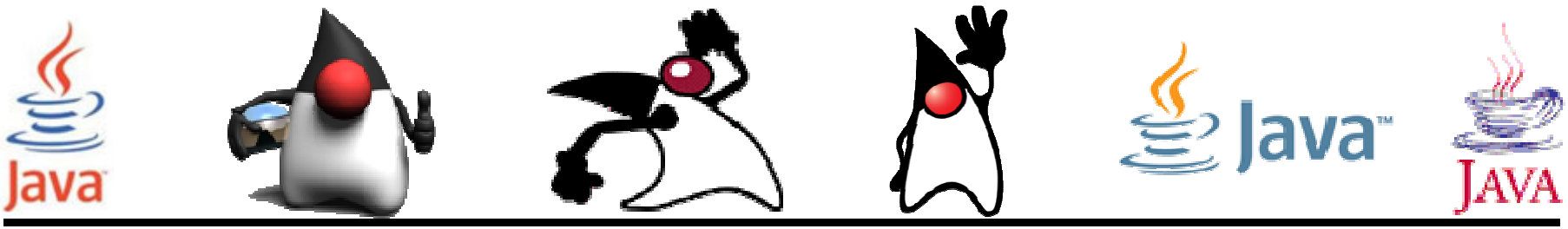


Clases Abstractas e Interfaces

Pedro Corcuera

Dpto. Matemática Aplicada y
Ciencias de la Computación

Universidad de Cantabria

corcuerp@unican.es

Objetivos

- Aprender a crear y utilizar clases y métodos abstractos
- Aprender a crear y utilizar interfaces

Índice

- Métodos y Clases Abstractos
- Qué una interface?
- Definiendo interfaces
- Implementando interfaces
- Constantes en interfaces
- Porqué usar interfaces?
- Interface vs. Clase Abstracta
- Interface como Tipo
- Herencia entre interfaces
- La interface Comparable

Métodos Abstractos

- Son métodos que no tienen implementación (body)
- Para crear un método abstracto sólo escribir la declaración del método sin el cuerpo y usar la palabra reservada `abstract`
  - Sin { }
- Ejemplo:

```
// Notese que no hay cuerpo
public abstract void algunMetodo();
```

Clase Abstracta

- Una clase abstracta es una clase que contiene uno o más métodos abstractos
- Una clase abstracta no se puede instanciar

```
// Da error de compilacion
MiClaseAbst a1 = new MiClaseAbst();
```

- Otra clase (clase concreta) tiene que proporcionar la implementación de los métodos abstractos
  - La clase concreta tiene que implementar todos los métodos abstractos de la clase abstracta para que sea usada para instanciarla
  - Las clases concretas usan la palabra reservada extends

Ejemplo de Clase Abstracta

```
public abstract class LivingThing {
  public void breath(){
    System.out.println("Living Thing breathing...");
  }
  public void eat(){
    System.out.println("Living Thing eating...");
  }
  /**
  * Abstract method walk()
  * Queremos que este metodo sea implementado
  * por una clase concreta.
  */
  public abstract void walk();
}
```

Entendiendo una Clase Abstracta

- Cuando una clase concreta extiende la clase abstracta `LivingThing`, debe implementar el método abstracto `walk()`, o también, la subclase también se convierte en una clase abstracta y por eso no puede instanciarse
- Ejemplo

```
public class Human extends LivingThing {
   public void walk(){
     System.out.println("Human walks...");
   }
}
```

Cuándo usar métodos y clases abstractas

- Los métodos abstractos son normalmente declarados donde dos o más subclases se espera que cumplan un papel similar en diferentes modos a través de diferentes implementaciones (polimorfismo)
  - Las subclases extienden la misma clase abstracta y proporcionan diferentes implementaciones para los método abstractos
- Usar clases abstractas para definir tipos amplios de comportamientos en la raíz de la jerarquía de clases y usar sus subclases para proporcionar los detalles de implementación de la clase abstracta

Qué una interface?

- Define una forma estándar y pública de especificar el comportamiento de clases (define un contrato)
- Todos lo métodos de una interface son métodos abstractos (firmas de métodos sin implementación)
- Una clase concreta debe implementar (implements) la interface, es decir, implementar todos los métodos
- Permite la implementación de clases con comportamientos comunes, sin importar su ubicación en la jerarquía de clases

Definiendo interfaces

- Sintaxis para definir una interface:

```
public interface [NombreInterface] {
// metodos sin cuerpo
}
```

- Ejemplo: interface que define relaciones entre dos objetos de acuerdo al “orden natural” de los objetos

```
public interface Relation {
  public boolean isGreater(Object a, Object b);
  public boolean isLess(Object a, Object b);
  public boolean isEqual(Object a, Object b);
}
```

Implementando interfaces

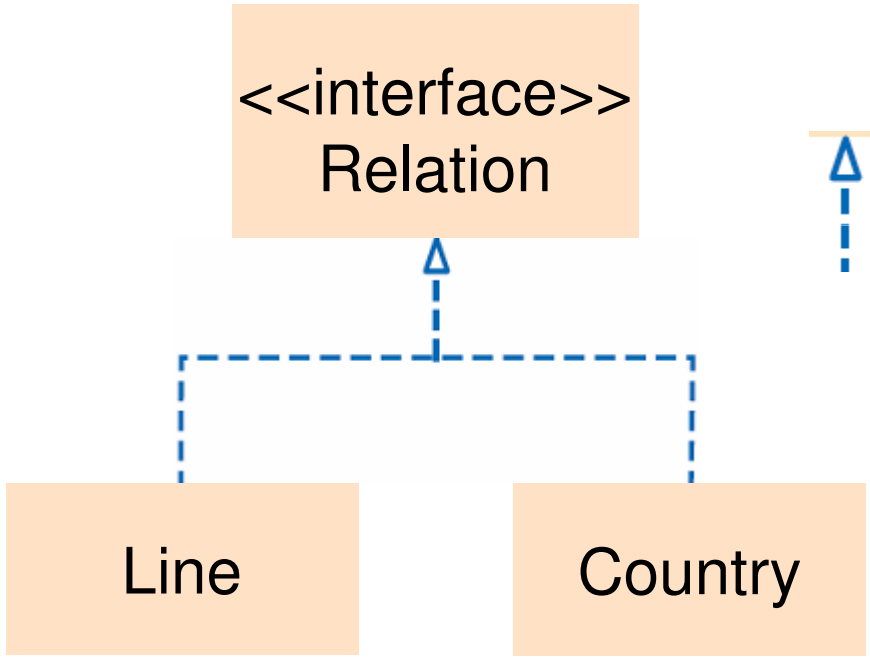
- Para crear una clase concreta que implementa una interface, se usa la palabra reservada `implements`

```
/**
* Clase Line implements Relation interface
*/
public class Line implements Relation {
  private double x1;
  private double x2;
  private double y1;
  private double y2;
  public Line(double x1, double x2, double y1, double y2){
    this.x1 = x1;
    this.x2 = x2;
    this.y1 = y1;
    this.y2 = y2;
  }
```

Implementando interfaces

```
  public double getLength(){
    double length = Math.sqrt((x2-x1)*(x2-x1) +
                    (y2-y1)* (y2-y1));  return length;
  }
  public boolean isGreater( Object a, Object b){
    double aLen = ((Line)a).getLength();
    double bLen = ((Line)b).getLength(); return (aLen > bLen);
  }
  public boolean isLess( Object a, Object b){
    double aLen = ((Line)a).getLength();
    double bLen = ((Line)b).getLength(); return (aLen < bLen);
  }
  public boolean isEqual( Object a, Object b){
    double aLen = ((Line)a).getLength();
    double bLen = ((Line)b).getLength();
    return (aLen == bLen);
  }
}
```

Diagrama de implementación de interface



La línea a trozos con una flecha se usa para indicar relaciones de *implements*

Constantes en interfaces

- Las interfaces no pueden tener variables de instancia, pero es legal especificar constantes
- Todas las variables en una interface son automáticamente `public static final` por lo que se puede omitir en la declaración

```
public interface SwingConstants {
   int NORTH = 1;
   int NORTHEAST = 2;
   int EAST = 3;
   . . .
}
```

Errores implementando interfaces

- Cuando una clase implementa una interface siempre debe implementar *todas* los métodos de la interface
- En la implementación debe declararse todos los métodos public

Porqué usar interfaces?

- Para revelar la interface de la programación de un objeto (funcionalidad del objeto) sin revelar su implementación (encapsulado)
  - La implementación puede cambiar sin afectar el llamador de la interface, que no necesita la implementación en tiempo de compilación
- Para tener implementación de métodos similares (comportamientos) en clases sin relacionar
- Para modelar herencia múltiple, imponiendo conjuntos múltiples de comportamientos a la clase

Interface vs. Clase Abstracta

- Todos los métodos de una interface son métodos abstractos mientras algunos métodos de una clase abstracta son métodos abstractos
  - Los métodos abstractos de una clase abstracta tienen el modificador abstract
- Una interfaz puede definir constantes mientras que una clase abstracta puede tener campos
- Las interfaces no tienen ninguna relación de herencia directa con una clase particular, se definen independientemente

Interface como Tipo

- La definición de una interface implica una definición de un nuevo tipo de referencia y por ello se puede usar el nombre de la interface como nombre de tipo
- Si se define una variable cuyo tipo es una interface, se le puede asignar un objeto que es una instancia de una clase que implementa la interface
- Ejemplo: la clase Person implementa PersonInterface

```
Person p1 = new Person();
PersonInterface pi1 = p1;
PersonInterface pi2 = new Person();
```

Interface y Clases: características comunes

- Interfaces y clases son tipos
  - Significa que una interface se puede usar en lugares donde una clase se puede usar
- Ejemplo: la clase Person implementa PersonInterface

```
PersonInterface pi = new Person();
```

- Tanto la interface como la clase definen métodos

Interface y Clases: características diferentes

- Todos los métodos de una interface son métodos abstractos
  - No tienen cuerpo
- No se puede crear una instancia de una interface. Ejemplo:

```
PersonInterface pi = new PersonInterface();
```

- Una interface sólo puede ser implementado por clases o extendido por otras interfaces

Relación de una interface a una clase

- Una clase concreta sólo puede extender una super clase, pero puede implementar múltiples interfaces
  - El lenguaje Java no permite herencia múltiple, pero las interfaces proporcionan una alternativa
- Todos los métodos abstractos de todas las interfaces tiene que ser implementados por la clase concreta

```
public class IndustrialStudent extends Student
             implements PersonInterface,
             OtraInterface, NesimaInterface {
  // todos los metodos abstractos de todas las
  // interfaces deben ser implementados
}
```

Herencia entre interfaces

- Las interfaces no son parte de la jerarquía de clases
- Sin embargo, las interfaces pueden tener relación de herencia entre ellas

```
public interface PersonInterface {
  void doSomething();
}
public interface StudentInterface
                  extends PersonInterface {
  void doExtraSomething();
}
```

Interface y Polimorfismo

- Las interfaces permiten polimorfismo, desde que el programa puede llamar un método de la interface y la versión apropiada del método será ejecutada dependiendo del tipo de la instancia del objeto pasado en la llamada del método de la interface

Reescribiendo una interface existente

- Si se tiene desarrollada una interface DoIt

```
public interface DoIt {
  void doSomething(int i, double x);
  int doSomethingElse(String s);
}
```

- Supongamos que posteriormente se desea añadir un tercer método en DoIt, quedando una nueva versión

```
public interface DoIt {
 void doSomething(int i, double x);
 int doSomethingElse(String s);
 boolean didItWork(int i, double x, String s);
}
```

Reescribiendo una interface existente

- Si se realiza este cambio, todas las clases que implementan la versión anterior de la interface DoIt no funcionarán porque no implementan todos los métodos de la interface
- Una solución es crear más interfaces (extendidas)

```
public interface DoItPlus extends DoIt {
 boolean didItWork(int i, double x, String s);
}
```

- Así se puede continuar usando la versión anterior o actualizar a la nueva interface

Cuándo usar una clase abstracta sobre interface

- Para métodos no abstractos se desea usarlos cuando se quiere proporcionar una implementación común para todas las subclases, reduciendo la duplicación
- Para métodos abstractos, la motivación es la misma que en la interface: imponer un comportamiento común para todas las subclases sin dictar cómo implementarla
- *Nota*: una clase concreta puede extender sólo una superclase si ésta está en la forma de clase concreta o abstracta

La interface Comparable

- La líbrería Java incluye un número importante de interfaces, entre ellas, `Comparable`
  - Requiere la implementación de un método: compareTo()
  - Se usa para comparar dos objetos
  - Es implementado por varios objetos de la API Java
  - Se puede implementar en las clases para aprovechar las potentes herramientas de Java como el ordenamiento
- Se invoca en un objeto, y se pasa otro
  - Invocado en un objeto a, devuelve valores: Negativo (a es anterior a b), Positivo (a es posterior a b), 0 (a es igual a b)

```
a.compareTo(b);
```

El tipo del parámetro de Comparable

- La interface `Comparable` usa un tipo especial de parámetro que le permite trabajar con cualquier tipo

```
public interface Comparable<T> {
  int compareTo(T other);
}
```

- El tipo <T> es un marcador de un tipo de objeto (tipos genéricos)
- La clase ArrayList usa la misma técnica con el tipo encerrado por < >

```
ArrayList<String> names = new ArrayList<String>();
```

Ejemplo de Comparable

- El método compareTo de la clase CuentaBanco compara cuentas de banco por su saldo
  - Los métodos de la interface deben ser públicos

```
public class CuentaBanco implements Comparable<CuentaBanco>
{
   . . .
   public int compareTo(CuentaBanco other)
   {
      if (balance < other.getSaldo()) { return -1; }
      if (balance > other.getSaldo()) { return 1; }
      return 0;
   }
   . . .
}
```

El parámetro es de tipo de la misma clase (CuentaBanco)

Uso de compareTo para ordenar

- El método `Arrays.sort` usa el método compareTo para ordenar los elementos del array
  - Una vez que la clase CuentaBanco implementa la interface Comparable, se puede ordenar (saldo ascendente) con el método Arrays.sort

```
CuentaBanco[] cuentas = new CuentaBanco[3];
cuentas[0] = new CuentaBanco(10000);
cuentas[1] = new CuentaBanco(0);
cuentas[2] = new CuentaBanco(2000);
Arrays.sort(cuentas);
```