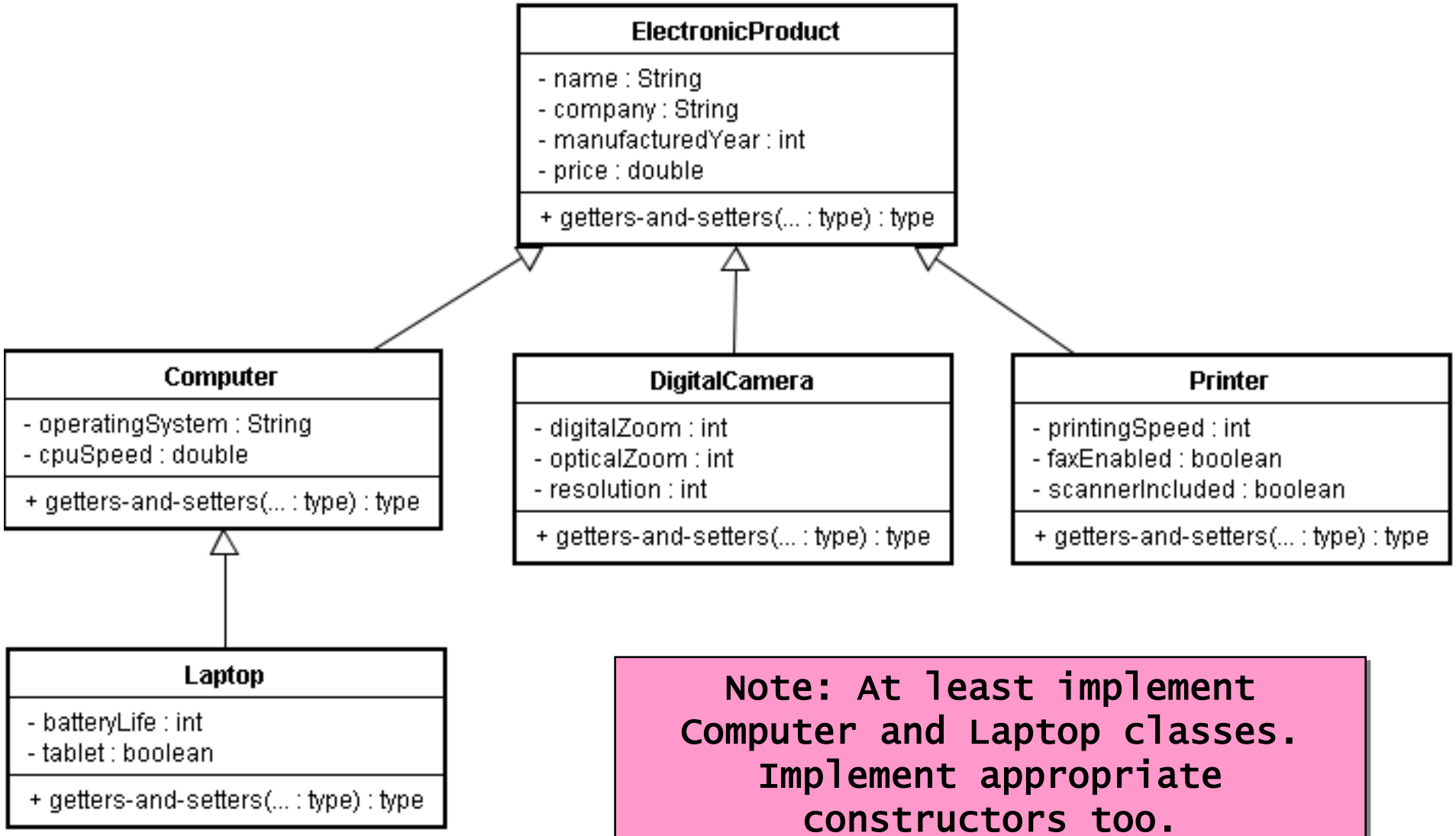
```
public class Cat extends Animal {

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

}
```

```
// added in by Java during compilation
super();
```

Exercise 1: Inheritance and Reuse

- You are part of a team tasked to build a new system to manage electronic products in a large electronic center
- At the moment the center only sells a few types of items:
  - Computers
  - Digital Cameras
  - Printers
- Your job is to implement the entity classes
- Implement the classes shown in the next slide

Exercise 1: Inheritance and Reuse



Note: At least implement Computer and Laptop classes. Implement appropriate constructors too.

Exercise 2: Inheritance and Reuse

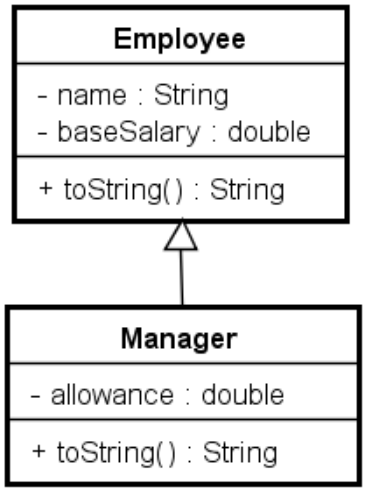
- Implement getNewWindowsBasedLongBatteryLifeTablet method of LaptopSearcher
  - Input: A list of laptops
  - Output: Laptops satisfying the following criteria:
    - Manufactured after 2008
    - Run on Windows operation system
    - Battery life more than 5 hours
    - Tablet PC
- Main method is provided

```
- Output -

T1001
T1003
```

LaptopSearcher
+ getNewWindowsBasedLongBatteryLifeTablet(laptops:ArrayList<Laptop>):ArrayList<Laptop>

Overriding

- An instance method in a subclass with the same signature (name, plus the number and the type of its parameters) and return type as an instance method in the subclass *overrides* the superclass's method.



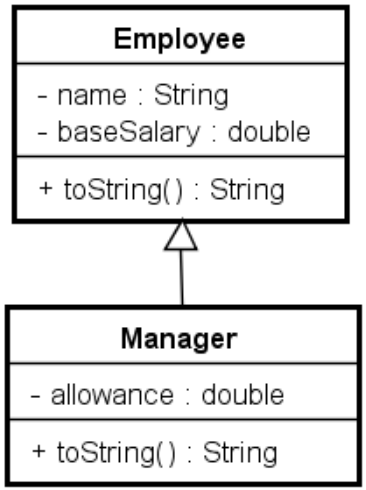

```
public class Employee {
    private String name;
    private double baseSalary;

    public String toString() {
        return "name=" + name
                + ",baseSalary="
                + baseSalary;
    }

}
```

```
public class Manager extends Employee {
    private double allowance;

    public String toString() {
        return super.toString()
            + ",allowance=" + allowance;
    }
}
```

The method toString overrides the method in Employee

Overriding

```
public class ManagerTest {
    public static void main(String[] args) {
        Manager m = new Manager("Lily", 1000, 100);
        String desc = m.toString();
        System.out.println(desc);
    }
}
```



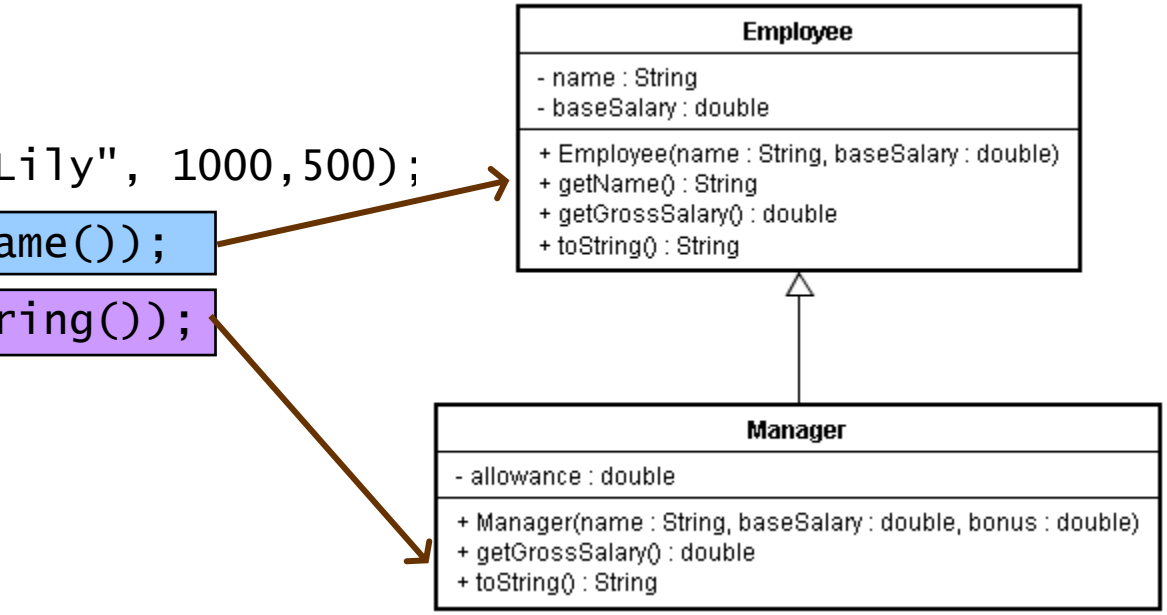

```
- Output -

name=Lily,baseSalary=1000.0,allowance=100.0
```

The method toString in Manager overrides the method in Employee

Which method to invoke

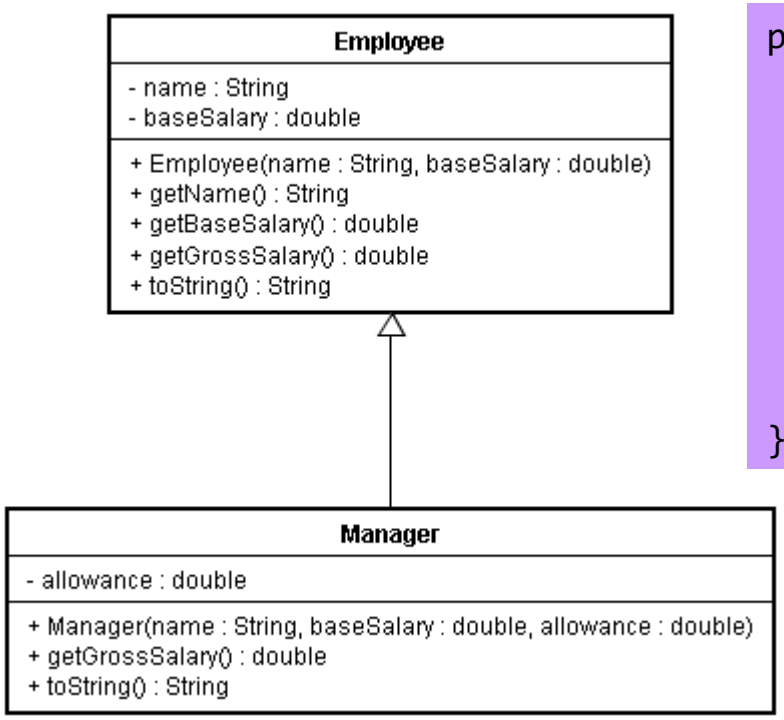
- The lowest one wins!

```
Manager m = new Manager("Lily", 1000,500);
System.out.println(m.getName());
System.out.println(m.tostring());
```



Exercise 3: Method Overriding

- Implement the `getGrossSalary` method in both the `Employee` and `Manager` classes.
- The formula to calculate the Manager's gross salary is `baseSalary + allowance`




```
public class EmployeeTest {
  public static void main(String[] args) {
    Manager m = new Manager("Peter", 5000, 2000);
    Employee e = new Employee("John", 3000);
    System.out.println("Manager's salary : "
        + m.getGrossSalary());
    System.out.println("Employee's salary : "
        + e.getGrossSalary());
  }
}
```

```
- Output -
Manager's salary : 7000.0
Employee's salary : 3000.0
```

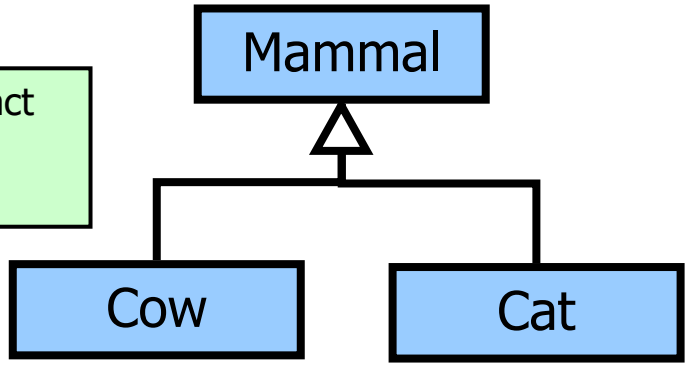
Abstract Class

- A generic base class
- Provides a partial implementation, leaving it to subclasses to complete the implementation

```
public abstract class Mammal {
  private String name;
  public Mammal(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }

  public abstract void makeNoise();

}
```

Classes with abstract method must be declared abstract

Abstract method has no implementation(body). It is overridden by subclasses.



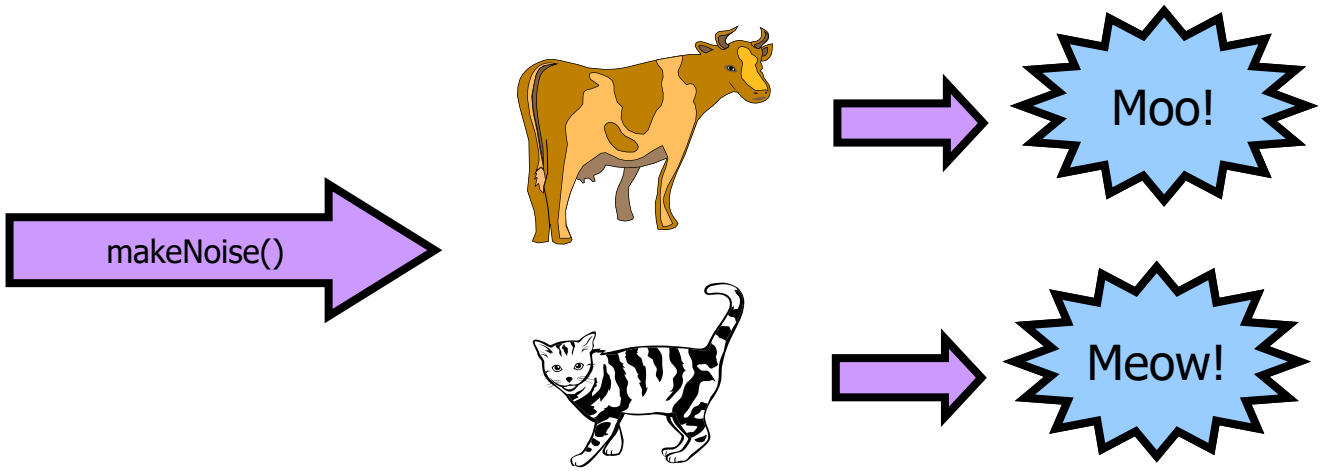
How to Create a Subclass?

Use the **extends** keyword followed by the superclass's name

```
public class Cow extends Mammal {
  public Cow(String name) {
    super(name);
  }

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

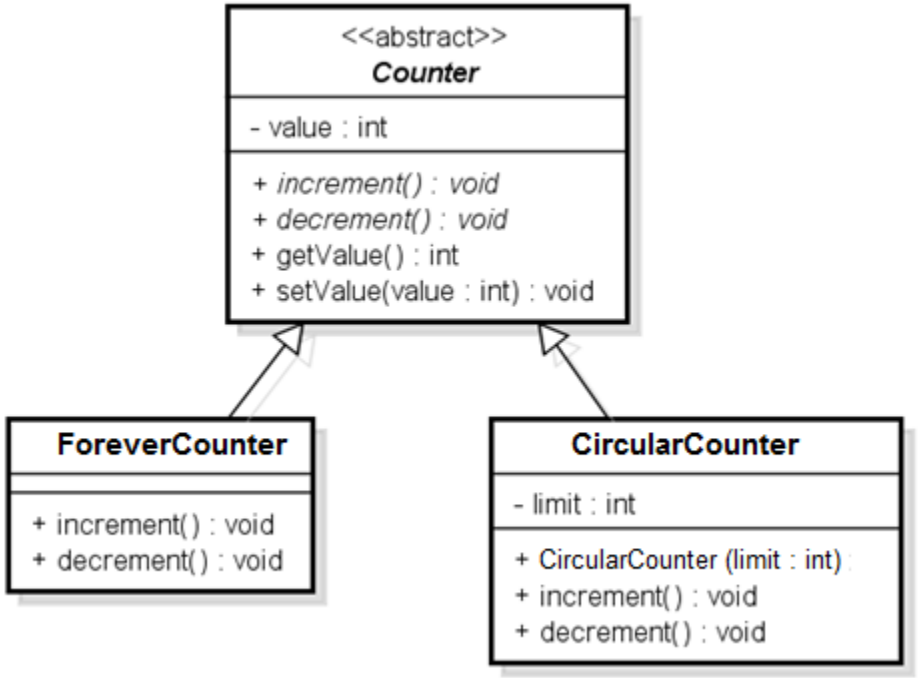
All abstract methods must be implemented in the subclass

```
public class Cat extends Mammal {
  public Cat(String name) {
    super(name);
  }

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



Exercise 4: Abstract Class

- Implement the following classes:
  - `Counter` is an abstract class (`increment`, `decrement` are abstract methods)
  - `ForeverCounter` will count from 0,1,2 ….
  - `CircularCounter` will count from 0,1,2, …limit,0,1,2 ..limit
- `CounterTest.java` is provided



Exercise 4: Abstract Class

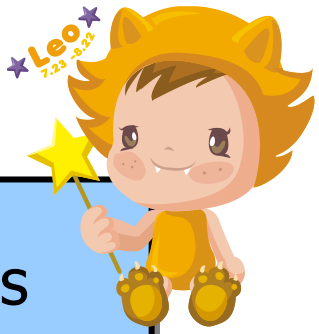
```
                    -Output -


Forever: 0, Circular: 0
Incrementing ...
Forever: 1, Circular: 1
Forever: 2, Circular: 2
Forever: 3, Circular: 3
Forever: 4, Circular: 4
Forever: 5, Circular: 5
Forever: 6, Circular: 0
Forever: 7, Circular: 1
Forever: 8, Circular: 2
Forever: 9, Circular: 3
Forever: 10, Circular: 4
Decrementing ...
Forever: 9, Circular: 3
Forever: 8, Circular: 2
Forever: 7, Circular: 1
Forever: 6, Circular: 0
Forever: 5, Circular: 5
Forever: 4, Circular: 4
Forever: 3, Circular: 3
Forever: 2, Circular: 2
Forever: 1, Circular: 1
Forever: 0, Circular: 0
```

Interfaces

- An interface is a group of related methods with empty bodies.

A template is specified using the **interface** keyword.

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



class

Complete Implementation

abstract class

Partial Implementation (Some methods have no body)

Interface

No implementation (All methods have no body)

Interfaces

- The subclass must provide implementations of the methods specified in the interface.

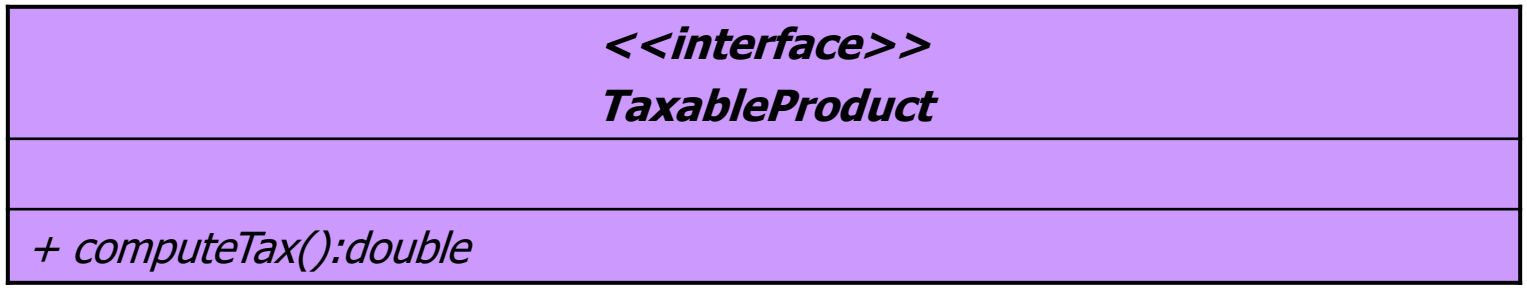
```
public class Cat extends Mammal implements Pet {
  public Cat(String name) {
    super(name);
  }

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

  public void tame() {
    System.out.println("Scratch behind the ear!");
  }
}
```

Use the keyword **implements** followed by the interface name.

The tame() method is defined.

Exercise 5a: Interfaces

- Create an interface TaxableProduct



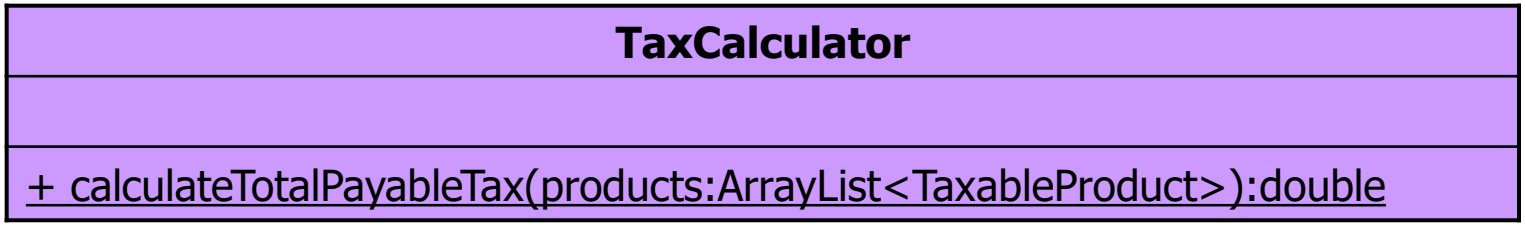
- Modify the classes in Exercise 1 so that we can compute the taxes (tax rate = 7%) for computers, digital cameras, laptops, and printers:
  - Find the best class to implement the interface
  - Implement the interface
- TaxTest.java is provided.

```
-Output -
The tax for a $2500.0 laptop is $175.0
```

Exercise 5b: Using Interfaces

- Create a class TaxCalculator with the static method calculateTotalPayableTax.

- The method accepts a list of TaxableProducts and output the total amount of tax that is to be paid for the products



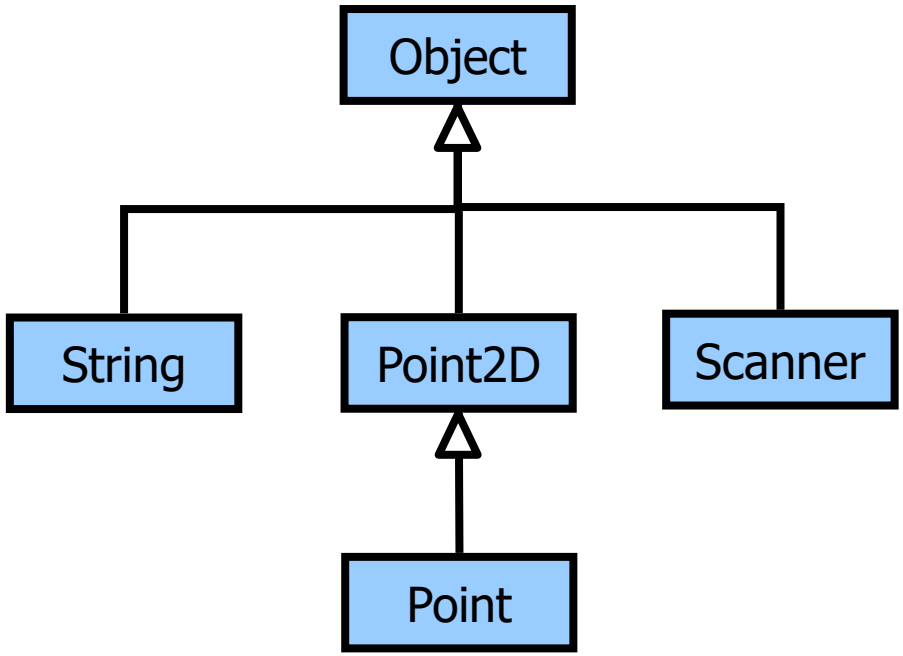
- TaxCalculatorTest.java is provided to test your application.

```
                    -Output -
The total tax is $525.0
```

Inherited Methods & Overriding

Java 5.0 Program Design P334

- All Java class is automatically an extension of the standard class Object.
- The class Object specifies some basic behaviors common to all objects.
  - Examples
    - toString()
    - equals()



The toString() Method

- Provides a meaningful textual representation of the object.
- Can be useful when debugging a program.

```
public class MyPoint {
   private int x;
   private int y;

   public MyPoint(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public String toString() {
        return "MyPoint[" + x + "," + y + "]";
    }
    public static void main(String[] args) {
        MyPoint p = new MyPoint(1,2);
        System.out.println(p);
    }
}
```

- With toString() method -

```
MyPoint[1,2]
```

- Without toString() method -

```
MyPoint@7d772e
```

The instanceof Operator

- The `instanceof` operator is use to check the type of object that the "reference" points to.

```
public class InstanceOfDemo {
  public static void main(String[] args) {
    Manager m = new Manager("Peter", 5000, 2000);
    Employee e = new Employee("John", 3000);
    System.out.println(e instanceof Manager);
    System.out.println(m instanceof Manager);
  }
}
```

```
- Output -
false
true
```

Exercise 6: instanceof

- Complete the calManagerAvgGrossSalary method in the EmployeeTest class.

```
import java.util.ArrayList;

public class EmployeeTest {

  public static double calManagerAvgGrossSalary(ArrayList<Employee> empList) {
    // complete the code
  }

  public static void main(String[] args) {
    ArrayList<Employee> empList = new ArrayList<Employee>();
    empList.add(new Manager("Albert", 5000, 2000));
    empList.add(new Manager("Benny", 7000, 1500));
    empList.add(new Manager("Charles", 9000, 1000));
    empList.add(new Employee("Danny", 1500));
    empList.add(new Employee("Edward", 4000));
    empList.add(new Employee("Fred", 3500));
    empList.add(new Employee("George", 3500));
    System.out.println(calManagerAvgGrossSalary(empList));
  }
}
```

```
- Output -
8500
```

The equals() method

- Determines whether its parameter is a object that is equivalent to the invoking object.

```
public class MyPoint {
  // ...

  public boolean equals(Object obj) {
      if (obj instanceof MyPoint) {
          MyPoint another = (MyPoint)obj;
          if (another.x == x && another.y == y) {
              return true;
          }
      }
      return false;
  }

  public static void main(String[] args) {
    MyPoint p1 = new MyPoint(1,2);
    MyPoint p2 = new MyPoint(1,2);
    System.out.println(p1.equals(p2));
  }
}
```

An instance of MyPoint object is an instance of Object. An instance of Object might not be an instance of MyPoint. Thus, we need to do a cast to tell the compiler than we know indeed it is a MyPoint object.

Returns false unless the object is an object created using the MyPoint class

The equals() Method: Guidelines

- Reflexivity
  - `x.equals(x)` should always be true.
- Symmetry
  - If `x.equals(y)` is true, then `y.equals(x)` should be true.
- Transitivity
  - If `x.equals(y)` and `y.equals(z)` are true, then `x.equals(z)` should be true
- Consistency
  - While the objects to which x and y refer are unchanged, repeated evaluations of `x.equals(y)` should return the same value.
- Physicality
  - `x.equals(null)` should return false

Exercise 7: Inherited Methods

- Write a Box class that represents the sides of a rectangle.
  - Implements a specific constructor that takes in 3 parameters
  - Overrides the equals() and toString() methods in Object.
- BoxTest.java is provided


```
-Output -
b1 == b2
b1 is Length=10.0,Breadth=9.0,Height=8.0
b2 is Length=10.0,Breadth=9.0,Height=8.0
```

Syntactic Polymorphism

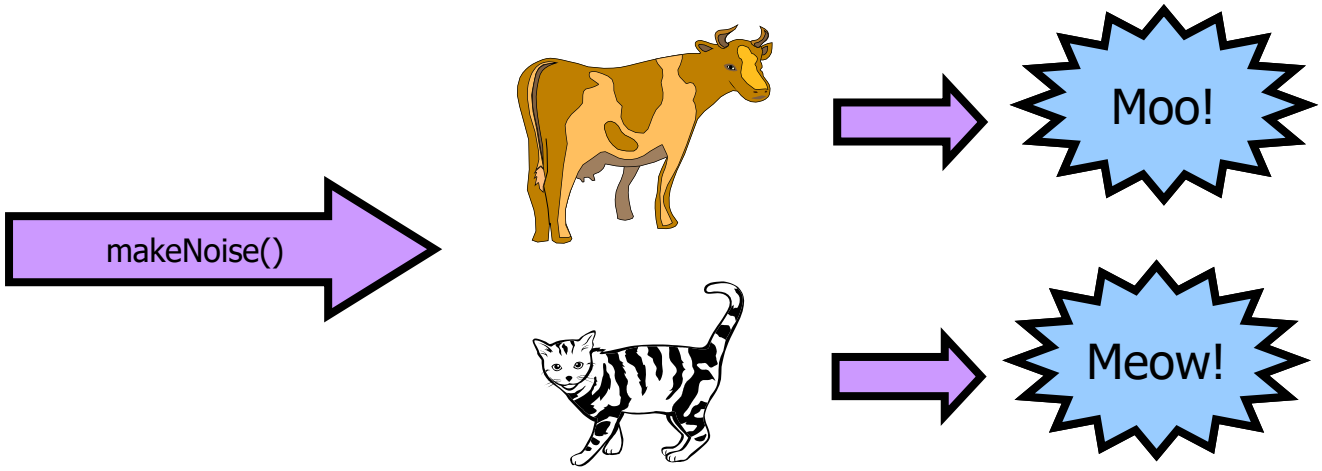
- Java can determine which method to invoke at compile time.
- Example
  - Function overloading like method `Math.min()`.
    - The method invoked depends on the types of the actual arguments.

```
int a, b, c;
double x, y, z;

c = Math.min(a, b);   // invokes integer min()
z = Math.min(x, y);   // invokes double min()
```

Pure Polymorphism

Java 5.0 Program Design P485

- Pure Polymorphism
  - The method to invoke can only be determined at execution time (Late binding).



Exercise 8: Polymorphism

- Create toString() methods for the following classes:
  - ElectronicProduct
  - Computer
  - Laptop
  - DigitalCamera
  - Printer
- The toString() methods should output useful information

Exercise 8: Polymorphism

- Implement createReport method of ProductReportCreator
  - Input: A list of electronic products
  - Output: Information of each product to the console
  - You should make use of the toString() methods
- Create a main method to test the functionality
- Running "java ProductReportCreator > report.txt" should produce the report in report.txt file

ProductReportCreator
+ createReport(products:ArrayList<ElectronicProduct>):void

Useful Interfaces: Comparable

https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html
https://docs.oracle.com/javase/tutorial/collections/interfaces/order.html

- Comparable interface is used to sort a collection of objects of a particular class.

Step 1: Make target class implements **Comparable**
Step 2: Write the method **compareTo**

```
public class Student implements Comparable<Student> {
  private String name;
  private int age;
  // …
  public int compareTo(Student another) {
    return name.compareTo(another.name);
  }
  public String toString() {
    return "( name=" + name + ", age=" + age + " )";
  }
}
```

`int compareTo(T o)`

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

compareTo method of String class: compares two strings lexicographically

Useful Interfaces: Comparable

Step 3: Use Collections.sort

```
import java.util.ArrayList;
import java.util.Collections;

public class StudentTest {


    public static void main(String[] args) {
        ArrayList<Student> sList = new ArrayList<Student>();

        sList.add(new Student("Charlie", 12));
        sList.add(new Student("Amy", 13));
        sList.add(new Student("Billy", 11));
        Collections.sort(sList);

        System.out.println(sList);
    }
}
```

```
- output -
[( name=Amy, age=13 ), ( name=Billy, age=11 ), ( name=Charlie, age=12 )]
```

Useful Interfaces: Comparator

https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html
https://docs.oracle.com/javase/tutorial/collections/interfaces/order.html

- Comparator interface is used to sort a collection of objects of a particular class.

Step 1: Create a **new** class that implements **Comparator**

Step 2: Implement the method **compare**

```
import java.util.*;
public class AgeComparator implements Comparator<Student> {

  public int compare(Student s1, Student s2) {
    return s1.getAge()-s2.getAge();
  }
}
```

```
int compare(T o1,
            T o2)
```

Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

Useful Interfaces: Comparator

Step 3: Use Collections.sort

```
import java.util.ArrayList;
import java.util.Collections;

public class StudentTest {


    public static void main(String[] args) {
        ArrayList<Student> sList = new ArrayList<Student>();

        sList.add(new Student("Charlie", 12));
        sList.add(new Student("Amy", 13));
        sList.add(new Student("Billy", 11));
        Collections.sort(sList, new AgeComparator());

        System.out.println(sList);
    }
}
```

```
- output -
[( name=Billy, age=11 ), ( name=Charlie, age=12 ), ( name=Amy, age=13 )]
```

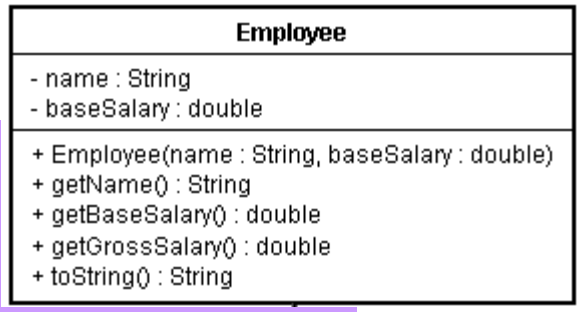
Comparable versus Comparator

- Comparable
  - Pros: The comparison can leverage private fields
  - Cons: Only 1 ordering behavior is possible

- Comparator
  - Pros: Allow us to specify different sorting order
  - Cons: Cannot directly leverage on private fields

Exercise 9: Comparable

- Use the Employee class created for Exercise 3.
- Modify the Employee class such that the following code produces the right output:

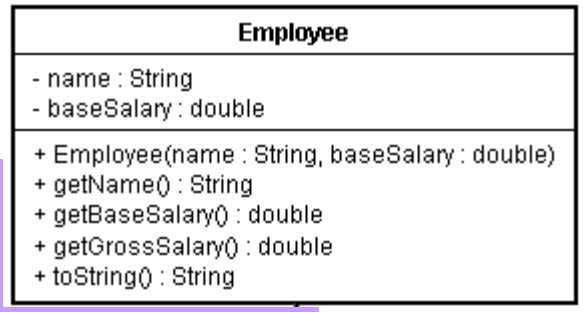



```
import java.util.*;

public class EmployeeSortingTest {
  public static void main(String[] args) {
    ArrayList<Employee> empList = new ArrayList<Employee>();
    empList.add(new Employee ("Peter", 5000));
    empList.add(new Employee("Zack", 3000));
    Collections.sort(empList);
    for (int i=0;i<empList.size();i++){
      System.out.println(empList.get(i));
    }
  }
}
```

```
- Output -
name=Zack,baseSalary=3000.0
name=Peter,baseSalary=5000.0
```

Exercise 10: Comparator - I

- Use the Employee class created for Exercise 3.
- Create **BaseSalaryComparator** class such that the following code produces the right output:

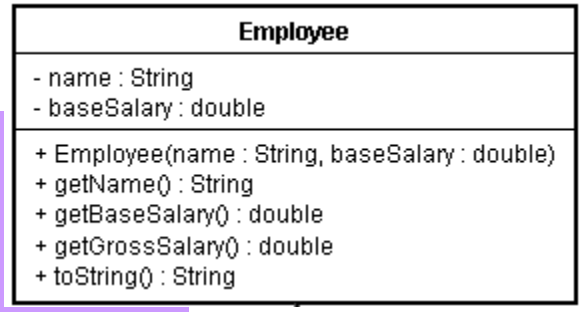



```
import java.util.*;

public class EmployeeSortingTest {
  public static void main(String[] args) {
    ArrayList<Employee> empList = new ArrayList<Employee>();
    empList.add(new Employee ("Peter", 5000));
    empList.add(new Employee("Zack", 3000));
    Collections.sort(empList,new BaseSalaryComparator());
    for (int i=0;i<empList.size();i++){
      System.out.println(empList.get(i));
    }
  }
}
```

```
- Output -
name=Zack,baseSalary=3000.0
name=Peter,baseSalary=5000.0
```

Exercise 11: Comparator - II

- Use the Employee class created for Exercise 3.
- Create **NameComparator** class such that the following code produces the right output:




```
import java.util.*;

public class EmployeeSortingTest {
  public static void main(String[] args) {
    empList.add(new Employee ("Peter", 5000));
    empList.add(new Employee("Zack", 3000));
    Collections.sort(empList,new NameComparator());
    for (int i=0;i<empList.size();i++){
      System.out.println(empList.get(i));
    }
  }
}
```

```
- Output -
name=Zack,baseSalary=3000.0
name=Peter,baseSalary=5000.0
```

Summary

- Inheritance
  - IS-A relationship
  - Code re-use
  - Prefix call with super
- Abstract class
  - Provides a partial implementation, leaving it to subclasses to complete the implementation.
- Interface
  - An interface is a group of related methods with empty bodies.
- Polymorphism
  - Syntatic
    - The method to invoke is determined at compile time.
  - Pure polymorphism
    - The method to invoke can only be determined at execution time.
- Comparable and Comparator