Programación en Java

Tema 3. Programación orientada a objetos en Java (Parte 2)

Luis Rodríguez Baena

Universidad Pontificia de Salamanca (campus Madrid)
Facultad de Informática

Herencia (I)

- Es necesario indicar que la nueva clase (subclase) "es como" la clase de la que procede (superclase) utilizando la palabra `extends`.

```
class EmpleadoEventual extends Empleado{
   final float PRECIO_HORA = 30;
   private int horasTrabajadas = 0;
}
```

- La subclase:
  - Hereda atributos y métodos.
    - ✓ Siempre y cuando tengan los modificadores de accesibilidad adecuados (acceso paquete, `public` o `protected`).
    - ✓ La subclase no tiene ningún miembro con el mismo nombre.
  - Añade atributos y métodos.
  - Modifica atributos y métodos.
- Java no admite herencia múltiple.

Herencia (II)

- Constructores de clases ampliadas.
  - Las subclases pueden emplear los mismos constructores que las superclases.
  - Es posible ampliarlos o crear atributos nuevos.
    - ✓ La instrucción `super(`*`argumentos`*`)` hace una llamada al constructor de la superclase que contenga esos argumentos.

```
EmpleadoEventual(){
        super();
        horasTrabajadas = 1;
}

EmpleadoEventual(long id,String n, int h){
   super(id,n);
   horasTrabajadas = h;
   sueldo = h * PRECIO_HORA;
}
```

Herencia (III)

- Redefinición de métodos.
  - Sobrecarga y redefinición.
  - Redefinición:
    - ✓ Sustituir su implementación.
    - ✓ La llamada al método de la superclase, invoca al método de la subclase.
    - ✓ Desde el método redefinido, se puede llamar al de la superclase con `super().`*`nombreMétodoSuperclase`* y añadir después algún comportamiento.

```
//En la clase Empleado
public String toString(){
    return "Id. Empleado: " + idEmpleado + "\nNombre: " + nombre +
           "\nSueldo: " + sueldo;
}
//En la clase EmpleadoEventual
public String toString(){
     return super.toString() + "\nHoras: " + horasTrabajadas;
}
```

  - No se pueden redefinir métodos `final` ni métodos de clase (`static`).

Herencia (IV)

- La clase Object.
  - Todas las clases Java provienen de la superclase `Object` del paquete `java.lang`.
  - Una variable de tipo `Object` puede utilizarse para referenciar cualquier clase Java.
  - Algunos métodos heredados de la clase `Object`.
    - ✓ `public boolean equals(Object obj).`
    - ✓ `public String toString().`
    - ✓ `public void finalize() thows Throwable.`
  - Algunos de estos métodos se pueden y/o se deben sobreescribir para dar un comportamiento específico para una clase.

Herencia (V)

```
public boolean equals(Empleado e){
     return this.idEmpleado == e.idEmpleado;
  }
…
EmpleadoEventual event =
       new EmpleadoEventual(1234567,"Manolo Robles",10);
EmpleadoEventual event2 =
       new EmpleadoEventual(1234567,"Juan Palotes",5);
System.out.println(event.equals(event2)); //Devuelve true
…
//En la clase empleado
 public String toString(){
      return "(" + idEmpleado + ") " +
                nombre + " Sueldo: " + sueldo;
  }
…
//En la clase EmpleadoEventual
public String toString(){
        return super.toString() + " Horas : " + horasTrabajadas;
  }
```

Herencia (VI)

- Clases abstractas.
  - Definen parte de la implementación, dejando a las clases ampliadas la tarea de proporcionar las implementaciones específicas de algunos o de todos sus métodos.
  - Las clases y los métodos abstractos se marcan con la palabra reservada `abstract`.
  - Una clase abstracta no es instanciable ya que no presentan toda la implementación.
  - Un método abstracto no puede ser static, ya que su implementación debe estar asociada a una instancia de la subclase.

Herencia (VII)

```
abstract class Empleado{
  ...
  abstract void calcularSueldo();
  public String toString(){
    return "Id. Empleado: " + idEmpleado + "\nNombre: " + nombre + "\nSueldo: " + sueldo;

  }
}
class EmpleadoFijo extends Empleado{
    double complemento = 0;
    ...
    void calcularSueldo(){
       this.sueldo = sueldo + complemento;
    }
    public String toString(){
       return super.toString() + "\nComplemento: " + complemento;
    }

}

class EmpleadoEventual extends Empleado{
   final float PRECIO_HORA = 30;
   private int horasTrabajadas = 0;
   ...
   void calcularSueldo(){
      sueldo = horasTrabajadas * PRECIO_HORA;
   }
   public String toString(){
      return super.toString() +"\nHoras: " + horasTrabajadas;
   }
}
```

Polimorfismo (I)

- Un método puede adoptar distintas formas dependiendo del método que lo invoque.
- La ligadura (conexión de un método con su llamada) se puede hacer en tiempo de ejecución.
  - Ligadura temprana (métodos normales, sobrecargados, `final`).
    - ✓ En tiempo de compilación y enlace, el compilador necesita saber la referencia sobre la que se aplica el método.
  - Ligadura tardía (polimorfismo).
    - ✓ La referencia al objeto que llama al método se hace en tiempo de ejecución.
    - ✓ No se sabe inicialmente el tipo de objeto que utiliza la llamada

Polimorfismo (II)

```
import java.io.*;

public class Empresa{
    static Empleado[] nomina;
    private final int NUM_EMPLEADOS = 0;

    Empresa(int n){
       nomina = new Empleado[n];
    }

    void listarEmpleados(){
       for(int i = 0;i <= nomina.length-1;i++)
          if(nomina[i] != null)
             System.out.println(i + " " + nomina[i]);
    }

    double calcularTotalSueldos(){
       double total = 0;
       for(int i = 0;i <= nomina.length -1 ; i++)
          if(nomina[i] != null){
             nomina[i].calcularSueldo();
             total += nomina[i].getSueldo();
          }
       return total;
    }
```

Polimorfismo (III)

```
public static void main(String[] args) throws IOException{
    BufferedReader teclado = new BufferedReader(new InputStreamReader(System.in));
    String tipoEmpleado;
    long id; String nom; double suel; double comp; int horas;
    Empresa  e = new Empresa(10);
    //Carga 5 empleados fijos o eventuales
    for(int i = 0; i <= 4; i++){
       //Se selecciona si el empleado es fijo o eventual
       System.out.print("Tipo de empleado (F/E):"); tipoEmpleado = teclado.readLine();
       System.out.print("Id. empleado: "); id = Long.parseLong(teclado.readLine());
       System.out.print("Nombre: "); nom = teclado.readLine();
       if(tipoEmpleado.equals("F")){
          System.out.print("Complemento: ");comp = Double.parseDouble(teclado.readLine());
          System.out.print("Sueldo: "); suel = Double.parseDouble(teclado.readLine());
          nomina[i] = new EmpleadoFijo(id,nom,comp);nomina[i].setSueldo(suel);
       }
       else{
          System.out.print("Horas trabajadas:");horas= Integer.parseInt(teclado.readLine());
          nomina[i] = new EmpleadoEventual(id,nom,horas);
       }
    }
    System.out.println("Total sueldos: " + e.calcularTotalSueldos());
    e.listarEmpleados();
    }
}
}
```

Polimorfismo (IV)

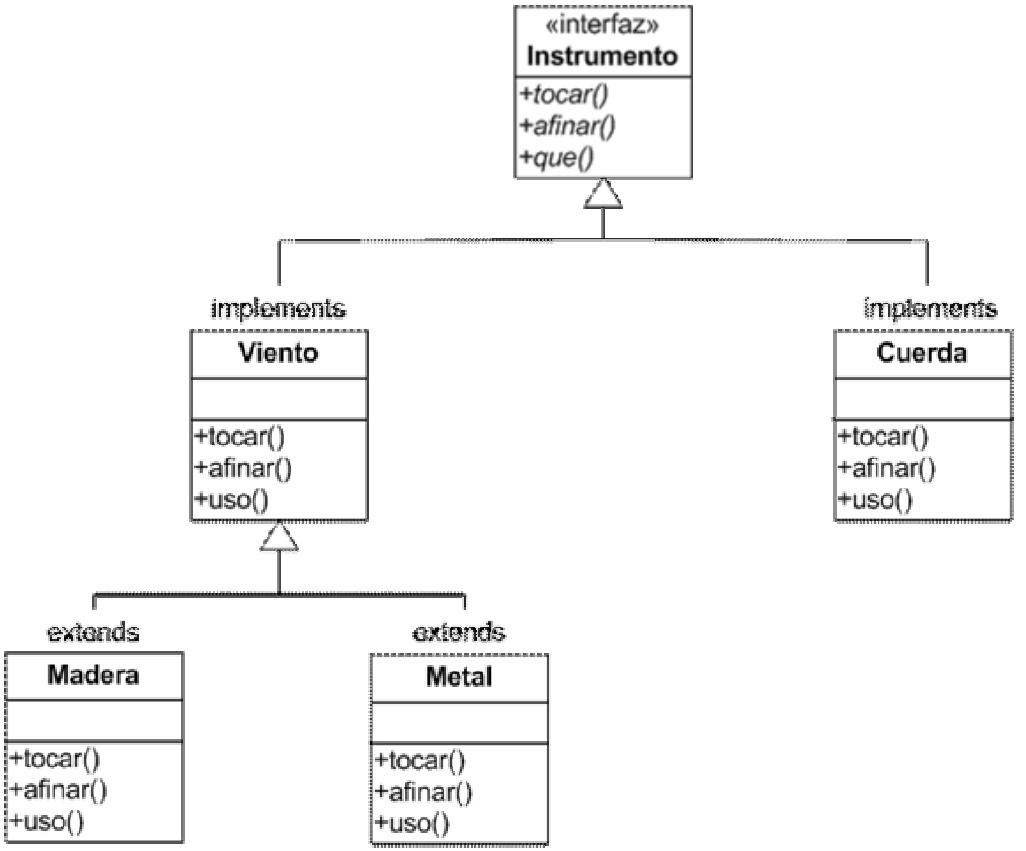
- La clase `Empresa` tiene un array de `Empleado` (superclase).
- El método `main` carga aleatoriamente y en tiempo de ejecución 5 empleados fijos o eventuales
- El método `calcularTotalSueldos` utiliza el método `calcularSueldo` de la superclase, que hace una ligadura tardía para utilizar el método adecuado de la clase ampliada.
- El método `listarEmpleados` de la clase `Empresa` también utiliza el polimorfismo para llamar al método `toString` adecuado.

Interfaces (I)

- Clase abstracta pura.
  - Incluye listas de los nombres de métodos, listas de argumentos y tipos de retorno, pero no la implementación.
  - También puede incluir atributos pero serán de clase (`static`) y constantes (`final`).
- Para declararlo se utiliza la palabra clave `interface` en vez de `class`.
  - Si se requiere acceso desde fuera del paquete puede llevar el modificador `public`.
    - ✓ En ese caso debe llevar el nombre del archivo fuente.
- Para que una clase utilice un interfaz, debe llevar la cláusula `implements`.

```
class Empleado implements Persona{…}.
```

Interfaces (II)



Interfaces (III)

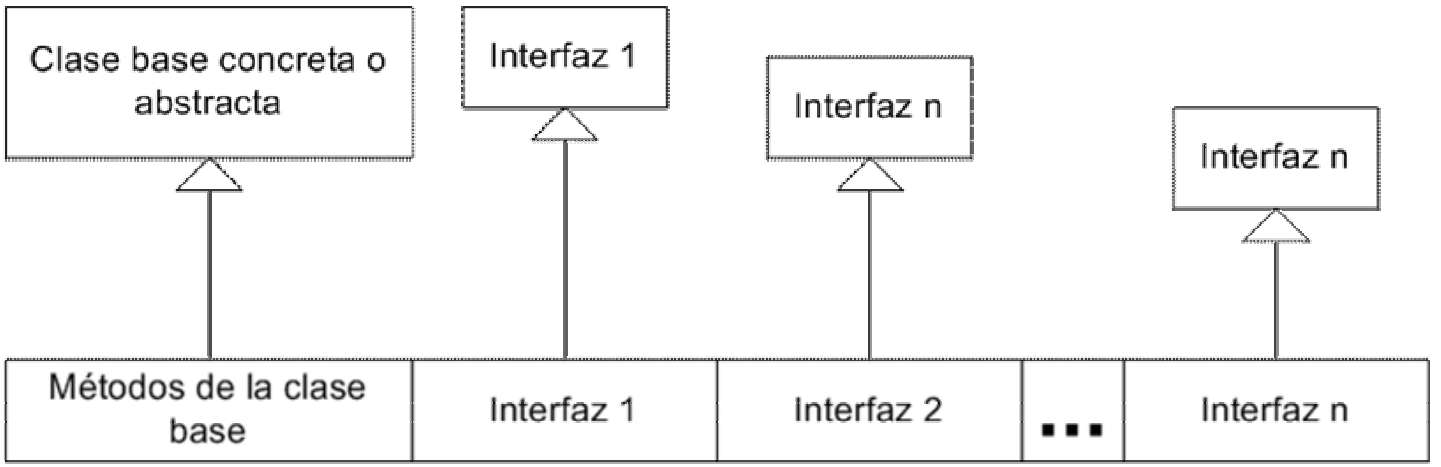
```
interface Instrumento{
   int i = 5; //estático y constante
   void tocar(); //automáticament public
   String que(); //idem
   void afinar();//idem
}
class Viento implements Instrumento{
   public void tocar(){
      System.out.println("Viento.tocar");
   }
   public String que(){
      return "Viento";
   }
   public void afinar(){
      System.out.println("Viento.afinar");
   }
}
class Cuerda implements Instrumento{
   public void tocar(){
      System.out.println("Cuerda.tocar");
   }
   public String que(){
      return "Cuerda";
   }
   public void afinar(){
      System.out.println("Cuerda.afinar");
   }
}
```

```
class Madera extends Viento{
   public void tocar(){
      System.out.println("Madera.tocar");
   }
   public void afinar(){
      System.out.println("Madera.afinar");
   }
}
class Metal extends Viento{
   public void tocar(){
      System.out.println("Metal.tocar");
   }
   public void afinar(){
      System.out.println("Metal.afinar");
   }
}
class Musica{
      static void afinar(Instrumento i){
               i.afinar();
      }
      static void afinarTodo(Instrumento[] orq){
               for(int i = 0;i<=orq.length-1;i++)
                        afinar(orq[i]);
      }
      public static void main(String args[]){
               Instrumento[] orquesta = new Instrumento[3];
               int i = 0;
               orquesta[i++] = new Cuerda();
               orquesta[i++] = new Metal();
               orquesta[i++] = new Madera();
               afinarTodo(orquesta);}}
```

Herencia múltiple (I)

- Las interfaces no tienen implementación, por lo que no tienen espacio de almacenamiento asociado.
- Es posible combinar varias interfaces en una clase para implementar la herencia múltiple.
- Una clase puede heredar implementaciones de una superclase e implementar en ella varias interfaces.
- Respecto a C++
  - No se pueden heredar atributos no `static`.
  - No se pueden heredar implementaciones de métodos.

Herencia múltiple (II)




```
class NuevaClase extends ClaseBase implements Interfaz1,Interfaz2,..,Interfazn{…}
```

Paquetes (I)

- Agrupación de clases relacionadas que se pueden distribuir de forma independiente.
  - Contienen clases, interfaces, subclases, otros subpaquetes y archivos relacionados (por ejemplo imágenes).
- Permiten:
  - Crear bibliotecas de clases.
  - Crear espacios de nombres para evitar conflictos de nombres entre los tipos.
  - Proporcionan un marco de protección.
    - ✓ Los miembros de un paquete pueden utilizar miembros de otras clases del paquete pero no accesibles desde el exterior.

Paquetes (II)

- Utilización de los paquetes.
  - Nombre completamente cualificado: nombre completo de la clase.
    - ✓ `nombrepaquete.nombreClase`

      ```
      java.util.Date d = new java.util.Date();
      ```
  - Importación de la clase:
    - ✓ `import java.util.Date;`, importa la clase `Date` del paquete.
    - ✓ `import java.util.*;`, importa todas las clases del paquete.

      ```
      Date d = new Date();
      ```
    - ✓ `import` no importa nada, sólo indica al compilador como puede formar el nombre completamente cualificado.
    - ✓ Orden de búsqueda de las clases:
      - × Busca en el tipo actual.
      - × Busca en un tipo anidado del tipo actual.
      - × Busca entre los tipo importados explicitamente (una sóla clase).
      - × Busca en otros tipos declarados del mismo paquete.
      - × Busca entre los tipos nombrados implícitamente (asterisco).

Paquetes (III)

- Incluir en un paquete.
  - Se utiliza la instrucción `package nombrepaquete;` en la primera línea no comentada del programa.
  - Para organizar los archivos `.class` en se utiliza la estructura de directorios del sistema de archivos local para codificar el nombre del paquete.
    - ✓ Para crear nombre de paquete únicos se suele utilizar el sistema de nombres de dominios de Internet para la primera parte del nombre del paquete.
- CLASSPATH ubica los directorios base donde se almacenan los paquetes.
  - Si `CLASSPATH = c:\java\clases;.` y se está utilizando el paquete `net.colimbo.util`,
  - las clases deberán ubicarse en el directorio `c:\java\clases\net\colimbo\util`.

Paquetes (IV)

- Las clases `Clase1` y `Clase2` están almacenadas en el directorio `c:\java\clases\net\colimbo\prueba`. La variable `CLASSPATH` está establecida a `CLASSPATH=.;c:\java\clases`.

```
package net.colimbo.prueba;
public class Clase1{
  public void metodo1(){
   System.out.println("Clase1.metodo1");
  }
}
```

```
package net.colimo.prueba;
public class Clase2{
  public void metodo2(){
   System.out.println("Clase2.metodo2");
  }
}
```

- La clase Prueba puede estar en cualquier otra ubicación.

```
import net.colimbo.prueba.*;
class Prueba{
   public static void main(String args[]){
        Clase1 c1 = new Clase1();
        Clase2 c2 = new Clase2();
        c1.metodo1(); c2.metodo2();
   }
}
```